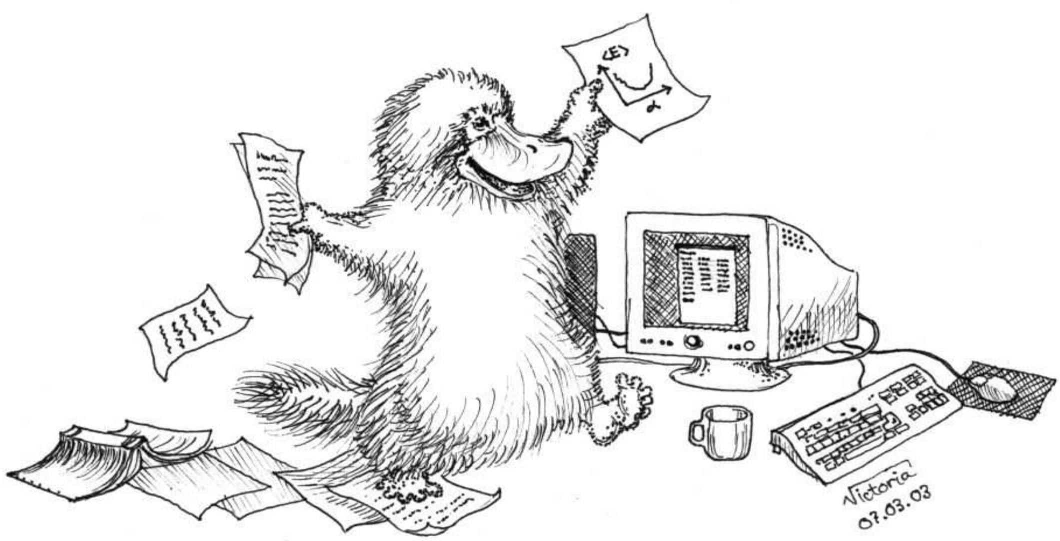
COMPUTATIONAL PHYSICS

Morten Hjorth-Jensen



University of Oslo, Fall 2008

Preface

> So, ultimately, in order to understand nature it may be necessary to have a deeper understanding of mathematical relationships. But the real reason is that the subject is enjoyable, and although we humans cut nature up in different ways, and we have different courses in different departments, such compartmentalization is really artificial, and we should take our intellectual pleasures where we find them. *Richard Feynman, The Laws of Thermodynamics.*

Why a preface you may ask? Isn't that just a mere exposition of a raison d'etre of an author's choice of material, preferences, biases, teaching philosophy etc.? To a large extent I can answer in the affirmative to that. A preface ought to be personal. Indeed, what you will see in the various chapters of these notes represents how I perceive computational physics should be taught.

This set of lecture notes serves the scope of presenting to you and train you in an algorithmic approach to problems in the sciences, represented here by the unity of three disciplines, physics, mathematics and informatics. This trinity outlines the emerging field of computational physics. Time is ripe for revising the old tale that if mathematics is the queen of sciences[1], then physics is king. Informatics ought definitely to belong among the princely.

Our insight in a physical system, combined with numerical mathematics gives us the rules for setting up an algorithm, viz a set of rules for solving a particular problem. Our understanding of the physical system under study is obviously gauged by the natural laws at play, the initial conditions, boundary conditions and other external constraints which influence the given system. Having spelled out the physics, for example in the form of a set of coupled partial differential equations, we need efficient numerical methods in order to set up the final algorithm. This algorithm is in turn coded into a computer program and executed on available computing facilities. To develop such an algorithmic approach, you will be exposed to several physics cases, spanning from the classical pendulum to quantum mechanical systems. We will also present some of the most popular algorithms from numerical mathematics used to solve a plethora of problems in the sciences. Finally we will codify these algorithms using some of the most widely used programming languages, presently C, C++ and Fortran and its most recent standard Fortran 2003[2]. However, a high-level and fully object-oriented language like Python is now emerging as a good alternative. In this text we offer an approach where one can write all programs in Python, C/C++ or Fortran. We will also show you how to develop large programs in Python interfacing C++ and/or Fortran functions for those parts of the program which CPU intensive.

Computer simulations are nowadays an integral part of contemporary basic and applied research in the sciences. Computation is becoming as important as theory and experiment. In physics, computational physics, theoretical physics and experimental physics are all equally important in our daily research and studies of physical systems. Physics is the unity of theory, experiment and computation[3]. Moreover, the ability "to compute" forms part of the essential repertoire of research scientists. Several new fields

[1] According to the German mathematician Karl Friedrich Gauss in the nineteenth century.

[2] Throughout this text we refer to Fortran 2003 as Fortran, implying the latest standard. Fortran 2008 will only add minor changes to Fortran 2003.

[3] We mentioned previously the trinity of physics, mathematics and informatics. Viewing physics as the trinity of theory, experiment and simulations is yet another example. It is obviously tempting to go beyond the sciences. History shows that triunes, trinities and for example triple deities permeate the Indo-European cultures (and probably all human cultures), from the ancient Celts and Hindus to modern days. The ancient Celts revered many such trinues, their world was divided into earth, sea and air, nature was divided in animal, vegetable and mineral and the cardinal colours were red, yellow and blue, just to mention a few. As a curious digression, it was a Gaulish Celt, Hilary, philosopher and bishop of Poitiers (AD 315-367) in his work *De Trinitate* who formulated the Holy Trinity concept of Christianity, perhaps in order to accomodate millenia of human divination practice.

within computational science have emerged and strengthened their positions in the last years, such as computational materials science, bioinformatics, computational mathematics and mechanics, computational chemistry and physics and so forth, just to mention a few. These fields underscore the importance of simulations as a means to gain novel insights into physical systems, especially for those cases where no analytical solutions can be found or an experiment is too complicated or expensive to carry out. To be able to simulate large quantal systems with many degrees of freedom such as strongly interacting electrons in a quantum dot will be of great importance for future directions in novel fields like nano-techonology. This ability often combines knowledge from many different subjects, in our case essentially from the physical sciences, numerical mathematics, computing languages, topics from high-performace computing and some knowledge of computers.

In 1999, when I started this course at the department of physics in Oslo, computational physics and computational science in general were still perceived by the majority of physicists and scientists as topics dealing with just mere tools and number crunching, and not as subjects of their own. The computational background of most students enlisting for the course on computational physics could span from dedicated hackers and computer freaks to people who basically had never used a PC. The majority of undergraduate and graduate students had a very rudimentary knowledge of computational techniques and methods. Questions like 'do you know of better methods for numerical integration than the trapezoidal rule' were not uncommon. I do happen to know of colleagues who applied for time at a supercomputing centre because they needed to invert matrices of the size of $10^4 \times 10^4$ since they were using the trapezoidal rule to compute integrals. With Gaussian quadrature this dimensionality was easily reduced to matrix problems of the size of $10^2 \times 10^2$, with much better precision.

Less than ten years later most students have now been exposed to a fairly uniform introduction to computers, basic programming skills and use of numerical exercises. Practically every undergraduate student in physics has now made a Matlab or Maple simulation of for example the pendulum, with or without chaotic motion. Nowadays most of you are familiar, through various undergraduate courses in physics and mathematics, with interpreted languages such as Maple, Matlab and/or Mathematica. In addition, the interest in scripting languages such as Python or Perl has increased considerably in recent years. The modern programmer would typically combine several tools, computing environments and programming languages. A typical example is the following. Suppose you are working on a project which demands extensive visualizations of the results. To obtain these results, that is to solve a physics problems like obtaining the density profile of Bose-Einstein condensate, you need however a program which is fairly fast when computational speed matters. In this case you would most likely write a high-performance computing program using Monte Carlo methods in languages which are taylored for that. These are represented by programming languages like Fortran and C++. However, to visualize the results you would find interpreted languages like Matlab or scripting languages like Python extremely suitable for your tasks. You will therefore end up writing for example a script in Matlab which calls a Fortran ot C++ program where the number crunching is done and then visualize the results of say a wave equation solver via Matlab's large library of visualization tools. Alternatively, you could organize everything into a Python or Perl script which does everything for you, calls the Fortran and/or C++ programs and performs the visualization in Matlab or Python. Used correctly, these tools, spanning from scripting languages to high-performance computing languages and vizualization programs, speed up your capability to solve complicated problems. Being multilingual is thus an advantage which not only applies to our globalized modern society but to computing environments as well. This text shows you how to use C++, Fortran and Pyhton as programming languages.

There is however more to the picture than meets the eye. Although interpreted languages like Matlab, Mathematica and Maple allow you nowadays to solve very complicated problems, and high-level languages like Python can be used to solve computational problems, computational speed and the capability to write an efficient code are topics which still do matter. To this end, the majority of scientists still use languages like C++ and Fortran to solve scientific problems. When you embark on a master or PhD thesis, you will most likely meet these high-performance computing languages. This course emphasizes thus the use of programming languages like Fortran, Python and C++ instead of interpreted ones like Matlab or Maple. You should however note that there are still large differences in computer time between for example numerical Python and a corresponding C++ program for many numerical applications in the physical sciences, with a code in C++ being the fastest. Loops are for example still slow in high-level languages like Python.

Computational speed is not the only reason for this choice of programming languages. Another important reason is that we feel that at a certain stage one needs to have some insights into the algorithm used, its stability conditions, possible pitfalls like loss of precision, ranges of applicability, the possibility to improve the algorithm and taylor it to special purposes etc etc. One of our major aims here is to present to you what we would dub 'the algorithmic approach', a set of rules for doing mathematics or a precise description of how to solve a problem. To device an algorithm and thereafter write a code for solving physics problems is a marvelous way of gaining insight into complicated physical systems. The algorithm you end up writing reflects in essentially all cases your own understanding of the physics and the mathematics (the way you express yourself) of the problem. We do therefore devote quite some space to the algorithms behind various functions presented in the text. Especially, insight into how errors propagate and how to avoid them is a topic we would like you to pay special attention to. Only then can you avoid problems like underflow, overflow and loss of precision. Such a control is not always achievable with interpreted languages and canned functions where the underlying algorithm and/or code is not easily accesible. Although we will at various stages recommend the use of library routines for say linear algebra[4], our belief is that one should understand what the given function does, at least to have a mere idea. With such a starting point, we strongly believe that it can be easier to develope more complicated programs on your own using Fortran, C++ or Python.

We have several other aims as well, namely:

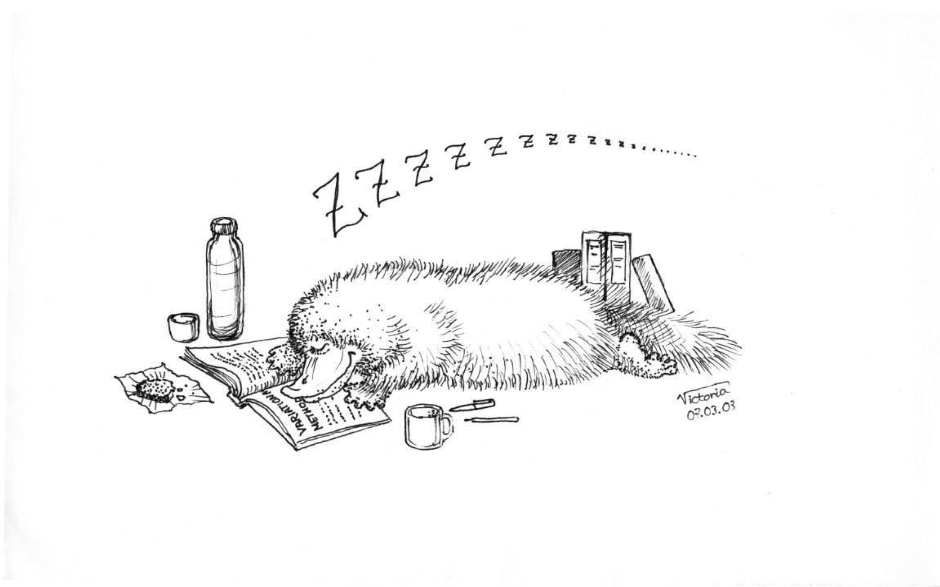
- We would like to give you an opportunity to gain a deeper understanding of the physics you have learned in other courses. In most courses one is normally confronted with simple systems which provide exact solutions and mimic to a certain extent the realistic cases. Many are however the comments like 'why can't we do something else than the particle in a box potential?'. In several of the projects we hope to present some more 'realistic' cases to solve by various numerical methods. This also means that we wish to give examples of how physics can be applied in a much broader context than it is discussed in the traditional physics undergraduate curriculum.
- To encourage you to "discover" physics in a way similar to how researchers learn in the context of research.
- Hopefully also to introduce numerical methods and new areas of physics that can be studied with the methods discussed.
- To teach structured programming in the context of doing science.
- The projects we propose are meant to mimic to a certain extent the situation encountered during a thesis or project work. You will tipically have at your disposal 2-3 weeks to solve numerically a given project. In so doing you may need to do a literature study as well. Finally, we would like you to write a report for every project.

[4] Such library functions are often taylored to a given machine's architecture and should accordingly run faster than user provided ones.

Our overall goal is to encourage you to learn about science through experience and by asking questions. Our objective is always understanding and the purpose of computing is further insight, not mere numbers! Simulations can often be considered as experiments. Rerunning a simulation need not be as costly as rerunning an experiment.

Needless to say, these lecture notes are upgraded continuously, from typos to new input. And we do always benefit from your comments, suggestions and ideas for making these notes better. It's through the scientific discourse and critics we advance. Moreover, I have benefitted immensely from many discussions with fellow colleagues and students. In particular I must mention Prof. Torgeir Engeland, whose input through the last years has considerably improved these lecture notes.

Finally, I would like to add a petit note on referencing. These notes have evolved over many years and the idea is that they should end up in the format of a web-based learning environment for doing computational science. It will be fully free and hopefully represent a much more efficient way of conveying teaching material than traditional textbooks. I have not yet settled on a specific format, so any input is welcome. At present however, it is very easy for me to upgrade and improve the material on say a yearly basis, from simple typos to adding new material. When accessing the web page of the course, you will have noticed that you can obtain all source files for the programs discussed in the text. Many people have thus written to me about how they should properly reference this material and whether they can freely use it. My answer is rather simple. You are encouraged to use these codes, modify them, include them in publications, thesis work, your lectures etc. As long as your use is part of the dialectics of science you can use this material freely. However, since many weekends have elapsed in writing several of these programs, testing them, sweating over bugs, swearing in front of a f*@?%g code which didn't compile properly ten minutes before monday morning's eight o'clock lecture etc etc, I would dearly appreciate in case you find these codes of any use, to reference them properly. That can be done in a simple way, refer to M. Hjorth-Jensen, *Lecture Notes on Computational Physics*, University of Oslo, (2008). The weblink to the course should also be included. Hope it is not too much to ask for. Enjoy!



Contents

Part I

Introduction to Numerical Methods in Physics

Chapter 1

Introduction

> ... Die Untersuchungsmethode, deren ich mich bedient habe und die auf ökonomische Probleme noch nicht angewandt wurde, macht die Lektüre der ersten Kapitel ziemlich schwierig, und es ist zu befürchten, daβ das französische Publikum, stets ungeduldig nach dem Ergebnis und begierig, den Zusammenhang zwischen den allgemeinen Grundsätzen und den Fragen zu erkennen, die es unmittelbar bewegen, sich abschrecken läβt, weil es nicht sofort weiter vordringen kann.
>
> Das ist ein Nachteil, gegen den ich nichts weiter unternehmen kann, als die nach Wahrheit strebenden Leser von vornherein darauf hinzuweisen und gefaβt zu machen. Es gibt keine Landstraβe für die Wissenschaft, und nur diejenigen haben, Aussicht, ihre lichten Höhen zu erreichen, die die Mühe nicht scheuen, ihre steilen Pfade zu erklimmen. *Karl Marx, preface to the french edition of 'Das Kapital', Vol. I*

In the physical sciences we often encounter problems of evaluating various properties of a given function $f(x)$. Typical operations are differentiation, integration and finding the roots of $f(x)$. In most cases we do not have an analytical expression for the function $f(x)$ and we cannot derive explicit formulae for derivatives etc. Even if an analytical expression is available, the evaluation of certain operations on $f(x)$ are so difficult that we need to resort to a numerical evaluation. More frequently, $f(x)$ is the result of complicated numerical operations and is thus known only at a set of discrete points and needs to be approximated by some numerical methods in order to obtain derivatives, etc etc.

The aim of these lecture notes is to give you an introduction to selected numerical methods which are encountered in the physical sciences. Several examples, with varying degrees of complexity, will be used in order to illustrate the application of these methods.

The text gives a survey over some of the most used methods in computational physics and each chapter ends with one or more applications to realistic systems, from the structure of a neutron star to the description of quantum mechanical systems through Monte-Carlo methods. Among the algorithms we discuss, are some of the top algorithms in computational science. In recent surveys by Dongarra and Sullivan [1] and Cipra [2], the list over the ten top algorithms of the 20th century include

1. The Monte Carlo method or Metropolis algorithm, devised by John von Neumann, Stanislaw Ulam, and Nicholas Metropolis, discussed in chapters 8-11.

2. The simplex method of linear programming, developed by George Dantzig.

3. Krylov Subspace Iteration method for large eigenvalue problems in particular, developed by Magnus Hestenes, Eduard Stiefel, and Cornelius Lanczos, discussed in chapter 12.

4. The Householder matrix decomposition, developed by Alston Householder and discussed in chapter 12.
5. The Fortran compiler, developed by a team lead by John Backus, codes used throughout this text.
6. The QR algorithm for eigenvalue calculation, developed by Joe Francis, discussed in chapter 12
7. The Quicksort algorithm, developed by Anthony Hoare.
8. Fast Fourier Transform, developed by James Cooley and John Tukey, discussed in chapter 19
9. The Integer Relation Detection Algorithm, developed by Helaman Ferguson and Rodney
10. The fast Multipole algorithm, developed by Leslie Greengard and Vladimir Rokhlin; (to calculate gravitational forces in an N-body problem normally requires N^2 calculations. The fast multipole method uses order N calculations, by approximating the effects of groups of distant particles using multipole expansions)

The topics we cover start with an introduction to C++ and Fortran programming combining it with a discussion on numerical precision, a point we feel is often neglected in computational science. This chapter serves also as input to our discussion on numerical derivation in chapter 3. In that chapter we introduce several programming concepts such as dynamical memory allocation and call by reference and value. Several program examples are presented in this chapter. For those who choose to program in C++ we give also an introduction to the auxiliary library Blitz++, which contains several useful classes for numerical operations on vectors and matrices. The link to Blitz++, matrices and selected algorithms for linear algebra problems are dealt with in chapter 4. Chapters 5 and 6 deal with the solution of non-linear equations and the finding of roots of polynomials and numerical interpolation, extrapolation and data fitting.

Therafter we switch to numerical integration for integrals with few dimensions, typically less than three, in chapter 7. The numerical integration chapter serves also to justify the introduction of Monte-Carlo methods discussed in chapters 8 and 9. There, a variety of applications are presented, from integration of multidimensional integrals to problems in statistical physics such as random walks and the derivation of the diffusion equation from Brownian motion. Chapter 10 continues this discussion by extending to studies of phase transitions in statistical physics. Chapter 11 deals with Monte-Carlo studies of quantal systems, with an emphasis on variational Monte Carlo methods and diffusion Monte Carlo methods. In chapter 12 we deal with eigensystems and applications to e.g., the Schrödinger equation rewritten as a matrix diagonalization problem. Problems from scattering theory are also discussed, together with the most used solution methods for systems of linear equations. Finally, we discuss various methods for solving differential equations and partial differential equations in chapters 13-15 with examples ranging from harmonic oscillations, equations for heat conduction and the time dependent Schrödinger equation. The emphasis is on various finite difference methods.

We assume that you have taken an introductory course in programming and have some familiarity with high-level or low-level and modern languages such as Java, Python, C++, Fortran 77/90/95, etc. Fortran[1] and C++ are examples of compiled low-level languages, in contrast to interpreted ones like Maple or Matlab. In such compiled languages the computer translates an entire subprogram into basic machine instructions all at one time. In an interpreted language the translation is done one statement at a time. This clearly increases the computational time expenditure. More detailed aspects of the above two programming languages will be discussed in the lab classes and various chapters of this text.

[1] With Fortran we will consistently mean Fortran 2003. There are no programming examples in Fortran 77 in this text.

There are several texts on computational physics on the market, see for example Refs. [3–10], ranging from introductory ones to more advanced ones. Most of these texts treat however in a rather cavalier way the mathematics behind the various numerical methods. We've also succumbed to this approach, mainly due to the following reasons: several of the methods discussed are rather involved, and would thus require at least a two-semester course for an introduction. In so doing, little time would be left for problems and computation. This course is a compromise between three disciplines, numerical methods, problems from the physical sciences and computation. To achieve such a synthesis, we will have to relax our presentation in order to avoid lengthy and gory mathematical expositions. You should also keep in mind that computational physics and science in more general terms consist of the combination of several fields and crafts with the aim of finding solution strategies for complicated problems. However, where we do indulge in presenting more formalism, we have borrowed heavily from several texts on mathematical analysis.

1.1 Choice of programming language

As programming language we have ended up with preferring C++, but all examples discussed in the text have their corresponding Fortran and Python programs on the webpage of this text.

Fortran (FORmula TRANslation) was introduced in 1957 and remains in many scientific computing environments the language of choice. The latest standard, Fortran [11–14], includes extensions that are familiar to users of C++. Some of the most important features of Fortran include recursive subroutines, dynamic storage allocation and pointers, user defined data structures, modules, and the ability to manipulate entire arrays. However, there are several good reasons for choosing C++ as programming language for scientific and engineering problems. Here are some:

- C++ is now the dominating language in Unix and Windows environments. It is widely available and is the language of choice for system programmers. It is very widespread for developments of non-numerical software

- The C++ syntax has inspired lots of popular languages, such as Perl, Python and Java.

- It is an extremely portable language, all Linux and Unix operated machines have a C++ compiler.

- In the last years there has been an enormous effort towards developing numerical libraries for C++. Numerous tools (numerical libraries such as MPI [15–17]) are written in C++ and interfacing them requires knowledge of C++. Most C++ and Fortran compilers compare fairly well when it comes to speed and numerical efficiency. Although Fortran 77 and C are regarded as slightly faster than C++ or Fortran, compiler improvements during the last few years have diminshed such differences. The Java numerics project has lost some of its steam recently, and Java is therefore normally slower than C++ or Fortran, see however the Java Numerics homepage for a discussion on numerical aspects of Java [18].

- Complex variables, one of Fortran's strongholds, can also be defined in the new ANSI C++ standard.

- C++ is a language which catches most of the errors as early as possible, typically at compilation time. Fortran has some of these features if one omits implicit variable declarations.

- C++ is also an object-oriented language, to be contrasted with C and Fortran. This means that it supports three fundamental ideas, namely objects, class hierarchies and polymorphism. Fortran has, through the `MODULE` declaration the capability of defining classes, but lacks inheritance, although polymorphism is possible. Fortran is then considered as an object-based programming language, to be contrasted with C++ which has the capability of relating classes to each other in a hierarchical way.

An important aspect of C++ is its richness with more than 60 keywords allowing for a good balance between object orientation and numerical efficiency. Furthermore, careful programming can results in an efficiency close to Fortran 77. The language is well-suited for large projects and has presently good standard libraries suitable for computational science projects, although many of these still lag behind the large body of libraries for numerics available to Fortran programmers. However, it is not difficult to interface libraries written in Fortran with C++ codes, if care is exercised. Other weak sides are the fact that it can be easy to write inefficient code and that there are many ways of writing the same things, adding to the confusion for beginners and professionals as well. The language is also under continuous development, which often causes portability problems.

C++ is also a difficult language to learn. Grasping the basics is rather straightforward, but takes time to master. A specific problem which often causes unwanted or odd errors is dynamic memory management.

The efficiency of C++ codes are close to those provided by Fortran. This means often that a code written in Fortran 77 can be faster, however for large numerical projects C++ and Fortran are to be preferred. If speed is an issue, one could port critical parts of the code to Fortran 77.

Future plans

Since our undergraduate curriculum has changed considerably from the beginning of Fall-2007, with the introduction of Python as programming language, the content of this course will change accordingly from the fall semester 2009. C++ and Fortran will then coexist with Python and students can choose between these three programming languages. The emphasis in the text will be on C++ programming, but how to interface C++ or Fortran programs with Python codes will also be discussed.

1.2 Designing programs

Before we proceed with a discussion of numerical methods, we would like to remind you of some aspects of program writing.

In writing a program for a specific algorithm (a set of rules for doing mathematics or a precise description of how to solve a problem), it is obvious that different programmers will apply different styles, ranging from barely readable [2] (even for the programmer) to well documented codes which can be used and extended upon by others in e.g., a project. The lack of readability of a program leads in many cases to credibility problems, difficulty in letting others extend the codes or remembering oneself what a certain statement means, problems in spotting errors, not always easy to implement on other machines, and so forth. Although you should feel free to follow your own rules, we would like to focus certain suggestions which may improve a program. What follows here is a list of our recommendations (or biases/prejudices).

[2] As an example, a bad habit is to use variables with no specific meaning, like x1, x2 etc, or names for subprograms which go like routine1, routine2 etc.

First about designing a program.

- Before writing a single line, have the algorithm clarified and understood. It is crucial to have a logical structure of e.g., the flow and organization of data before one starts writing.
- Always try to choose the simplest algorithm. Computational speed can be improved upon later.
- Try to write a as clear program as possible. Such programs are easier to debug, and although it may take more time, in the long run it may save you time. If you collaborate with other people, it reduces spending time on debugging and trying to understand what the codes do. A clear program will also allow you to remember better what the program really does!
- Implement a working code with emphasis on design for extensions, maintenance etc. Focus on the design of your code in the beginning and don't think too much about efficiency before you have a thoroughly debugged and verified your program. A rule of thumb is the so-called $80-20$ rule, 80 % of the CPU time is spent in 20 % of the code and you will experience that typically onlya small part of your code is responsible for most of the CPU expenditure. Therefore, spend most of your time in devising a good algorithm.
- The planning of the program should be from top down to bottom, trying to keep the flow as linear as possible. Avoid jumping back and forth in the program. First you need to arrange the major tasks to be achieved. Then try to break the major tasks into subtasks. These can be represented by functions or subprograms. They should accomplish limited tasks and as far as possible be independent of each other. That will allow you to use them in other programs as well.
- Try always to find some cases where an analytical solution exists or where simple test cases can be applied. If possible, devise different algorithms for solving the same problem. If you get the same answers, you may have coded things correctly or made the same error twice.
- When you have a working code, you should start thinking of the efficiency. Analyze the efficiency with a tool (profiler) to predict the CPU-intensive parts. Attack then the CPU-intensive parts after the program reproduces benchmark results.

However, although we stress that you should post-pone a discussion of the efficiency of your code to the stage when you are sure that it runs correctly, there are some simple guidelines to follow when you design the algorithm.

- Avoid lists, sets etc., when arrays can be used without too much waste of memory. Avoid also calls to functions in the innermost loop since that produces an overhead in the call.
- Heavy computation with small objects might be inefficient, e.g., vector of class complex objects
- Avoid small virtual functions (unless they end up in more than (say) 5 multiplications)
- Save object-oriented constructs for the top level of your code.
- Use taylored library functions for various operations, if possible.
- Reduce pointer-to-pointer-to....-pointer links inside loops.
- Avoid implicit type conversion, use rather the explicit keyword when declaring constructors in C++.

- Never return (copy) of an object from a function, since this normally implies a hidden allocation.

Finally, here are some of our favoured approaches for writing a code.

- Use always the standard ANSI version of the programming language. Avoid local dialects if you wish to port your code to other machines.
- Add always comments to describe what a program or subprogram does. Comment lines help you remember what you did e.g., one month ago.
- Declare all variables. Avoid totally the `IMPLICIT` statement in Fortran. The program will be more readable and help you find errors when compiling.
- Do not use `GOTO` structures in Fortran. Although all varieties of spaghetti are great culinaric temptations, spaghetti-like Fortran with many `GOTO` statements is to be avoided. Extensive amounts of time may be wasted on decoding other authors' programs.
- When you name variables, use easily understandable names. Avoid `v1` when you can use `speed_of_light`. Associatives names make it easier to understand what a specific subprogram does.
- Use compiler options to test program details and if possible also different compilers. They make errors too.
- Writing codes in C++ and Fortran may often lead to segmentation faults. This means in most cases that we are trying to access elements of an array which are not available. When developing a code it is then useful to compile with debugging options. The use of debuggers like **gdb** is something we highly recommend during the development of a program.

Chapter 2

Introduction to C++ and Fortran

Computers in the future may weigh no more than 1.5 tons. *Popular Mechanics, 1949*

There is a world market for maybe five computers. *Thomas Watson, IBM chairman, 1943*

2.1 Introduction

This chapters aims at catching two birds with a stone; to introduce to you essential features of the programming languages C++ and Fortran with a brief reminder on Python specific topics, and to stress problems like overflow, underflow, round off errors and eventually loss of precision due to the finite amount of numbers a computer can represent. The programs we discuss are taylored to these aims.

2.2 Getting started

In programming languages[1] we encounter data entities such as constants, variables, results of evaluations of functions etc. Common to these objects is that they can be represented through the type concept. There are intrinsic types and derived types. Intrinsic types are provided by the programming language whereas derived types are provided by the programmer. If one specifies the type to be for example `INTEGER (KIND=2)` for Fortran [2] or `short int/int` in C++, the programmer selects a particular date type with 2 bytes (16 bits) for every item of the class `INTEGER (KIND=2)` or `int`. Intrinsic types come in two classes, numerical (like integer, real or complex) and non-numeric (as logical and character). The general form for declaring variables is `data type name of variable` and Table 2.2 lists the standard variable declarations of C++ and Fortran (note well that there be may compiler and machine differences from the table below). An important aspect when declaring variables is their region of validity.

[1] For more detailed texts on C++ programming in engineering and science are the books by Flowers [19] and Barton and Nackman [20]. The classic text on C++ programming is the book of Bjarne Stoustrup [21]. See also the lecture notes on C++ at `http://heim.ifi.uio.no/~hpl/INF-VERK4830`. For Fortran we recommend the online lectures at `http://folk.uio.no/gunnarw/INF-VERK4820`. These web pages contain extensive references to other C++ and Fortran resources. Both web pages contain enough material, lecture notes and exercises, in order to serve as material for own studies. The Fortran 95 standard is well documented in Refs. [11–13] while the new details of Fortran 2003 can be found in Ref. [14]. The reader should note that this is not a text on C++ or Fortran . It is therefore important than one tries to find additional literature on these programming languages. Good Python texts on scientific computing are [22, 23].

[2] Our favoured display mode for Fortran statements will be capital letters for language statements and low key letters for user-defined statements. Note that Fortran does not distinguish between capital and low key letters while C++ does.

Inside a function we define a a variable through the expression `int var` or `INTEGER :: var`. The question is whether this variable is available in other functions as well, moreover where is var initialized and finally, if we call the function where it is declared, is the value conserved from one call to the other?

type in C++ and Fortran	bits	range
int/INTEGER (2)	16	-32768 to 32767
unsigned int	16	0 to 65535
signed int	16	-32768 to 32767
short int	16	-32768 to 32767
unsigned short int	16	0 to 65535
signed short int	16	-32768 to 32767
int/long int/INTEGER(4)	32	-2147483648 to 2147483647
signed long int	32	-2147483648 to 2147483647
float/REAL(4)	32	10^{-44} to 10^{+38}
double/REAL(8)	64	10^{-322} to $10e^{+308}$

Table 2.1: Examples of variable declarations for C++ and Fortran . We reserve capital letters for Fortran declaration statements throughout this text, although Fortran is not sensitive to upper or lowercase letters. Note that there are machines which allow for more than 64 bits for doubles. The ranges listed here may therefore vary.

Both C++ and Fortran operate with several types of variables and the answers to these questions depend on how we have defined for example an integer via the statement `int var`. Python on the other hand does not use variable or function types (they are not explicitely written), allowing thereby for a better potential for reuse of the code.

The following list may help in clarifying the above points:

type of variable	validity
local variables	defined within a function, only available within the scope of the function.
formal parameter	If it is defined within a function it is only available within that specific function.
global variables	Defined outside a given function, available for all functions from the point where it is defined.

In Table 2.2 we show a list of some of the most used language statements in Fortran and C++.

In addition, both C++ and Fortran allow for complex variables. In Fortran we would declare a complex variable as `COMPLEX (KIND=16):: x, y` which refers to a double with word length of 16 bytes. In C++ we would need to include a complex library through the statements

```
#include <complex>
complex<double> x, y;
```

We will discuss the above declaration **complex**<**double**> x,y; in more detail in appendix A.

Fortran	C++
Program structure	
PROGRAM something	main ()
FUNCTION something(input)	double (int) something(input)
SUBROUTINE something(inout)	
Data type declarations	
REAL (4) x, y	float x, y;
REAL(8) :: x, y	double x, y;
INTEGER :: x, y	int x,y;
CHARACTER :: name	char name;
REAL(8), DIMENSION(dim1,dim2) :: x	double x[dim1][dim2];
INTEGER, DIMENSION(dim1,dim2) :: x	int x[dim1][dim2];
LOGICAL :: x	
TYPE name	struct name {
declarations	declarations;
END TYPE name	}
POINTER :: a	double (int) *a;
ALLOCATE	new;
DEALLOCATE	delete;
Logical statements and control structure	
IF (a == b) THEN	if (a == b)
b=0	{ b=0;
ENDIF	}
DO WHILE (logical statement)	while (logical statement)
do something	{do something
ENDDO	}
IF (a>= b) THEN	if (a >= b)
b=0	{ b=0;
ELSE	else
a=0	a=0; }
ENDIF	
SELECT CASE (variable)	switch(variable)
CASE (variable=value1)	{
do something	case 1:
CASE (. . .)	variable=value1;
. . .	do something;
	break;
END SELECT	case 2:
	do something; break; . . .
	}
DO i=0, end, 1	for(i=0; i<= end; i++)
do something	{ do something ;
ENDDO	}

Table 2.2: Elements of programming syntax.

2.2.1 Scientific hello world

Our first programming encounter is the 'classical' one, found in almost every textbook on computer languages, the 'hello world' code, here in a scientific disguise. We present first the C version.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/cpp/program1.cpp

```
/* comments in C begin like this and end with */
#include <stdlib.h> /* atof function */
#include <math.h>   /* sine function */
#include <stdio.h>  /* printf function */

int main (int argc, char* argv[])
{
  double r, s;          /* declare variables */
  r = atof(argv[1]);    /* convert the text argv[1] to double */
  s = sin(r);
  printf("Hello, World! sin(%g)=%g\n", r, s);
  return 0;             /* success execution of the program */
}
```

The compiler must see a declaration of a function before you can call it (the compiler checks the argument and return types). The declaration of library functions appears in so-called header files that must be included in the program, for example **#include** <stdlib.h>.

We call three functions atof, sin, printf and these are declared in three different header files. The main program is a function called main with a return value set to an integer, returning 0 if success. The operating system stores the return value, and other programs/utilities can check whether the execution was successful or not. The command-line arguments are transferred to the main function through the statement **int** main (**int** argc, **char*** argv[]). The integer argc stands for the number of command-line arguments, set to one in our case, while argv is a vector of strings containing the command-line arguments with argv[0] containing the name of the program and argv[1], argv[2], ... are the command-line args, i.e., the number of lines of input to the program.

This means that we would run the programs as

```
mhjensen@compphys:./myprogram.exe 0.3
```

argv[0] while the text string 0.2 enters argv[1].

Here we define a floating point variable, see also below, through the keywords **float** for single precision real numbers and *double* for double precision. The function atof transforms a text (argv[1]) to a float. The sine function is declared in math.h, a library which is not automatically included and needs to be linked when computing an executable file.

With the command printf we obtain a formatted printout. The printf syntax is used for formatting output in many C-inspired languages (Perl, Python, awk, partly C++).

In C++ this program can be written as

```
// A comment line begins like this in C++ programs
using namespace std;
#include <iostream>
int main (int argc, char* argv[])
{
//  convert the text argv[1] to double using atof:
  double r = atof(argv[1]);
```

```
  double s = sin(r);
  cout << "Hello, World! sin(" << r << ")=" << s << endl;
// success
  return 0;
}
```

We have replaced the call to printf with the standard C++ function cout. The header file iostream is then needed. In addition, we don't need to declare variables like r and s at the beginning of the program. I personally prefer however to declare all variables at the beginning of a function, as this gives me a feeling of greater readability. Note that we have used the declaration **using namespace** std;. Namespace is a way to collect all functions defined in C++ libraries. If we omit this declaration on top of the program we would have to add the declaration std in front of cout or cin. Our program would then read

```
// Hello world code without using namespace std
#include <iostream>
int main (int argc, char* argv[])
{
//  convert the text argv[1] to double using atof:
  double r = atof(argv[1]);
  double s = sin(r);
  std::cout << "Hello, World! sin(" << r << ")=" << s << endl;
// success
  return 0;
}
```

Another feature which is worth noting is that we have skipped exception handlings here. In chapter 3 we discuss examples that test our input from the command line. But it is easy to add such a feature, as shown in our modified hello world program

```
// Hello world code with exception handling
using namespace std;
#include <iostream>
int main (int argc, char* argv[])
{
// Read in output file, abort if there are too few command-line arguments
    if( argc <= 1 ){
      cout << "Bad Usage: " << argv[0] <<
      " read also a number on the same line, e.g., prog.exe 0.2" << endl;
      exit(1);    //  here the program stops.
    }
//  convert the text argv[1] to double using atof:
  double r = atof(argv[1]);
  double s = sin(r);
  cout << "Hello, World! sin(" << r << ")=" << s << endl;
// success
  return 0;
}
```

Here we test that we have more than one argument. If not, the program stops and writes to screen an error message.

To run these programs, you need first to compile and link them in order to obtain an executable file under operating systems like e.g., UNIX or Linux. Before we proceed we give therefore examples on how to obtain an executable file under Linux/Unix.

In order to obtain an executable file for a C++ program, the following instructions under Linux/Unix can be used

```
c++ -c -Wall myprogram.c
c++ -o myprogram myprogram.o
```

where the compiler is called through the command `c++`. The compiler option -Wall means that a warning is issued in case of non-standard language. The executable file is in this case `myprogram`. The option `-c` is for compilation only, where the program is translated into machine code, while the `-o` option links the produced object file `myprogram.o` and produces the executable `myprogram` .

The corresponding Fortran code is

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/f90/program1.f90

```fortran
PROGRAM shw
   IMPLICIT NONE
   REAL (KIND =8) :: r           ! Input number
   REAL (KIND=8)  :: s           ! Result

!  Get a number from user
   WRITE(*,*) 'Input a number: '
   READ(*,*) r
!  Calculate the sine of the number
   s = SIN(r)
!  Write result to screen
   WRITE(*,*) 'Hello World! SINE of ', r, ' =', s
END PROGRAM shw
```

The first statement must be a program statement; the last statement must have a corresponding end program statement. Integer numerical variables and floating point numerical variables are distinguished. The names of all variables must be between 1 and 31 alphanumeric characters of which the first must be a letter and the last must not be an underscore. Comments begin with a ! and can be included anywhere in the program. Statements are written on lines which may contain up to 132 characters. The asterisks (*,*) following WRITE represent the default format for output, i.e., the output is e.g., written on the screen. Similarly, the READ(*,*) statement means that the program is expecting a line input. Note also the IMPLICIT NONE statement which we strongly recommend the use of. In many Fortran 77 programs one can find statements like IMPLICIT REAL*8(a-h,o-z), meaning that all variables beginning with any of the above letters are by default floating numbers. However, such a usage makes it hard to spot eventual errors due to misspelling of variable names. With IMPLICIT NONE you have to declare all variables and therefore detect possible errors already while compiling. I recommend strongly that you declare all variables when using Fortran.

We call the Fortran compiler (using free format) through

```
f90 -c -free myprogram.f90
f90 -o myprogram.x  myprogram.o
```

Under Linux/Unix it is often convenient to create a so-called makefile, which is a script which includes possible compiling commands, in order to avoid retyping the above lines every once and then we have made modifcations to our program. A typical makefile for the above *cc* compiling options is listed below

```
# General makefile for c - choose PROG =   name of given program
```

```
# Here we define compiler option, libraries and the  target
CC= c++ -Wall
PROG= myprogram

# Here we make the executable file
${PROG} :          ${PROG}.o
                   ${CC} ${PROG}.o -o ${PROG}

# whereas here we create the object file

${PROG}.o :       ${PROG}.cpp
                  ${CC} -c ${PROG}.cpp
```

If you name your file for 'makefile', simply type the command **make** and Linux/Unix executes all of the statements in the above makefile. Note that C++ files have the extension .cpp

For Fortran, a similar makefile is

```
# General makefile for F90 - choose PROG =   name of given program

# Here we define compiler options, libraries and the  target
F90= f90
PROG= myprogram

# Here we make the executable file
${PROG} :          ${PROG}.o
                   ${F90} ${PROG}.o -o ${PROG}

# whereas here we create the object file

${PROG}.o :       ${PROG}.f90
                  ${F90} -c ${PROG}.f
```

Finally, for the sake of completeness, we list the corresponding Python code

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/python/program1.py

```
#!/usr/bin/env python

import sys, math
# Read in a string a convert it to a float
r = float(sys.argv[1])
s = math.sin(r)
print "Hello, World! sin(%g)=%12.6e" % (r,s)
```

where we have used a formatted printout with scientific notation.

2.3 Representation of integer numbers

In Fortran a keyword for declaration of an integer is INTEGER (KIND=n) , n = 2 reserves 2 bytes (16 bits) of memory to store the integer variable wheras n = 4 reserves 4 bytes (32 bits). In Fortran, although it may be compiler dependent, just declaring a variable as INTEGER , reserves 4 bytes in memory as default.

In C++ keywords are short int , int , long int , long long int . The byte-length is compiler dependent within some limits. The GNU C++-compilers (called by gcc or g++) assign 4 bytes (32 bits) to variables declared by int and long int . Typical byte-lengths are 2, 4, 4 and 8 bytes, for the types given above. To see how many bytes are reserved for a specific variable, C++ has a library function called sizeof(type) which returns the number of bytes for type .

An example of a program declaration is

Fortran:	INTEGER (KIND=2) ::	age_of_participant
C++:	short int	age_of_participant;

Note that the (KIND=2) can be written as (2). Normally however, we will for Fortran programs just use the 4 bytes default assignment INTEGER .

In the above examples one bit is used to store the sign of the variable age_of_participant and the other 15 bits are used to store the number, which then may range from zero to $2^{15} - 1 = 32767$. This should definitely suffice for human lifespans. On the other hand, if we were to classify known fossiles by age we may need

Fortran:	INTEGER (4) ::	age_of_fossile
C++:	int	age_of_fossile;

Again one bit is used to store the sign of the variable age_of_fossile and the other 31 bits are used to store the number which then may range from zero to $2^{31} - 1 = 2.147.483.647$. In order to give you a feeling how integer numbers are represented in the computer, think first of the decimal representation of the number 417

$$417 = 4 \times 10^2 + 1 \times 10^1 + 7 \times 10^0,$$

which in binary representation becomes

$$417 = 1 \times a_n 2^n + a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \cdots + a_0 2^0,$$

where the coefficients a_k with $k = 0, \ldots, n$ are zero or one. They can be calculated through successive division by 2 and using the remainder in each division to determine the numbers a_n to a_0. A given integer in binary notation is then written as

$$a_n 2^n + a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \cdots + a_0 2^0.$$

In binary notation we have thus

$$(417)_{10} = (110100001)_2,$$

since we have

$$(110100001)_2 = 1 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0.$$

To see this, we have performed the following divisions by 2

417/2=208	remainder 1	coefficient of 2^0 is 1
208/2=104	remainder 0	coefficient of 2^1 is 0
104/2=52	remainder 0	coefficient of 2^2 is 0
52/2=26	remainder 0	coefficient of 2^3 is 0
26/2=13	remainder 0	coefficient of 2^4 is 0
13/2= 6	remainder 1	coefficient of 2^5 is 1
6/2= 3	remainder 0	coefficient of 2^6 is 0
3/2= 1	remainder 1	coefficient of 2^7 is 1
1/2= 0	remainder 1	coefficient of 2^8 is 1

We see that nine bits are sufficient to represent 417. Normally we end up using 32 bits as default for integers, meaning that our number reads

$$(417)_{10} = (00000000000000000000000110100001)_2,$$

A simple program which performs these operations is listed below. Here we employ the modulus operation (with division by 2), which in C++ is given by the a%2 operator. In Fortran we would call the function MOD(a,2) in order to obtain the remainder of a division by 2.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/cpp/program2.cpp

```
using namespace std;
#include <iostream>

int main (int argc, char* argv[])
{
   int i;
   int terms[32]; // storage of a0, a1, etc, up to 32 bits
   int number = atoi(argv[1]);
// initialise the term a0, a1 etc
   for (i=0; i < 32 ; i++){ terms[i] = 0;}
   for (i=0; i < 32 ; i++){
        terms[i] = number%2;
        number /= 2;
   }
// write out results
   cout << `` Number of bytes used= '' << sizeof(number) << endl;
   for (i=0; i < 32 ; i++){
        cout << `` Term nr: `` << i << ``Value= `` << terms[i];
        cout << endl;
   }
  return 0;
}
```

The C++ function **sizeof** yields the number of bytes reserved for a specific variable. Note also the **for** construct. We have reserved a fixed array which contains the values of a_i being 0 or 1, the remainder of a division by two. We have enforced the integer to be represented by 32 bits, or four bytes, which is the default integer representation.

Note that for 417 we need 9 bits in order to represent it in a binary notation, while a number like the number 3 is given in an 32 bits word as

$$(3)_{10} = (00000000000000000000000000000011)_2.$$

For this number 2 significant bits would be enough.

With these prerequesites in mind, it is rather obvious that if a given integer variable is beyond the range assigned by the declaration statement we may encounter problems.

If we multiply two large integers $n_1 \times n_2$ and the product is too large for the bit size allocated for that specific integer assignement, we run into an overflow problem. The most significant bits are lost and the least significant kept. Using 4 bytes for integer variables the result becomes

$$2^{20} \times 2^{20} = 0.$$

However, there are compilers or compiler options that preprocess the program in such a way that an error message like 'integer overflow' is produced when running the program. Here is a small program which may cause overflow problems when running (try to test your own compiler in order to be sure how such problems need to be handled).

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/cpp/program3.cpp

```
// Program to calculate 2**n
using namespace std;
#include <iostream>

int main()
{
   int  int1, int2, int3;
// print to screen
   cout << "Read in the exponential N for 2^N =\n";
// read from screen
   cin >> int2;
   int1 = (int) pow(2., (double) int2);
   cout << " 2^N * 2^N = " << int1*int1 << "\n";
   int3 = int1 - 1;
   cout << " 2^N*(2^N - 1) = " << int1 * int3  << "\n";
   cout << " 2^N- 1 = " << int3  << "\n";
   return 0;
}
// End: program main()
```

If we run this code with an exponent $N = 32$, we obtain the following output

```
2^N * 2^N = 0
 2^N*(2^N - 1) = -2147483648
 2^N- 1 = 2147483647
```

We notice that 2^{64} exceeds the limit for integer numbers with 32 bits. The program returns 0. This can be dangerous, since the results from the operation $2^N(2^N - 1)$ is obviously wrong. One possibility to avoid such cases is to add compilation option which flag if an overflow or underflow is reached.

2.3.1 Fortran codes

The corresponding Fortran code is

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/f90/program2.f90

```
PROGRAM binary_integer
```

```
IMPLICIT NONE
  INTEGER  i, number, terms(0:31) ! storage of a0, a1, etc, up to 32 bits,
! note array length running from 0:31. Fortran allows negative indexes as
    well.

  WRITE(*,*) 'Give a number to transform to binary notation'
  READ(*,*) number
! Initialise the terms a0, a1 etc
  terms = 0
! Fortran takes only integer loop variables
  DO i=0, 31
     terms(i) = MOD(number,2)  ! Modulus function in Fortran
     number = number/2
  ENDDO
! write out results
  WRITE(*,*) 'Binary representation '
  DO i=0, 31
    WRITE(*,*)' Term nr and value', i, terms(i)
  ENDDO

END PROGRAM binary_integer
```

and

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/f90/program3.f90

```
PROGRAM integer_exp
  IMPLICIT NONE
  INTEGER :: int1, int2, int3
  ! This is the begin of a comment line in Fortran 90
  ! Now we read from screen the variable int2
  WRITE(*,*) 'Read in the number to be exponentiated'
  READ(*,*) int2
  int1=2**int2
  WRITE(*,*) '2^N*2^N', int1*int1
  int3=int1-1
  WRITE(*,*) '2^N*(2^N-1)', int1*int3
  WRITE(*,*) '2^N-1', int3

END PROGRAM integer_exp

In Fortran the modulus division is performed by the intrinsic function \
    lstinline{MOD(number,2)}
in case of a division by $2$. The exponentation of a number is given by for
    example \lstinline{2**N}
instead of the call to the $\lstinline{pow} function in C++.
```

2.3.2 Python codes

In preparation for fall 2009

2.4 Real numbers and numerical precision

An important aspect of computational physics is the numerical precision involved. To design a good algorithm, one needs to have a basic understanding of propagation of inaccuracies and errors involved in calculations. There is no magic recipe for dealing with underflow, overflow, accumulation of errors and loss of precision, and only a careful analysis of the functions involved can save one from serious problems.

Since we are interested in the precision of the numerical calculus, we need to understand how computers represent real and integer numbers. Most computers deal with real numbers in the binary system, or octal and hexadecimal, in contrast to the decimal system that we humans prefer to use. The binary system uses 2 as the base, in much the same way that the decimal system uses 10. Since the typical computer communicates with us in the decimal system, but works internally in e.g., the binary system, conversion procedures must be executed by the computer, and these conversions involve hopefully only small roundoff errors

Computers are also not able to operate using real numbers expressed with more than a fixed number of digits, and the set of values possible is only a subset of the mathematical integers or real numbers. The so-called word length we reserve for a given number places a restriction on the precision with which a given number is represented. This means in turn, that for example floating numbers are always rounded to a machine dependent precision, typically with 6-15 leading digits to the right of the decimal point. Furthermore, each such set of values has a processor-dependent smallest negative and a largest positive value.

Why do we at all care about rounding and machine precision? The best way is to consider a simple example first. In the following example we assume that we can represent a floating number with a precision of 5 digits only to the right of the decimal point. This is nothing but a mere choice of ours, but mimicks the way numbers are represented in the machine.

Suppose we wish to evaluate the function

$$f(x) = \frac{1 - cos(x)}{sin(x)},$$

for small values of x. If we multiply the denominator and numerator with $1 + cos(x)$ we obtain the equivalent expression

$$f(x) = \frac{sin(x)}{1 + cos(x)}.$$

If we now choose $x = 0.007$ (in radians) our choice of precision results in

$$sin(0.007) \approx 0.69999 \times 10^{-2},$$

and

$$cos(0.007) \approx 0.99998.$$

The first expression for $f(x)$ results in

$$f(x) = \frac{1 - 0.99998}{0.69999 \times 10^{-2}} = \frac{0.2 \times 10^{-4}}{0.69999 \times 10^{-2}} = 0.28572 \times 10^{-2},$$

while the second expression results in

$$f(x) = \frac{0.69999 \times 10^{-2}}{1 + 0.99998} = \frac{0.69999 \times 10^{-2}}{1.99998} = 0.35000 \times 10^{-2},$$

which is also the exact result. In the first expression, due to our choice of precision, we have only one relevant digit in the numerator, after the subtraction. This leads to a loss of precision and a wrong result due to a cancellation of two nearly equal numbers. If we had chosen a precision of six leading digits, both expressions yield the same answer. If we were to evaluate $x \sim \pi$, then the second expression for $f(x)$ can lead to potential losses of precision due to cancellations of nearly equal numbers.

This simple example demonstrates the loss of numerical precision due to roundoff errors, where the number of leading digits is lost in a subtraction of two near equal numbers. The lesson to be drawn is that we cannot blindly compute a function. We will always need to carefully analyze our algorithm in the search for potential pitfalls. There is no magic recipe however, the only guideline is an understanding of the fact that a machine cannot represent correctly **all** numbers.

2.4.1 Representation of real numbers

Real numbers are stored with a decimal precision (or mantissa) and the decimal exponent range. The mantissa contains the significant figures of the number (and thereby the precision of the number). A number like $(9.90625)_{10}$ in the decimal representation is given in a binary representation by

$$(1001.11101)_2 = 1\times 2^3 + 0\times 2^2 + 0\times 2^1 + 1\times 2^0 + 1\times 2^{-1} + 1\times 2^{-2} + 1\times 2^{-3} + 0\times 2^{-4} + 1\times 2^{-5},$$

and it has an exact machine number representation since we need a finite number of bits to represent this number. This representation is however not very practical. Rather, we prefer to use a scientific notation. In the decimal system we would write a number like 9.90625 in what is called the normalized scientific notation. This means simply that the decimal point is shifted and appropriate powers of 10 are supplied. Our number could then be written as

$$9.90625 = 0.990625 \times 10^1,$$

and a real non-zero number could be generalized as

$$x = \pm r \times 10^{\mathrm{n}},$$

with a r a number in the range $1/10 \leq r < 1$. In a similar way we can represent a binary number in scientific notation as

$$x = \pm q \times 2^{\mathrm{m}},$$

with a q a number in the range $1/2 \leq q < 1$. This means that the mantissa of a binary number would be represented by the general formula

$$(0.a_{-1}a_{-2}\dots a_{-n})_2 = a_{-1}\times 2^{-1} + a_{-2}\times 2^{-2} + \dots + a_{-n}\times 2^{-n}.$$

In a typical computer, floating-point numbers are represented in the way described above, but with certain restrictions on q and m imposed by the available word length. In the machine, our number x is represented as

$$x = (-1)^s \times \mathrm{mantissa} \times 2^{\mathrm{exponent}},$$

where s is the sign bit, and the exponent gives the available range. With a single-precision word, 32 bits, 8 bits would typically be reserved for the exponent, 1 bit for the sign and 23 for the mantissa. This means that if we define a variable as

Fortran:	REAL (4) ::	size_of_fossile
C++:	float	size_of_fossile;

we are reserving 4 bytes in memory, with 8 bits for the exponent, 1 for the sign and and 23 bits for the mantissa, implying a numerical precision to the sixth or seventh digit, since the least significant digit is given by $1/2^{23} \approx 10^{-7}$. The range of the exponent goes from $2^{-128} = 2.9 \times 10^{-39}$ to $2^{127} = 3.4 \times 10^{38}$, where 128 stems from the fact that 8 bits are reserved for the exponent.

A modification of the scientific notation for binary numbers is to require that the leading binary digit 1 appears to the left of the binary point. In this case the representation of the mantissa q would be $(1.f)_2$ and $1 \leq q < 2$. This form is rather useful when storing binary numbers in a computer word, since we can always assume that the leading bit 1 is there. One bit of space can then be saved meaning that a 23 bits mantissa has actually 24 bits. This means explicitely that a binary number with 23 bits for the mantissa reads

$$(1.a_{-1}a_{-2}\dots a_{-23})_2 = 1 \times 2^0 + a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + \dots + a_{-n} \times 2^{-23}.$$

As an example, consider the 32 bits binary number

$$(10111110111101000000000000000000)_2,$$

where the first bit is reserved for the sign, 1 in this case yielding a negative sign. The exponent m is given by the next 8 binary numbers 01111101 resulting in 125 in the decimal system. However, since the exponent has eight bits, this means it has $2^8 - 1 = 255$ possible numbers in the interval $-128 \leq m \leq 127$, our final exponent is $125 - 127 = -2$ resulting in 2^{-2}. Inserting the sign and the mantissa yields the final number in the decimal representation as

$$-2^{-2}\left(1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5}\right) = (-0.4765625)_{10}.$$

In this case we have an exact machine representation with 32 bits (actually, we need less than 23 bits for the mantissa).

If our number x can be exactly represented in the machine, we call x a machine number. Unfortunately, most numbers cannot and are thereby only approximated in the machine. When such a number occurs as the result of reading some input data or of a computation, an inevitable error will arise in representing it as accurately as possible by a machine number.

A floating number x, labelled $fl(x)$ will therefore always be represented as

$$fl(x) = x(1 \pm \epsilon_x), \tag{2.1}$$

with x the exact number and the error $|\epsilon_x| \leq |\epsilon_M|$, where ϵ_M is the precision assigned. A number like $1/10$ has no exact binary representation with single or double precision. Since the mantissa

$$1.\,(a_{-1}a_{-2}\dots a_{-n})_2$$

is always truncated at some stage n due to its limited number of bits, there is only a limited number of real binary numbers. The spacing between every real binary number is given by the chosen machine precision. For a 32 bit words this number is approximately $\epsilon_M \sim 10^{-7}$ and for double precision (64 bits) we have $\epsilon_M \sim 10^{-16}$, or in terms of a binary base as 2^{-23} and 2^{-52} for single and double precision, respectively.

2.4.2 Machine numbers

To understand that a given floating point number can be written as in Eq. (2.1), we assume for the sake of simplicity that we work with real numbers with words of length 32 bits, or four bytes. Then a given number x in the binary representation can be represented as

$$x = (1.a_{-1}a_{-2}\dots a_{-23}a_{-24}a_{-25}\dots)_2 \times 2^n,$$

or in a more compact form

$$x = r \times 2^n,$$

with $1 \leq r < 2$ and $-126 \leq n \leq 127$ since our exponent is defined by eight bits.

In most cases there will not be an exact machine representation of the number x. Our number will be placed between two exact 32 bits machine numbers x_- and x_+. Following the discussion of Kincaid and Cheney [24] these numbers are given by

$$x_- = (1.a_{-1}a_{-2}\dots a_{-23})_2 \times 2^n,$$

and

$$x_+ = \left((1.a_{-1}a_{-2}\dots a_{-23}))_2 + 2^{-23}\right) \times 2^n.$$

If we assume that our number x is closer to x_- we have that the absolute error is constrained by the relation

$$|x - x_-| \leq \frac{1}{2}|x_+ - x_-| = \frac{1}{2} \times 2^{n-23} = 2^{n-24}.$$

A similar expression can be obtained if x is closer to x_+. The absolute error conveys one type of information. However, we may have cases where two equal absolute errors arise from rather different numbers. Consider for example the decimal numbers $a = 2$ and $\overline{a} = 2.001$. The absolute error between these two numbers is 0.001. In a similar way, the two decimal numbers $b = 2000$ and $\overline{b} = 2000.001$ give exactly the same absolute error. We note here that $\overline{b} = 2000.001$ has more leading digits than b.

If we compare the relative errors

$$\frac{|a - \overline{a}|}{|a|} = 1.0 \times 10^{-3}, \qquad \frac{|b - \overline{b}|}{|b|} = 1.0 \times 10^{-6},$$

we see that the relative error in b is much smaller than the relative error in a. We will see below that the relative error is intimately connected with the number of leading digits in the way we approximate a real number. The relative error is therefore the quantity of interest in scientific work. Information about the absolute error is normally of little use in the absence of the magnitude of the quantity being measured.

We define then the relative error for x as

$$\frac{|x - x_-|}{|x|} \leq \frac{2^{n-24}}{r \times 2^n} = \frac{1}{q} \times 2^{-24} \leq 2^{-24}.$$

Instead of using x_- and x_+ as the machine numbers closest to x, we introduce the relative error

$$\frac{|x - \overline{x}|}{|x|} \leq 2^{n-24},$$

with $\overline{x}$ being the machine number closest to x. Defining

$$\epsilon_x = \frac{\overline{x} - x}{x},$$

we can write the previous inequality

$$fl(x) = x(1 + \epsilon_x)$$

where $|\epsilon_x| \leq \epsilon_M = 2^{-24}$ for variables of length 32 bits. The notation $fl(x)$ stands for the machine approximation of the number x. The number ϵ_M is given by the specified machine precision, approximately 10^{-7} for single and 10^{-16} for double precision, respectively.

There are several mathematical operations where an eventual loss of precision may appear. A subraction, especially important in the definition of numerical derivatives discussed in chapter 3 is one important operation. In the computation of derivatives we end up subtracting two nearly equal quantities. In case of such a subtraction $a = b - c$, we have

$$fl(a) = fl(b) - fl(c) = a(1 + \epsilon_a),$$

or

$$fl(a) = b(1 + \epsilon_b) - c(1 + \epsilon_c),$$

meaning that

$$fl(a)/a = 1 + \epsilon_b \frac{b}{a} - \epsilon_c \frac{c}{a},$$

and if $b \approx c$ we see that there is a potential for an increased error in the machine representation of $fl(a)$. This is because we are subtracting two numbers of equal size and what remains is only the least significant part of these numbers. This part is prone to roundoff errors and if a is small we see that (with $b \approx c$)

$$\epsilon_a \approx \frac{b}{a}(\epsilon_b - \epsilon_c),$$

can become very large. The latter equation represents the relative error of this calculation. To see this, we define first the absolute error as

$$|fl(a) - a|,$$

whereas the relative error is

$$\frac{|fl(a) - a|}{a} \leq \epsilon_a.$$

The above subraction is thus

$$\frac{|fl(a) - a|}{a} = \frac{|fl(b) - f(c) - (b - c)|}{a},$$

yielding

$$\frac{|fl(a) - a|}{a} = \frac{|b\epsilon_b - c\epsilon_c|}{a}.$$

An interesting question is then how many significant binary bits are lost in a subtraction $a = b - c$ when we have $b \approx c$. The loss of precision theorem for a subtraction $a = b - c$ states that [24]: *if b and c are positive normalized floating-point binary machine numbers with* $b > c$ *and*

$$2^{-r} \leq 1 - \frac{c}{b} \leq 2^{-s}, \tag{2.2}$$

then at most r *and at least* s *significant binary bits are lost in the subtraction* $b - c$. For a proof of this statement, see for example Ref. [24].

But even additions can be troublesome, in particular if the numbers are very different in magnitude. Consider for example the seemingly trivial addition $1 + 10^{-8}$ with 32 bits used to represent the various variables. In this case, the information containing in 10^{-8} is simply lost in the addition. When we perform the addition, the computer equates first the exponents of the two numbers to be added. For 10^{-8} this has however catastrophic consequences since in order to obtain an exponent equal to 10^0, bits in the mantissa are shifted to the right. At the end, all bits in the mantissa are zeros.

This means in turn that for calculations involving real numbers (if we omit the discussion on overflow and underflow) we need to carefully understand the behavior of our algorithm, and test all possible cases where round-off errors and loss of precision can arise. Other cases which may cause serious problems are singularities of the type $0/0$ which may arise from functions like $sin(x)/x$ as $x \to 0$. Such problems may also need the restructuring of the algorithm.

2.5 Programming examples on loss of precision and round-off errors

2.5.1 Algorithms for e^{-x}

In order to illustrate the above problems, we discuss here some famous and perhaps less famous problems, including a discussion on specific programming features as well.

We start by considering three possible algorithms for computing e^{-x}:

1. by simply coding
$$e^{-x} = \sum_{n=0}^{\infty}(-1)^n \frac{x^n}{n!}$$
2. or to employ a recursion relation for
$$e^{-x} = \sum_{n=0}^{\infty} s_n = \sum_{n=0}^{\infty}(-1)^n \frac{x^n}{n!}$$
using
$$s_n = -s_{n-1}\frac{x}{n},$$
3. or to first calculate
$$e^{x} = \sum_{n=0}^{\infty} s_n$$
and thereafter taking the inverse
$$e^{-x} = \frac{1}{e^x}$$

Below we have included a small program which calculates

$$e^{-x} = \sum_{n=0}^{\infty}(-1)^n \frac{x^n}{n!},$$

for x-values ranging from 0 to 100 in steps of 10. When doing the summation, we can always define a desired precision, given below by the fixed value for the variable TRUNCATION$= 1.0E-10$, so that for a certain value of $x > 0$, there is always a value of $n = N$ for which the loss of precision in terminating the series at $n = N$ is always smaller than the next term in the series $\frac{x^N}{N!}$. The latter is implemented through the while{...} statement.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/cpp/program4.cpp

```
// Program to calculate function exp(-x)
// using straightforward summation with differing  precision
using namespace std;
#include <iostream>
// type float:  32 bits precision
// type double: 64 bits precision
#define    TYPE          double
#define    PHASE(a)      (1 - 2 * (abs(a) % 2))
#define    TRUNCATION    1.0E-10
// function declaration
```

```
TYPE factorial(int);

int main()
{
   int    n;
   TYPE   x, term, sum;
   for(x = 0.0; x < 100.0; x += 10.0)  {
     sum  = 0.0;                 //initialization
     n    = 0;
     term = 1;
     while(fabs(term) > TRUNCATION)  {
         term =  PHASE(n) * (TYPE) pow((TYPE) x,(TYPE) n) / factorial(n);
         sum += term;
         n++;
     }  // end of while() loop
     cout << `` x =’’ << x << `` exp = `` << exp(-x) << `` series = `` <<
         sum;
     cout  << `` number of terms = " << n << endl;
   } // end of for() loop
   return 0;
} // End: function main()


//      The function factorial()
//      calculates and returns n!

TYPE factorial(int n)
{
   int  loop;
   TYPE fac;
   for(loop = 1, fac = 1.0; loop <= n; loop++)  {
      fac *= loop;
   }
   return fac;
} // End: function factorial()
```

There are several features to be noted[3]. First, for low values of x, the agreement is good, however for larger x values, we see a significant loss of precision. Secondly, for $x = 70$ we have an overflow problem, represented (from this specific compiler) by NaN (not a number). The latter is easy to understand, since the calculation of a factorial of the size 171! is beyond the limit set for the double precision variable factorial. The message NaN appears since the computer sets the factorial of 171 equal to zero and we end up having a division by zero in our expression for e^{-x}.

The overflow problem can be dealt with via a recurrence formula[4] for the terms in the sum, so that we avoid calculating factorials. A simple recurrence formula for our equation

$$\exp{(-x)} = \sum_{n=0}^{\infty} s_n = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!},$$

[3]Note that different compilers may give different messages and deal with overflow problems in different ways.

[4]Recurrence formulae, in various disguises, either as ways to represent series or continued fractions, are among the most commonly used forms for function approximation. Examples are Bessel functions, Hermite and Laguerre polynomials, discussed for example in chapter 7.

x	$\exp(-x)$	Series	Number of terms in series
0.0	0.100000E+01	0.100000E+01	1
10.0	0.453999E-04	0.453999E-04	44
20.0	0.206115E-08	0.487460E-08	72
30.0	0.935762E-13	-0.342134E-04	100
40.0	0.424835E-17	-0.221033E+01	127
50.0	0.192875E-21	-0.833851E+05	155
60.0	0.875651E-26	-0.850381E+09	171
70.0	0.397545E-30	NaN	171
80.0	0.180485E-34	NaN	171
90.0	0.819401E-39	NaN	171
100.0	0.372008E-43	NaN	171

Table 2.3: Result from the brute force algorithm for $\exp(-x)$.

is to note that

$$s_n = -s_{n-1}\frac{x}{n},$$

so that instead of computing factorials, we need only to compute products. This is exemplified through the next program.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/cpp/program5.cpp

```
// program to compute exp(-x) without factorials
using namespace std;
#include <iostream>
#define TRUNCATION    1.0E-10

int main()
{
   int        loop, n;
   double     x, term, sum;

   for(loop = 0; loop <= 100; loop += 10){
     x    = (double) loop;          // initialization
     sum  = 1.0;
     term = 1;
     n    = 1;
     while(fabs(term) > TRUNCATION){
         term *= -x/((double) n);
         sum  += term;
         n++;
     } // end while loop
     cout << ``x ='' << x << ``exp = `` << exp(-x) << ``series = `` << sum;
     cout  << ``number of terms = " << n << endl;
   } // end of for loop
}  //    End: function main()
```

In this case, we do not get the overflow problem, as can be seen from the large number of terms. Our

x	$\exp(-x)$	Series	Number of terms in series
0.000000	0.10000000E+01	0.10000000E+01	1
10.000000	0.45399900E-04	0.45399900E-04	44
20.000000	0.20611536E-08	0.56385075E-08	72
30.000000	0.93576230E-13	-0.30668111E-04	100
40.000000	0.42483543E-17	-0.31657319E+01	127
50.000000	0.19287498E-21	0.11072933E+05	155
60.000000	0.87565108E-26	-0.33516811E+09	182
70.000000	0.39754497E-30	-0.32979605E+14	209
80.000000	0.18048514E-34	0.91805682E+17	237
90.000000	0.81940126E-39	-0.50516254E+22	264
100.000000	0.37200760E-43	-0.29137556E+26	291

Table 2.4: Result from the improved algorithm for $\exp(-x)$.

results do however not make much sense for larger values of x. Decreasing the truncation test will not help! (try it). This is a much more serious problem.

In order better to understand this problem, let us consider the case of $x = 20$, which already differs largely from the exact result. Writing out each term in the summation, we obtain the largest term in the sum appears at $n = 19$, with a value that equals -43099804. However, for $n = 20$ we have almost the same value, but with an interchanged sign. It means that we have an error relative to the largest term in the summation of the order of $43099804 \times 10^{-10} \approx 4 \times 10^{-2}$. This is much larger than the exact value of 0.21×10^{-8}. The large contributions which may appear at a given order in the sum, lead to strong roundoff errors, which in turn is reflected in the loss of precision. We can rephrase the above in the following way: Since $\exp(-20)$ is a very small number and each term in the series can be rather large (of the order of 10^8, it is clear that other terms as large as 10^8, but negative, must cancel the figures in front of the decimal point and some behind as well. Since a computer can only hold a fixed number of significant figures, all those in front of the decimal point are not only useless, they are crowding out needed figures at the right end of the number. Unless we are very careful we will find ourselves adding up series that finally consists entirely of roundoff errors! An analysis of the contribution to the sum from various terms shows that the relative error made can be huge. This results in an unstable computation, since small errors made at one stage are magnified in subsequent stages.

To this specific case there is a simple cure. Noting that $\exp(x)$ is the reciprocal of $\exp(-x)$, we may use the series for $\exp(x)$ in dealing with the problem of alternating signs, and simply take the inverse. One has however to beware of the fact that $\exp(x)$ may quickly exceed the range of a double variable.

2.5.2 Fortran codes

The Fortran programs are rather similar in structure to the C++ program.

In Fortran Real numbers are written as 2.0 rather than 2 and declared as REAL (KIND=8) or REAL (KIND=4) for double or single precision, respectively. In general we discorauge the use of single precision in scientific computing, the achieved precision is in general not good enough. Fortran uses a do construct to have the computer execute the same statements more than once. Note also that Fortran does not allow floating numbers as loop variables. In the example below we use both a do construct for the loop over x and a DO WHILE construction for the truncation test, as in the C++ program. One could altrenatively use the EXIT statement inside a do loop. Fortran has also if statements as in C++. The IF construct allows the execution of a sequence of statements (a block) to depend on a condition. The if construct is a compound statement and begins with IF ... THEN and ends with ENDIF. Examples of more general IF constructs using ELSE and ELSEIF statements are given in other program examples. Another feature to observe is the CYCLE command, which allows a loop variable to start at a new value.

Subprograms are called from the main program or other subprograms. In the C++ codes we declared a function TYPE factorial(**int**);. Subprograms are always called functions in C++. If we declare it with **void** is has the same meaning as subroutines in Fortran,. Subroutines are used if we have more than one return value. In the example below we compute the factorials using the function factorial . This function receives a dummy argument n. INTENT(IN) means that the dummy argument cannot be changed within the subprogram. INTENT(OUT) means that the dummy argument cannot be used within the subprogram until it is given a value with the intent of passing a value back to the calling program. The statement INTENT(INOUT) means that the dummy argument has an initial value which is changed and passed back to the calling program. We recommend that you use these options when calling subprograms. This allows better control when transfering variables from one function to another. In chapter 3 we discuss call by value and by reference in C++. Call by value does not allow a called function to change the value of a given variable in the calling function. This is important in order to avoid unintentional changes of variables when transfering data from one function to another. The INTENT construct in Fortran allows such a control. Furthermore, it increases the readability of the program.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/f90/program4.f90

```fortran
! In this module you can define for example global constants
MODULE constants
  ! definition of variables for double precisions and complex variables
  INTEGER, PARAMETER :: dp = KIND(1.0D0)
  INTEGER, PARAMETER :: dpc = KIND((1.0D0,1.0D0))
  ! Global Truncation parameter
  REAL(DP), PARAMETER, PUBLIC ::  truncation=1.0E-10
END MODULE constants

! Here you can include specific functions which can be used by
! many subroutines or functions

MODULE functions

CONTAINS
  REAL(DP) FUNCTION factorial(n)
    USE CONSTANTS
    INTEGER, INTENT(IN) :: n
    INTEGER :: loop

    factorial = 1.0_dp
    IF ( n > 1 ) THEN
       DO loop = 2, n
          factorial=factorial*loop
       ENDDO
    ENDIF
  END FUNCTION factorial

END MODULE functions
```

```
!  Main program starts here
PROGRAM exp_prog
   USE constants
   USE functions
   IMPLICIT NONE
   REAL (DP) :: x, term, final_sum
   INTEGER :: n, loop_over_x

   !  loop over x-values
   DO loop_over_x=0, 100, 10
       x=loop_over_x
       !  initialize the EXP sum
       final_sum= 0.0_dp; term = 1.0_dp; n = 0
       DO WHILE ( ABS(term) > truncation)
           term = ((-1.0_dp)**n)*(x**n)/ factorial(n)
           final_sum=final_sum+term
           n=n+1
       ENDDO
       !  write the argument x, the exact value, the computed value and n
       WRITE(*,*) x ,EXP(-x), final_sum, n
   ENDDO

END PROGRAM exp_prog
```

The **MODULE** declaration in Fortran allows one to place functions like the one which calculates the factorials. Note also the usage of the module **constants** where we define double and complex variables. If one wishes to switch to another precision, one just needs to change the declaration in one part of the program only. This hinders possible errors which arise if one has to change variable declarations in every function and subroutine. In addition we have defined a global variable **truncation** which is accessible to all functions which have the **USE** constants declaration. These declarations have to come before any variable declarations and **IMPLICIT NONE** statement.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/f90/program5.f90

```
! In this module you can define for example global constants
MODULE constants
   ! definition of variables for double precisions and complex variables
   INTEGER,  PARAMETER :: dp = KIND(1.0D0)
   INTEGER, PARAMETER :: dpc = KIND((1.0D0,1.0D0))
   ! Global Truncation parameter
   REAL(DP), PARAMETER, PUBLIC ::  truncation=1.0E-10
END MODULE constants

PROGRAM improved_exp
   USE constants
   IMPLICIT NONE
   REAL (dp) :: x, term, final_sum
   INTEGER  :: n, loop_over_x

   !  loop over x-values, no floats as loop variables
   DO loop_over_x=0, 100, 10
       x=loop_over_x
       !  initialize the EXP sum
```

```
      final_sum=1.0 ; term=1.0 ; n = 1
      DO WHILE ( ABS(term) > truncation)
         term = -term*x/FLOAT(n)
         final_sum=final_sum+term
         n=n+1
      ENDDO
      ! write the argument x, the exact value, the computed value and n
      WRITE(*,*) x ,EXP(-x), final_sum, n
   ENDDO

END PROGRAM improved_exp
```

2.5.3 Further examples

Summing $1/n$

Let us look at another roundoff example which may surprise you more. Consider the series

$$s_1 = \sum_{n=1}^{N} \frac{1}{n},$$

which is finite when N is finite. Then consider the alternative way of writing this sum

$$s_2 = \sum_{n=N}^{1} \frac{1}{n},$$

which when summed analytically should give $s_2 = s_1$. Because of roundoff errors, numerically we will get $s_2 \neq s_1$! Computing these sums with single precision for $N = 1.000.000$ results in $s_1 = 14.35736$ while $s_2 = 14.39265$! Note that these numbers are machine and compiler dependent. With double precision, the results agree exactly, however, for larger values of N, differences may appear even for double precision. If we choose $N = 10^8$ and employ double precision, we get $s_1 = 18.9978964829915355$ while $s_2 = 18.9978964794618506$, and one notes a difference even with double precision.

This example demonstrates two important topics. First we notice that the chosen precision is important, and we will always recommend that you employ double precision in all calculations with real numbers. Secondly, the choice of an appropriate algorithm, as also seen for e^{-x}, can be of paramount importance for the outcome.

The standard algorithm for the standard deviation

Yet another example is the calculation of the standard deviation σ when σ is small compared to the average value $\overline{x}$. Below we illustrate how one of the most frequently used algorithms can go wrong when single precision is employed.

However, before we proceed, let us define σ and $\overline{x}$. Suppose we have a set of N data points, represented by the one-dimensional array $x(i)$, for $i = 1, N$. The average value is then

$$\overline{x} = \frac{\sum_{i=1}^{N} x(i)}{N},$$

while

$$\sigma = \sqrt{\frac{\sum_i x(i)^2 - \overline{x} \sum_i x(i)}{N-1}}.$$

Let us now assume that

$$x(i) = i + 10^5,$$

and that $N = 127$, just as a mere example which illustrates the kind of problems which can arise when the standard deviation is small compared with the mean value $\overline{x}$.

The standard algorithm computes the two contributions to σ separately, that is we sum $\sum_i x(i)^2$ and subtract thereafter $\overline{x}\sum_i x(i)$. Since these two numbers can become nearly equal and large, we may end up in a situation with potential loss of precision as an outcome.

The second algorithm on the other hand computes first $x(i) - \overline{x}$ and then squares it when summing up. With this recipe we may avoid having nearly equal numbers which cancel.

Using single precision results in a standard deviation of $\sigma = 40.05720139$ for the first and most used algorithm, while the exact answer is $\sigma = 36.80579758$, a number which also results from the above second algorithm. With double precision, the two algorithms result in the same answer.

The reason for such a difference resides in the fact that the first algorithm includes the subtraction of two large numbers which are squared. Since the average value for this example is $\overline{x} = 100063.00$, it is easy to see that computing $\sum_i x(i)^2 - \overline{x}\sum_i x(i)$ can give rise to very large numbers with possible loss of precision when we perform the subtraction. To see this, consider the case where $i = 64$. Then we have

$$x_{64}^2 - \overline{x}x_{64} = 100352,$$

while the exact answer is

$$x_{64}^2 - \overline{x}x_{64} = 100064!$$

You can even check this by calculating it by hand.

The second algorithm computes first the difference between $x(i)$ and the average value. The difference gets thereafter squared. For the second algorithm we have for $i = 64$

$$x_{64} - \overline{x} = 1,$$

and we have no potential for loss of precision.

The standard text book algorithm is expressed through the following program, where we have also added the second algorithm

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/cpp/program6.cpp

```
// program to calculate the mean and standard deviation of
// a user created data set stored in array x[]
using namespace std;
#include <iostream>
int main()
{
    int      i;
    float    sum, sumsq2, xbar, sigma1, sigma2;
    // array declaration with fixed dimension
    float    x[127];
    //  initialise the data set
    for ( i=0; i < 127 ; i++){
        x[i] = i + 100000.;
    }
    //  The variable sum is just the sum over all elements
    //  The variable sumsq2 is the sum over x^2
    sum=0.;
```

```
    sumsq2=0.;
    //  Now we use the text book algorithm
    for ( i=0; i < 127; i++){
        sum += x[i];
        sumsq2 += pow((double) x[i],2.);
    }
    //  calculate the average and sigma
    xbar=sum/127.;
    sigma1=sqrt((sumsq2-sum*xbar)/126.);
    /*
    **  Here comes the second algorithm where we evaluate
    **  separately first the average and thereafter the
    **  sum which defines the standard deviation. The average
    **  has already been evaluated through xbar
    */
    sumsq2=0.;
    for ( i=0; i < 127; i++){
       sumsq2 += pow( (double) (x[i]-xbar),2.);
    }
    sigma2=sqrt(sumsq2/126.);
    cout << "xbar = `` << xbar << ``sigma1 = `` << sigma1 << ``sigma2 = ``
       << sigma2;
    cout << endl;
    return 0;
}// End: function main()
```

The corresponding Fortran program is given below.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/f90/program6.f90

```
PROGRAM standard_deviation
  IMPLICIT NONE
  REAL (KIND = 4) :: sum, sumsq2, xbar
  REAL (KIND = 4) :: sigma1, sigma2
  REAL (KIND = 4), DIMENSION (127) :: x
  INTEGER :: i

  x=0;
  DO i=1, 127
     x(i) = i + 100000.
  ENDDO
  sum=0.; sumsq2=0.
  !       standard deviation calculated with the first algorithm
  DO i=1, 127
     sum = sum +x(i)

     sumsq2 = sumsq2+x(i)**2
  ENDDO
  !       average
  xbar=sum/127.
  sigma1=SQRT((sumsq2-sum*xbar)/126.)
  !       second algorithm to evaluate the standard deviation
  sumsq2=0.
  DO i=1, 127
```

arithmetic operators		relation operators	
operator	effect	operator	effect
$-$	Subtraction	$>$	Greater than
$+$	Addition	$>=$	Greater or equal
$*$	Multiplication	$<$	Less than
$/$	Division	$<=$	Less or equal
$\%$ or MOD	Modulus division	$==$	Equal
$--$	Decrement	$!=$	Not equal
$++$	Increment		

Table 2.5: Relational and arithmetic operators. The relation operators act between two operands. Note that the increment and decrement operators $++$ and $--$ are not available in Fortran .

Logical operators		
C++	Effect	Fortran
0	False value	.FALSE.
1	True value	.TRUE.
!x	Logical negation	.NOT.x
x&& y	Logical AND	x.AND.y
x||y	Logical inclusive OR	x.OR.y

Table 2.6: List of logical operators in C++ and Fortran .

```
      sumsq2=sumsq2+(x(i)-xbar)**2
   ENDDO
   sigma2=SQRT(sumsq2/126.)
   WRITE(*,*) xbar, sigma1, sigma2

END PROGRAM standard_deviation
```

2.6 *Additional features of C++ and Fortran*

2.6.1 *Operators in C++*

In the previous program examples we have seen several types of operators. In the tables below we summarize the most important ones. Note that the modulus in C++ is represented by the operator % whereas in Fortran we employ the intrinsic function MOD. Note also that the increment operator ++ and the decrement operator $--$ is not available in Fortran . In C++ these operators have the following meaning

++x;	or	x++;	has the same meaning as	x = x + 1;
$--$x;	or	x$--$;	has the same meaning as	x = x $-$ 1;

Table 2.5 lists several relational and arithmetic operators. Logical operators in C++ and Fortran are listed in 2.6. while Table 2.7 shows bitwise operations.

C++ offers also interesting possibilities for combined operators. These are collected in Table 2.8.

Bitwise operations		
C++	Effect	Fortran
~i	Bitwise complement	NOT(j)
i&j	Bitwise and	IAND(i,j)
i^j	Bitwise exclusive or	IEOR(i,j)
i|j	Bitwise inclusive or	IOR(i,j)
i<<j	Bitwise shift left	ISHFT(i,j)
i>>n	Bitwise shift right	ISHFT(i,-j)

Table 2.7: List of bitwise operations.

Expression	meaning	expression	meaning
`a += b;`	`a = a + b;`	`a -= b;`	`a = a - b;`
`a *= b;`	`a = a * b;`	`a /= b;`	`a = a / b;`
`a %= b;`	`a = a % b;`	`a «= b;`	`a = a « b;`
`a »= b;`	`a = a » b;`	`a &= b;`	`a = a & b;`
`a |= b;`	`a = a | b;`	`a ∧= b;`	`a = a ∧ b;`

Table 2.8: C++ specific expressions.

Finally, we show some special operators pertinent to C++ only. The first one is the `?` operator. Its action can be described through the following example

```
A = expression1 ?  expression2 :  expression3;
```

Here `expression1` is computed first. If this is *"true"* ($\neq 0$), then `expression2` is computed and assigned A. If `expression1` is *"false"*, then `expression3` is computed and assigned A.

2.6.2 *Pointers and arrays in C++.*

In addition to constants and variables C++ contain important types such as pointers and arrays (vectors and matrices). These are widely used in most C++ program. C++ allows also for pointer algebra, a feature not included in Fortran . Pointers and arrays are important elements in C++. To shed light on these types, consider the following setup

`int name` — defines an integer variable called `name`. It is given an address in memory where we can store an integer number.

`&name` — is the address of a specific place in memory where the integer `name` is stored. Placing the operator & in front of a variable yields its address in memory.

`int *pointer` — defines and an integer pointer and reserves a location in memory for this specific variable The content of this location is viewed as the address of another place in memory where we have stored an integer.

Note that in C++ it is common to write `int*` **pointer** while in C one usually writes `int` ***pointer**. Here are some examples of legal C++ expressions.

```
name = 0x56;                            /* name gets the hexadecimal value hex 56.   */
pointer = &name;                        /* pointer points to name.                   */
printf("Address of name = %p",pointer); /* writes out the address of name.           */
printf("Value of name= %d",*pointer);   /* writes out the value of name.             */
```

Here's a program which illustrates some of these topics.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/cpp/program7.cpp

```
1   using namespace std;
2   main()
3     {
4       int var;
5       int *pointer;
6
7       pointer = &var;
8       var  = 421;
9       printf("Address of the integer variable var : %p\n",&var);
10      printf("Value of var : %d\n", var);
11      printf("Value of the integer pointer variable: %p\n",pointer);
12      printf("Value which pointer is pointing at :  %d\n",*pointer);
13      printf("Address of the pointer variable : %p\n",&pointer);
14    }
```

Line	Comments
4	• Defines an integer variable var.
5	• Define an integer pointer – reserves space in memory.
7	• The content of the adddress of pointer is the address of var.
8	• The value of var is 421.
9	• Writes the address of var in hexadecimal notation for pointers %p.
10	• Writes the value of var in decimal notation%d.

The ouput of this program, compiled with g++, reads

```
Address of the integer variable var : 0xbfffeb74
Value of var: 421
Value of integer pointer variable : 0xbfffeb74
The value which pointer is pointing at :  421
Address of the pointer variable : 0xbfffeb70
```

In the next example we consider the link between arrays and pointers.

`int matr[2]`	defines a matrix with two integer members – `matr[0]` og `matr[1]`.
`matr`	is a pointer to `matr[0]`.
`(matr + 1)`	is a pointer to `matr[1]`.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter02/cpp/program8.cpp

```
using namespace std;
#included <iostream>
int main()
```

```
4      {
5          int matr[2];
6          int *pointer;
7          pointer = &matr[0];
8          matr[0] = 321;
9          matr[1] = 322;
10         printf("\nAddress of the matrix element matr[1]: %p",&matr[0]);
11         printf("\nValue of the matrix element  matr[1]; %d",matr[0]);
12         printf("\nAddress of the matrix element matr[2]: %p",&matr[1]);
13         printf("\nValue of the matrix element  matr[2]: %d\n", matr[1]);
14         printf("\nValue of the pointer : %p",pointer);
15         printf("\nValue which pointer points at  : %d",*pointer);
16         printf("\nValue which  (pointer+1) points at: %d\n",*(pointer+1));
17         printf("\nAddress of the pointer variable: %p\n",&pointer);
18     }
```

You should especially pay attention to the following

Line	
5	• Declaration of an integer array matr with two elements
6	• Declaration of an integer pointer
7	• The pointer is initialized to point at the first element of the array matr.
8–9	• Values are assigned to the array matr.

The ouput of this example, compiled again with g++, is

```
Address of the matrix element matr[1]: 0xbfffef70
Value of the  matrix element  matr[1]; 321
Address of the matrix element matr[2]: 0xbfffef74
Value of the matrix element  matr[2]: 322
Value of the pointer: 0xbfffef70
The value pointer points at: 321
The value that (pointer+1) points at:  322
Address of the pointer variable : 0xbfffef6c
```

2.6.3 Macros in C++

In C we can define macros, typically global constants or functions through the define statements shown in the simple C-example below for

```
#define ONE     1
#define TWO     ONE + ONE
#define THREE   ONE + TWO

main()
   {
      printf("ONE=%d, TWO=%d, THREE=%d",ONE,TWO,THREE);
   }
```

In C++ the usage of macros is discouraged and you should rather use the declaration for constant variables. You would then replace a statement like **#define** ONE 1 with **const int** ONE = 1;. There is typically much less use of macros in C++ than in C. C++ allows also the definition of our own types based on other existing data types. We can do this using the keyword typedef, whose format is: **typedef** existing _type **new**_type_name ;, where existing_type is a C++ fundamental or compound type and new_type_name is the name for the new type we are defining. For example:

```
typedef char new_name;
typedef unsigned int word ;
typedef char * test;
typedef char field [50];
```

In this case we have defined four data types: new_name, word, test and field as char, unsigned int, char* and char[50] respectively, that we could perfectly use in declarations later as any other valid type

```
new_name mychar, anotherchar, *ptc1;
word myword;
test ptc2;
field name;
```

The use of typedef does not create different types. It only creates synonyms of existing types. That means that the type of myword can be considered to be either word or unsigned int, since both are in fact the same type. Using typedef allows to define an alias for a type that is frequently used within a program. It is also useful to define types when it is possible that we will need to change the type in later versions of our program, or if a type you want to use has a name that is too long or confusing.

In C we could define macros for functions as well, as seen below.

```
1.   #define   MIN(a,b)      ( ((a) < (b)) ?  (a) : (b) )
2.   #define   MAX(a,b)      ( ((a) > (b)) ?  (a) : (b) )
3.   #define   ABS(a)        ( ((a) < 0)   ? -(a) : (a) )
4.   #define   EVEN(a)       ( (a) %2 == 0 ?   1  :  0  )
5.   #define   TOASCII(a)    ( (a)  & 0x7f )
```

In C++ we would replace such function definition by employing so-called **inline** functions. Three of the above functions could then read

```
inline double MIN(double a,double b) (return (((a)<(b)) ? (a):(b));)
inline double MAX(double a,double b)(return (((a)>(b)) ? (a):(b));)
inline double ABS(double a) (return (((a)<0) ? -(a):(a));)
```

where we have defined the transferred variables to be of type **double**. The functions also return a **double** type. These functions could easily be generalized through the use of classes and templates, see chapter 4, to return whather types of real, complex or integer variables.

Inline functions are very useful, especially if the overhead for calling a function implies a significant fraction of the total function call cost. When such function call overhead is significant, a function definition can be preceded by the keyword **inline**. When this function is called, we expect the compiler to generate inline code without function call overhead. However, although inline functions eliminate function call overhead, they can introduce other overheads. When a function is inlined, its code is duplicated for each call. Excessive use of **inline** may thus generate large programs. Large programs can cause excessive paging in virtual memory systems. Too many inline functions can also lengthen compile and link times, on the other hand not inlining small functions like the above that do small computations, can make programs bigger and slower. However, most modern compilers know better than programmer which functions to inline or not. When doing this, you should also test various compiler options. With the compiler option $-O3$ inlining is done automatically by basically all modern compilers.

A good strategy, recommended in many C++ textbooks, is to write a code without inline functions first. As we also suggested in the introductory chapter, you should first write a as simple and clear as possible program, without a strong emphasis on computational speed. Thereafter, when profiling the program one can spot small functions which are called many times. These functions can then be candidates for inlining. If the overall time comsumption is reduced due to inlining specific functions, we can proceed to other sections of the program which could be speeded up.

Another problem with inlined functions is that on some systems debugging an inline function is difficult because the function does not exist at runtime.

2.6.4 Structures in C++ and TYPE in Fortran

A very important part of a program is the way we organize our data and the flow of data when running the code. This is often a neglected aspect especially during the development of an algorithm. A clear understanding of how data are represented makes the program more readable and easier to maintain and extend upon by other users. Till now we have studied elementary variable declarations through keywords like **int** or INTEGER, **double** or REAL(KIND(8) and **char** or its Fortran 90 equivalent CHARACTER. These declarations could also be extended to general multi-dimensional arrays.

However, C++ and Fortran offer other ways as well by which we can organize our data in a more transparent and reusable way. One of these options is through the **struct** declaration of C++, or the correspondingly similar TYPE in Fortran. The latter data type will also be discussed in chapter 4.

The following example illustrates how we could make a general variable which can be reused in defining other variables as well.

Suppose you would like to make a general program which treats quantum mechanical problems from both atomic physics and nuclear physics. In atomic and nuclear physics the single-particle degrees are represented by quantum numbers such orbital angular momentum, total angular momentum, spin and energy. An independent particle model is often assumed as the starting point for building up more complicated many-body correlations in systems with many interacting particles. In atomic physics the effective degrees of freedom are often reduced to electrons interacting with each other, while in nuclear physics the system is described by neutrons and protons. The structure single_particle_descript contains a list over different quantum numbers through various pointers which are initialized by a calling function.

```
struct single_particle_descript{
            int total_orbits;
            int* n;
            int* lorb;
            int* m_l;
            int* jang;
            int* spin;
            double* energy;
            char* orbit_status
     };
```

To describe an atom like Neon we would need three single-particle orbits to describe the ground state wave function if we use a single-particle picture, i.e., the $1s$, $2s$ and $2p$ single-particle orbits. These orbits have a degeneray of $2(2l+1)$, where the first number stems from the possible spin projections and the second from the possible projections of the orbital momentum. In total there are 10 possible single-particle orbits when we account for spin and orbital momentum projections. In this case we would thus need to allocate memory for arrays containing 10 elements.

The above structure is written in a generic way and it can be used to define other variables as well.

For electrons we could write `struct single_particle_descript electrons;` and is a new variable with the name `electrons` containing all the elements of `single_particle_descript`.

The following program segment illustrates how we access these elements To access these elements we could e.g., read from a given device the various quantum numbers:

```
for ( int i = 0; i < electrons.total_orbits; i++){
    cout << `` Read in the quantum numbers for electron i: `` << i <<
        endl;
    cin >> electrons.n[i];
    cin > electrons.lorb[i];
    cin >> electrons.m_l[i];
    cin >> electrons.jang[i];
    cin >> electrons.spin[i];
}
```

The structure `single_particle_descript` can also be used for defining quantum numbers of other particles as well, such as neutrons and protons throughthe new variables `struct single_particle_descript protons` and `struct single_particle_descript neutrons`

The corresponding declaration in Fortran is given by the `TYPE` construct, seen in the following example.

```
TYPE, PUBLIC :: single_particle_descript
    INTEGER :: total_orbits
    INTEGER, DIMENSION(:), POINTER :: n, lorb, jang, spin, m_l
    CHARACTER (LEN=10), DIMENSION(:), POINTER :: orbit_status
    REAL(8), DIMENSION(:), POINTER :: energy
 END TYPE single_particle_descript
```

This structure can again be used to define variables like `electrons`, `protons` and `neutrons` through the statement `TYPE ( single_particle_descript ) :: electrons, protons, neutrons`. More detailed examples on the use of these variable declarations, classes and templates will be given in subsequent chapters and in the appendix A.

2.7 Exercises and projects

Exercise 2.1: Converting from decimal to binary representation

Set up an algorithm which converts a floating number given in the decimal representation to the binary representation. You may or may not use a scientific representation. Write thereafter a program which implements this algorithm.

Exercise 2.2: Summing series

a) Make a program which sums

$$s_{\rm up} = \sum_{n=1}^{N} \frac{1}{n},$$

and

$$s_{\rm down} = \sum_{n=N}^{n=1} \frac{1}{n}.$$

The program should read N from screen and write the final output to screen.

b) Compare s_{up} og s_{down} for different N using both single and double precision for N up to $N = 10^{10}$. Which of the above formula is the most realiable one? Try to give an explanation of possible differences. One possibility for guiding the eye is for example to make a log-log plot of the relative difference as a function of N in steps of 10^n with $n = 1, 2, \ldots, 10$. This means you need to compute $log_{10}(|(s_{\text{up}}(N) - s_{\text{down}}(N))/s_{\text{down}}(N)|)$ as function of $log_{10}(N)$.

Exercise 2.3: Finding alternative expressions

Write a program which computes

$$f(x) = x - \sin x,$$

for a wide range of values of x. Make a careful analysis of this function for values of x near zero. For $x \approx 0$ you may consider to write out the series expansions of $\sin x$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Use the loss of precision theorem of Eq. (2.2) to show that the loss of bits can be limited to at most one bit by restricting x so that

$$1 - \frac{\sin x}{x} \geq \frac{1}{2}.$$

One finds then that x must at least be 1.9, implying that for $|x| < 1.9$ we need to carefully consider the series expansion. For $|x| \geq 1.9$ we can use directly the expression $x - \sin x$.

For $|x| < 1.9$ you should device a recurrence relation for the terms in the series expansion in order to avoid having to compute very large factorials.

Exercise 2.4: Computing e^{-x}

Assume that you do not have access to the intrinsic function for e^x. Write your own algorithm for e^{-x} for all possible values of x, with special care on how to avoid the loss of precision problems discussed in the text. Write thereafter a program which implements this algorithm.

Exercise 2.5: Computing the quadratic equation

The classical quadratic equation $ax^2 + bx + c =$ with solution

$$x = \left(-b \pm \sqrt{b^2 - 4ac}\right)/2a,$$

needs particular attention when $4ac$ is small relative to b^2. Find an algorithm which yields stable results for all possible values of a, b and c. Write thereafter a program and test the results of your computations.

Exercise 2.6: Fortran, C++ and Python functions for machine rounding

Write a Fortran program which reads a real number x and computes the precision in bits (using the function DIGIT(x))for single and double precision, the smallest positive number (using TINY(x)), the largets positive number (using the function HUGE(x)) and the number of leading digits (using the function **PRECISION**(x)). Try thereafter to find similar functionalities in C++ and Python.

Exercise 2.7: Nearest machine number

Write an algorithm and program which reads in a real number x and finds the two nearest machine numbers x_- and x_+, the corresponding relative errors and absolute errors.

Exercise 2.8: Recurrence relations

Recurrence relations are extremely useful in representing functions, and form expedient ways of representing important classes of functions used in the Sciences. We will see two such examples in the discussion below. One example of recurrence relations appears in studies of Fourier series, which enter studies of wave mechanics, be it either in classical systems or quantum mechanical ones. We may need to calculate in an efficient way sums like

$$F(x) = \sum_{n=0}^{N} a_n cos(nx), \tag{2.3}$$

where the coefficients a_n are known numbers and x is the argument of the function $F()$. If we want to solve this problem right on, we could write a simple repetitive loop that multiplies each of the cosines with its respective coefficient a_n like

```
for ( n=0; n < N; n++){
    f +=  an*cos(n*x)
}
```

Even though this seems rather straightforward, it may actually yield a waste of computer time if N is large. The interesting point here is that through the three-term recurrence relation

$$cos(n-1)x - 2cos(x)cos(nx) + cos(n+1)x = 0, \tag{2.4}$$

we can express the entire finite Fourier series in terms of $cos(x)$ and two constants. The essential device is to define a new sequence of coefficients b_n recursively by

$$b_n = (2cos(x))b_{n-1} - b_{n+2} + a_n \qquad n = 0, \dots N-1, N, \tag{2.5}$$

defining $b_{N+1} = b_{N+2} + .. \cdots = 0$ for all $n > N$, the upper limit. We can then determine all the b_n coefficients from a_n and one evaluation of $2cos(x)$. If we replace a_n with b_n in the sum for $F(x)$ in Eq. (2.3) we obtain

$$\begin{aligned} F(x) = \quad & b_N\left[cos(Nx) - 2cos((N-1)x)cos(x) + cos((N-2)x)\right] + \\ & b_{N-1}\left[cos((N-1)x) - 2cos((N-2)x)cos(x) + cos((N-3)x)\right] + \dots \\ & b_2\left[cos(2x) - 2cos^2(x) + 1\right] + b_1\left[cos(x) - 2cos(x)\right] + b_0. \end{aligned} \tag{2.6}$$

Using Eq. (2.4) we obtain the final result

$$F(x) = b_0 - b_1 cos(x), \tag{2.7}$$

and b_0 and b_1 are determined from Eq. (2.3). The latter relation is after Chensaw. This method of evaluating finite series of orthogonal functions that are connected by a linear recurrence is a technique generally available for all standard special functions in mathematical physics, like Legendre polynomials, Bessel functions etc. They all involve two or three terms in the recurrence relations. The general relation can then be written as

$$F_{n+1}(x) = \alpha_n(x)F_n(x) + \beta_n(x)F_{n-1}(x).$$

Evaluate the function $F(x) = \sum_{n=0}^{N} a_n cos(nx)$ in two ways: first by computing the series of Eq. (reffour-1) and then using the equation given in Eq. (2.5). Assume that $a_n = (n+2)/(n+1)$, set e.g., $N = 1000$ and try with different x-values as input.

In project 2.1 we will see another example of recurrence relations used to compute the associated Legendre functions.

Exercise 2.9: Continued fractions

Often, especially when one encounters singular behaviors, one may need to rewrite the function to be evaluated in terms of a taylor expansion. Another possibility is to used so-called continued fractions, which may be viewed as generalizations of a Taylor expansion. When dealing with continued fractions, one possible approach is that of successive substitutions. Let us illustrate this by a simple example, namely the solution of a second order equation

$$x^2 - 4x - 1 = 0, \tag{2.8}$$

which we rewrite as

$$x = \frac{1}{4+x},$$

which in turn could be represented through an iterative substitution process

$$x_{n+1} = \frac{1}{4+x_n},$$

with $x_0 = 0$. This means that we have

$$x_1 = \frac{1}{4},$$

$$x_2 = \frac{1}{4+\frac{1}{4}},$$

$$x_3 = \frac{1}{4+\frac{1}{4+\frac{1}{4}}},$$

and so forth. This is often rewritten in a compact way as

$$x_n = x_0 + \frac{a1}{x_1 + \frac{a_2}{x_2 + \frac{a_3}{x_3 + \frac{a_4}{x_4 + \dots}}}},$$

or as

$$x_n = x_0 + \frac{a1}{x_1+}\frac{a2}{x_2+}\frac{a3}{x_3+}\dots$$

Write a program which implements this continued fraction algorithm and solve iteratively Eq. (2.8). The exact solution is $x = 0.23607$ while already after three iterations you should obtain $x_3 = 0.236111$.

Project 2.1: Special functions, spherical harmonics and associated Legendre polynomials

Many physics problems have spherical harmonics as solutions, such as the angular part of the Schrödinger equation for the hydrogen atom or the angular part of the three-dimensional wave equation or Poisson's equation.

The spherical harmonics for a given orbital momentum L, its projection M for $-L \leq M \leq L$ and angles $\theta \in [0,\pi]$ and $\phi \in [0,2\pi]$ are given by

$$Y_L^M(\theta,\phi) = \sqrt{\frac{(2L+1)(L-M)!}{4\pi(L+M)!}} P_L^M(cos(\theta)) \exp{(iM\phi)},$$

The functions $P_L^M(cos(\theta)$ are the so-called associated Legendre functions. They are normally determined via the usage of recurrence relations. Recurrence relations are unfortunately often unstable, but the following relation is stable (with $x = cos(\theta)$)

$$(L-M)P_L^M(x) = x(2L-1)P_{L-1}^M(x) - (L+M-1)P_{L-2}^M(x),$$

and with the analytic (on closed form) expressions

$$P_M^M(x) = (-1)^M(2M-1)!!(1-x^2)^{M/2},$$

and

$$P_{M+1}^M(x) = x(2M+1)P_M^M(x),$$

we have the starting values and the equations necessary for generating the associated Legendre functions for a general value of L.

a) Make first a function which computes the associated Legendre functions for different values of L and M. Compare with the closed-form results listed in chapter 7.

b) Make thereafter a program which calculates the real part of the spherical harmonics

c) Make plots for various $L = M$ as functions of θ (set $\phi = 0$) and study the behavior as L is increased. Try to explain why the functions become more and more narrow as L increases. In order to make these plots you can use for example gnuplot, as discussed in appendix A.5.

d) Study also the behavior of the spherical harmonics when θ is close to 0 and when it approaches 180 degrees. Try to extract a simple explanation for what you see.

Chapter 3

Numerical differentiation

3.1 Introduction

Numerical integration and differentiation are some of the most frequently needed methods in computational physics. Quite often we are confronted with the need of evaluating either f' or an integral $\int f(x)dx$. The aim of this chapter is to introduce some of these methods with a critical eye on numerical accuracy, following the discussion in the previous chapter.

The next section deals essentially with topics from numerical differentiation. There we present also the most commonly used formulae for computing first and second derivatives, formulae which in turn find their most important applications in the numerical solution of ordinary and partial differential equations. This section serves also the scope of introducing some more advanced C++-programming concepts, such as call by reference and value, reading and writing to a file and the use of dynamic memory allocation.

3.2 Numerical differentiation

The mathematical definition of the derivative of a function $f(x)$ is

$$\frac{df(x)}{dx} = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h}$$

where h is the step size. If we use a Taylor expansion for $f(x)$ we can write

$$f(x+h) = f(x) + hf'(x) + \frac{h^2 f''(x)}{2} + \dots$$

We can then set the computed derivative $f'_c(x)$ as

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} \approx f'(x) + \frac{hf''(x)}{2} + \dots$$

Assume now that we will employ two points to represent the function f by way of a straight line between x and $x+h$. Fig. 3.1 illustrates this subdivision.

This means that we can represent the derivative with

$$f'_2(x) = \frac{f(x+h)-f(x)}{h} + O(h),$$

where the suffix 2 refers to the fact that we are using two points to define the derivative and the dominating error goes like $O(h)$. This is the forward derivative formula. Alternatively, we could use the backward derivative formula

$$f'_2(x) = \frac{f(x) - f(x-h)}{h} + O(h).$$

If the second derivative is close to zero, this simple two point formula can be used to approximate the derivative. If we however have a function like $f(x) = a + bx^2$, we see that the approximated derivative becomes

$$f'_2(x) = 2bx + bh,$$

while the exact answer is $2bx$. Unless h is made very small, and b is not too large, we could approach the exact answer by choosing smaller and smaller and values for h. However, in this case, the subtraction in the numerator, $f(x+h) - f(x)$ can give rise to roundoff errors and eventually a loss of precision.

A better approach in case of a quadratic expression for $f(x)$ is to use a 3-step formula where we evaluate the derivative on both sides of a chosen point x_0 using the above forward and backward two-step formulae and taking the average afterward. We perform again a Taylor expansion but now around $x_0 \pm h$, namely

$$f(x = x_0 \pm h) = f(x_0) \pm hf' + \frac{h^2 f''}{2} \pm \frac{h^3 f'''}{6} + O(h^4), \tag{3.1}$$

which we rewrite as

$$f_{\pm h} = f_0 \pm hf' + \frac{h^2 f''}{2} \pm \frac{h^3 f'''}{6} + O(h^4).$$

Calculating both $f_{\pm h}$ and subtracting we obtain that

$$f'_3 = \frac{f_h - f_{-h}}{2h} - \frac{h^2 f'''}{6} + O(h^3),$$

and we see now that the dominating error goes like h^2 if we truncate at the scond derivative. We call the term $h^2 f'''/6$ the truncation error. It is the error that arises because at some stage in the derivation, a Taylor series has been truncated. As we will see below, truncation errors and roundoff errors play an important role in the numerical determination of derivatives.

For our expression with a quadratic function $f(x) = a + bx^2$ we see that the three-point formula f'_3 for the derivative gives the exact answer $2bx$. Thus, if our function has a quadratic behavior in x in a certain region of space, the three-point formula will result in reliable first derivatives in the interval $[-h, h]$. Using the relation

$$f_h - 2f_0 + f_{-h} = h^2 f'' + O(h^4),$$

we can define the second derivative as

$$f'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} + O(h^2).$$

We could also define five-points formulae by expanding to two steps on each side of x_0. Using a Taylor expansion around x_0 in a region $[-2h, 2h]$ we have

$$f_{\pm 2h} = f_0 \pm 2hf' + 2h^2 f'' \pm \frac{4h^3 f'''}{3} + O(h^4). \tag{3.2}$$

Using Eqs. (3.1) and (3.2), multiplying f_h and f_{-h} by a factor of 8 and subtracting $(8f_h - f_{2h}) - (8f_{-h} - f_{-2h})$ we arrive at a first derivative given by

$$f'_{5c} = \frac{f_{-2h} - 8f_{-h} + 8f_h - f_{2h}}{12h} + O(h^4),$$

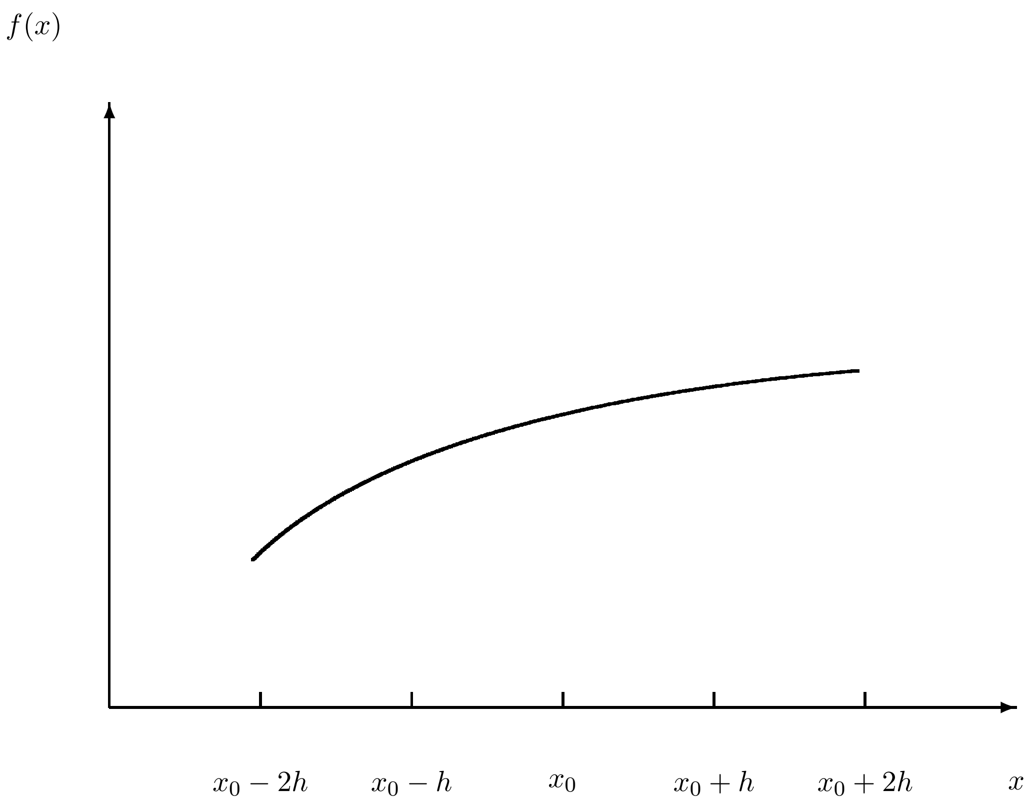


Figure 3.1: Demonstration of the subdivision of the x-axis into small steps h. Each point corresponds to a set of values $x, f(x)$. The value of x is incremented by the step length h. If we use the points x_0 and $x_0 + h$ we can draw a straight line and use the slope at this point to determine an approximation to the first derivative. See text for further discussion.

with a dominating error of the order of h^4 at the price of only two additional function evaluations. This formula can be useful in case our function is represented by a fourth-order polynomial in x in the region $[-2h, 2h]$. Note however that this function includes two additional function evaluations, implying a more time-consuming algorithm. Furthermore, the two additional subtraction can lead to a larger risk of loss of numerical precision when h becomes small. Solving for example a differential equation which involves the first derivative, one needs always to strike a balance between numerical accurary and the time needed to achieve a given result.

It is possible to show that the widely used formulae for the first and second derivatives of a function can be written as

$$\frac{f_h - f_{-h}}{2h} = f_0' + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j}, \tag{3.3}$$

and

$$\frac{f_h - 2f_0 + f_{-h}}{h^2} = f_0'' + 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j}, \tag{3.4}$$

and we note that in both cases the error goes like $O(h^{2j})$. These expressions will also be used when we evaluate integrals.

To show this for the first and second derivatives starting with the three points $f_{-h} = f(x_0 - h)$, $f_0 = f(x_0)$ and $f_h = f(x_0 + h)$, we have that the Taylor expansion around $x = x_0$ gives

$$a_{-h} f_{-h} + a_0 f_0 + a_h f_h = a_{-h} \sum_{j=0}^{\infty} \frac{f_0^{(j)}}{j!} (-h)^j + a_0 f_0 + a_h \sum_{j=0}^{\infty} \frac{f_0^{(j)}}{j!} (h)^j, \tag{3.5}$$

where a_{-h}, a_0 and a_h are unknown constants to be chosen so that $a_{-h} f_{-h} + a_0 f_0 + a_h f_h$ is the best possible approximation for f_0' and f_0''. Eq. (3.5) can be rewritten as

$$a_{-h} f_{-h} + a_0 f_0 + a_h f_h = [a_{-h} + a_0 + a_h] f_0$$
$$+ [a_h - a_{-h}] h f_0' + [a_{-h} + a_h] \frac{h^2 f_0''}{2} + \sum_{j=3}^{\infty} \frac{f_0^{(j)}}{j!} (h)^j \left[(-1)^j a_{-h} + a_h\right].$$

To determine f_0', we require in the last equation that

$$a_{-h} + a_0 + a_h = 0,$$

$$-a_{-h} + a_h = \frac{1}{h},$$

and

$$a_{-h} + a_h = 0.$$

These equations have the solution

$$a_{-h} = -a_h = -\frac{1}{2h},$$

and

$$a_0 = 0,$$

yielding

$$\frac{f_h - f_{-h}}{2h} = f_0' + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j}.$$

To determine f_0'', we require in the last equation that

$$a_{-h} + a_0 + a_h = 0,$$

$$-a_{-h} + a_h = 0,$$

and

$$a_{-h} + a_h = \frac{2}{h^2}.$$

These equations have the solution

$$a_{-h} = -a_h = -\frac{1}{h^2},$$

and

$$a_0 = -\frac{2}{h^2},$$

yielding

$$\frac{f_h - 2f_0 + f_{-h}}{h^2} = f_0'' + 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j}.$$

3.2.1 The second derivative of e^x

As an example, let us calculate the second derivatives of $\exp(x)$ for various values of x. Furthermore, we will use this section to introduce three important C++-programming features, namely reading and writing to a file, call by reference and call by value, and dynamic memory allocation. We are also going to split the tasks performed by the program into subtasks. We define one function which reads in the input data, one which calculates the second derivative and a final function which writes the results to file.

Let us look at a simple case first, the use of `printf` and `scanf`. If we wish to print a variable defined as **double** speed_of_sound; we could for example write `printf (''speed_of_sound = %lf\n'', speed_of_sound);`.

In this case we say that we transfer the value of this specific variable to the function `printf`. The function `printf` *can however not change the value of this variable* (there is no need to do so in this case). Such a call of a specific function is called *call by value*. The crucial aspect to keep in mind is that the value of this specific variable does not change in the called function.

When do we use call by value? And why care at all? We do actually care, because if a called function has the possibility to change the value of a variable when this is not desired, calling another function with this variable may lead to totally wrong results. In the worst cases you may even not be able to spot where the program goes wrong.

We do however use call by value when a called function simply receives the value of the given variable without changing it.

If we however wish to update the value of say an array in a called function, we refer to this call as **call by reference**. What is transferred then is the address of the first element of the array, and the called function has now access to where that specific variable 'lives' and can thereafter change its value.

The function `scanf` is then an example of a function which receives the address of a variable and is allowed to modify it. Afterall, when calling `scanf` we are expecting a new value for a variable. A typical call could be `scanf(''%lf\n'', &speed_of_sound);`.

Consider now the following program

```
//
// This program module
//  demonstrates memory allocation and data transfer in
```

```
   // between functions in  C++
   //

#include <stdio.h>                                    //  Standard ANSI-C++ include files
#include <stdlib.h>

int main(int argc, char *argv[])
{
   int a:                                                                  // line 1
   int *b;                                                                 // line 2

   a = 10;                                                                 // line 3
   b = new int[10];                                                        // line 4
   for(i = 0; i < 10; i++) {
      b[i] = i;                                                            // line 5
   }
   func( a,b);                                                             // line 6
  return 0;
} // End: function main()

void func( int x, int *y)                                                  // line 7
{
   x += 7;                                                                 // line 8
   *y += 10;                                                               // line 9
   y[6] += 10;                                                             // line 10
   return;                                                                 // line 11
} // End: function func()
```

There are several features to be noted.

- Lines 1,2: Declaration of two variables a and b. The compiler reserves two locations in memory. The size of the location depends on the type of variable. Two properties are important for these locations – the address in memory and the content in the

- Line 3: The value of a is now 10.

- Line 4: Memory to store 10 integers is reserved. The address to the first location is stored in b. The address of element number 6 is given by the expression (b + 6).

- Line 5: All 10 elements of b are given values: b[0] = 0, b[1] = 1, ..., b[9] = 9;

- Line 6: The main() function calls the function func() and the program counter transfers to the first statement in func(). With respect to data the following happens. The content of a (= 10) and the content of b (a memory address) are copied to a stack (new memory location) associated with the function func()

- Line 7: The variable x and y are local variables in func(). They have the values – x = 10, y = address of the first element in b in the main() program.

- Line 8: The local variable x stored in the stack memory is changed to 17. Nothing happens with the value a in main().

– Line 9: The value of y is an address and the symbol *y stands for the position in memory which has this address. The value in this location is now increased by 10. This means that the value of b[0] in the main program is equal to 10. Thus func() has modified a value in main().

– Line 10: This statement has the same effect as line 9 except that it modifies element b[6] in main() by adding a value of 10 to what was there originally, namely 6.

– Line 11: The program counter returns to main(), the next expression after *func(a,b);*. All data on the stack associated with func() are destroyed.

– The value of a is transferred to func() and stored in a new memory location called x. Any modification of x in func() does not affect in any way the value of a in main(). This is called **transfer of data by value**. On the other hand the next argument in func() is an address which is transferred to func(). This address can be used to modify the corresponding value in main(). In the programming language C it is expressed as a modification of the value which y points to, namely the first element of b. This is called **transfer of data by reference** and is a method to transfer data back to the calling function, in this case main().

C++ allows however the programmer to use solely call by reference (note that call by reference is implemented as pointers). To see the difference between C and C++, consider the following simple examples. In C we would write

```
int n; n =8;
func(&n); /* &n is a pointer to n */
....
void func(int *i)
{
  *i = 10; /* n is changed to 10 */
  ....
}
```

whereas in C++ we would write

```
int n; n =8;
func(n); // just transfer n itself
....
void func(int& i)
{
  i = 10; // n is changed to 10
  ....
}
```

Note well that the way wex have defined the input to the function `func(int& i)` or `func(int *i)` decides how we transfer variables to a specific function. The reason why we emphasize the difference between call by value and call by reference is that it allows the programmer to avoid pitfalls like unwanted changes of variables. However, many people feel that this reduces the readability of the code. It is more or less common in C++ to use call by reference, since it gives a much cleaner code. Recall also that behind the curtain references are usually implemented as pointers. When we transfer large objects such a matrices and vectors one should always use call by reference. Copying such objects to a called function slows down considerably the execution. If you need to keep the value of a call by reference object, you should use the **const** declaration.

In programming languages like Fortran one uses only call by reference, but you can flag whether a called function or subroutine is allowed or not to change the value by declaring for example an integer value as INTEGER, INTENT(IN):: i. The local function cannot change the value of i. Declaring a transferred values as INTEGER, INTENT(OUT):: i. allows the local function to change the variable i.

Initialisations and main program

In every program we have to define the functions employed. The style chosen here is to declare these functions at the beginning, followed thereafter by the main program and the detailed task performed by each function. Another possibility is to include these functions and their statements before the main program, meaning that the main program appears at the very end. I find this programming style less readable however since I prefer to read a code from top to bottom. A further option, specially in connection with larger projects, is to include these function definitions in a user defined header file. The following program shows also (although it is rather unnecessary in this case due to few tasks) how one can split different tasks into specialized functions. Such a division is very useful for larger projects and programs.

In the first version of this program we use a more C-like style for writing and reading to file. At the end of this section we include also the corresponding C++ and Fortran files.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter03/cpp/program1.cpp

```
/*
**     Program to compute the second derivative of exp(x).
**     Three calling functions are included
**     in this version. In one function we read in the data from screen,
**     the next function computes the second derivative
**     while the last function prints out data to screen.
*/
using namespace std;
# include  <iostream>

void initialise (double *, double *, int *);
void second_derivative( int, double, double, double *, double *);
void output( double *, double *, double, int);

int main()
{
        // declarations of variables
        int number_of_steps;
        double x, initial_step;
        double *h_step, *computed_derivative;
        //  read in input data from screen
        initialise (&initial_step, &x, &number_of_steps);
        //  allocate space in memory for the one-dimensional arrays
        //  h_step and computed_derivative
        h_step =  new double[number_of_steps];
        computed_derivative = new double[number_of_steps];
        //  compute the second derivative of exp(x)
        second_derivative( number_of_steps, x, initial_step, h_step,
                           computed_derivative);
        //  Then we print the results to file
        output(h_step, computed_derivative, x, number_of_steps );
        // free memory
```

```
        delete [] h_step;
        delete [] computed_derivative;
        return 0;
}   // end main program
```

We have defined three additional functions, one which reads in from screen the value of x, the initial step length h and the number of divisions by 2 of h. This function is called initialise . To calculate the second derivatives we define the function second_derivative . Finally, we have a function which writes our results together with a comparison with the exact value to a given file. The results are stored in two arrays, one which contains the given step length h and another one which contains the computed derivative.

These arrays are defined as pointers through the statement

```
double *h_step, *computed_derivative;
```

A call in the main function to the function second_derivative looks then like this

```
second_derivative( number_of_steps, x, intial_step, h_step,
    computed_derivative);
```

while the called function is declared in the following way

```
void second_derivative(int number_of_steps, double x, double *h_step,double
    *computed_derivative);
```

indicating that **double** *h_step, **double** *computed_derivative; are pointers and that we transfer the address of the first elements. The other variables **int** number_of_steps, **double** x; are transferred by value and are not changed in the called function.

Another aspect to observe is the possibility of dynamical allocation of memory through the **new** function. In the included program we reserve space in memory for these three arrays in the following way h_step = **new double**[number_of_steps]; and computed_derivative = **new double**[number_of_steps]; When we no longer need the space occupied by these arrays, we free memory through the declarations **delete** [] h_step; and **delete** [] computed_derivative;

The function initialise

```
//     Read in from screen the initial step, the number of steps
//     and the value of x

void initialise (double *initial_step,  double *x, int *number_of_steps)
{
   printf("Read in from screen initial step, x and number of steps\n");
   scanf("%lf %lf %d",initial_step, x, number_of_steps);
   return;
}  // end of function initialise
```

This function receives the addresses of the three variables **double** * initial_step , **double** *x, **int** * number_of_steps; and returns updated values by reading from screen.

The function second_derivative

```
//  This function computes the second derivative

void second_derivative( int number_of_steps, double x,
                        double initial_step, double *h_step,
                        double *computed_derivative)
{
        int counter;
        double h;
        //     calculate the step size
        //     initialise the derivative, y and x (in minutes)
        //     and iteration counter
        h = initial_step;
        //  start computing for different step sizes
        for (counter=0; counter < number_of_steps; counter++ )
        {
           //  setup arrays with derivatives and step sizes
           h_step[counter] = h;
           computed_derivative[counter] =
                       (exp(x+h)-2.*exp(x)+exp(x-h))/(h*h);
           h = h*0.5;
         } // end of do loop
         return;
}   // end of function second derivative
```

The loop over the number of steps serves to compute the second derivative for different values of h. In this function the step is halved for every iteration (you could obviously change this to larger or smaller step variations). The step values and the derivatives are stored in the arrays h_step and **double** computed_derivative.

The output function

This function computes the relative error and writes to a chosen file the results.

The last function here illustrates how to open a file, write and read possible data and then close it. In this case we have fixed the name of file. Another possibility is obviously to read the name of this file together with other input parameters. The way the program is presented here is slightly unpractical since we need to recompile the program if we wish to change the name of the output file.

An alternative is represented by the following program C program. This program reads from screen the names of the input and output files.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter03/cpp/program2.cpp

```
#include <stdio.h>
#include <stdlib.h>
int col:

int main(int argc, char *argv[])
{
    FILE *in, *out;
    int c;
    if( argc < 3)  {
    printf("You have to read in :\n");
    printf("in_file and out_file \n");
```

```
12      exit(1);
13      in = fopen( argv[1], "r");}     // returns pointer to the in_file
14      if( inn == NULL )  {          // can't find in_file
15         printf("Can't find the input file %s\n", argv[1]);
16         exit(1);
17      }
18      out = fopen( argv[2], "w");      // returns a pointer to the out_file
19      if( ut == NULL )  {          // can't find out_file
20         printf("Can't find the output file %s\n", argv[2]);
21         exit(1);
22      }
    ... program statements

23      fclose(in);
24      fclose(out);
25      return 0;
}
```

This program has several interesting features.

Line	Program comments
5	• `main()` takes three arguments, given by argc. argv points to the following: the name of the program, the first and second arguments, in this case file names to be read from screen.
7	• C++ has a `data type` called `FILE`. The pointers `in` and `out` point to specific files. They must be of the type `FILE`.
10	• The command line has to contain 2 filenames as parameters.
13–17	• The input file has to exit, else the pointer returns NULL. It has only read permission.
18–22	• Same for the output file, but now with write permission only.
23–24	• Both files are closed before the main program ends.

The above represents a standard procedure in C for reading file names. C++ has its own class for such operations.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter03/cpp/program3.cpp

```
/*
**     Program to compute the second derivative of exp(x).
**     In this version we use C++ options for reading and
**     writing files and data. The rest of the code is as in
**     programs/chapter3/program1.cpp
**     Three calling functions are included
**     in this version. In one function we read in the data from screen,
**     the next function computes the second derivative
**     while the last function prints out data to screen.
*/
using namespace std;
# include  <iostream>
# include <fstream>
# include <iomanip>
# include <cmath>
void initialise (double *, double *, int *);
```

```
void second_derivative( int, double, double, double *, double *);
void output( double *, double *, double, int);

ofstream ofile;

int main(int argc, char* argv[])
{
    // declarations of variables
    char *outfilename;
    int number_of_steps;
    double x, initial_step;
    double *h_step, *computed_derivative;
    // Read in output file, abort if there are too few command-line
       arguments
    if( argc <= 1 ){
      cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
      exit(1);
    }
    else{
      outfilename=argv[1];
    }
    ofile.open(outfilename);
    //  read in input data from screen
    initialise (&initial_step, &x, &number_of_steps);
    //  allocate space in memory for the one-dimensional arrays
    //  h_step and computed_derivative
    h_step = new double[number_of_steps];
    computed_derivative = new double[number_of_steps];
    //  compute the second derivative of exp(x)
    second_derivative( number_of_steps, x, initial_step, h_step,
                       computed_derivative);
    //  Then we print the results to file
   output(h_step, computed_derivative, x, number_of_steps );
    // free memory
    delete [] h_step;
    delete [] computed_derivative;
    // close output file
    ofile.close();
    return 0;
}   // end main program
```

The main part of the code includes now an object declaration `ofstream ofile` which is included in C++ and allows the programmer to open and declare files. This is done via the statement `ofile.open(outfilename);`. We close the file at the end of the main program by writing `ofile.close();`. There is a corresponding object for reading inputfiles. In this case we declare prior to the main function, or in an evantual header file, `ifstream ifile` and use the corresponding statements `ifile.open(infilename);` and `ifile.close();` for opening and closing an input file. Note that we have declared two character variables `char* outfilename;` and `char* infilename;`. In order to use these options we need to include a corresponding library of functions using `#include <fstream>`.

One of the problems with C++ is that formatted output is not as easy to use as the printf and scanf functions in C. The output function using the C++ style is included below.

```
//     function to write out the final results
void output(double *h_step, double *computed_derivative, double x,
            int number_of_steps )
{
      int i;
      ofile << "RESULTS:" << endl;
      ofile << setiosflags(ios::showpoint | ios::uppercase);
      for( i=0; i < number_of_steps; i++)
        {
        ofile << setw(15) << setprecision(8) << log10(h_step[i]);
        ofile << setw(15) << setprecision(8) <<
        log10(fabs(computed_derivative[i]-exp(x))/exp(x)) << endl;
         }
}  // end of function output
```

The function setw(15) reserves an output of 15 spaces for a given variable while setprecision (8) yields eight leading digits. To use these options you have to use the declaration **# include** <iomanip>

Before we discuss the results of our calculations we list here the corresponding Fortran program. The corresponding Fortran example is

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter03/f90/program1.f90

```
!      Program to compute the second derivative of exp(x).
!      Only one calling function is included.
!      It computes the second derivative and is included in the
!      MODULE functions as a separate method
!      The variable h is the step size. We also fix the total number
!      of divisions by 2 of h. The total number of steps is read from
!      screen
MODULE constants
  ! definition of variables for double precisions and complex variables
  INTEGER, PARAMETER :: dp = KIND(1.0D0)
  INTEGER, PARAMETER :: dpc = KIND((1.0D0,1.0D0))
END MODULE constants

! Here you can include specific functions which can be used by
! many subroutines or functions

MODULE functions
USE constants
IMPLICIT NONE
CONTAINS
  SUBROUTINE derivative(number_of_steps, x, initial_step, h_step, &
         computed_derivative)
    USE constants
    INTEGER, INTENT(IN) :: number_of_steps
    INTEGER :: loop
    REAL(DP), DIMENSION(number_of_steps), INTENT(INOUT) :: &
         computed_derivative, h_step
    REAL(DP), INTENT(IN) :: initial_step, x
    REAL(DP) :: h
    !      calculate the step size
    !      initialise the derivative, y and x (in minutes)
```

```
      !      and iteration counter
      h = initial_step
      ! start computing for different step sizes
      DO loop=1,  number_of_steps
         !  setup arrays with derivatives and step sizes
         h_step(loop) = h
         computed_derivative(loop) = (EXP(x+h)-2.*EXP(x)+EXP(x-h))/(h*h)
         h = h*0.5
      ENDDO
   END SUBROUTINE derivative

END MODULE functions

PROGRAM second_derivative
   USE constants
   USE functions
   IMPLICIT NONE
   ! declarations of variables
   INTEGER :: number_of_steps, loop
   REAL(DP) :: x, initial_step
   REAL(DP), ALLOCATABLE, DIMENSION(:) :: h_step, computed_derivative
   !  read in input data from screen
   WRITE(*,*) 'Read in initial step, x value and number of steps'
   READ(*,*) initial_step, x, number_of_steps
   ! open file to write results on
   OPEN(UNIT=7,FILE='out.dat')
   !  allocate space in memory for the one-dimensional arrays
   !  h_step and computed_derivative
   ALLOCATE(h_step(number_of_steps),computed_derivative(number_of_steps))
   ! compute the second derivative of exp(x)
   ! initialize the arrays
   h_step = 0.0_dp; computed_derivative = 0.0_dp
   CALL  derivative(number_of_steps,x,initial_step,h_step,computed_derivative
       )

   !  Then we print the results to file
   DO loop=1,  number_of_steps
      WRITE(7,'(E16.10,2X,E16.10)') LOG10(h_step(loop)),&
      LOG10 ( ABS ( (computed_derivative(loop)-EXP(x))/EXP(x)))
   ENDDO
   ! free memory
   DEALLOCATE( h_step, computed_derivative)
   ! close the output file
   CLOSE(7)

END PROGRAM second_derivative
```

The **MODULE** declaration in Fortran allows one to place functions like the one which calculates second derivatives in a module. Since this is a general method, one could extend its functionality by simply transfering the name of the function to differentiate. In our case we use explicitely the exponential function, but there is nothing which hinders us from defining other functions. Note the usage of the module **constants** where we define double and complex variables. If one wishes to switch to another precision, one just needs to change the declaration in one part of the program only. This hinders possible errors which arise if one has to change variable declarations in every function and subroutine. Finally, dynamic memory allocation and deallocation is in Fortran done with the keywords **ALLOCATE**(array(**size**)) and **DEALLOCATE**(array). Although most compilers deallocate and thereby free space in memory when leaving a function, you should always deallocate an array when it is no longer needed. In case your arrays are very large, this may block unnecessarily large fractions of the memory. Furthermore, you should always initialise arrays. In the example above, we note that Fortran allows us to simply write **h_step = 0.0_dp; computed_derivative = 0.0_dp**, which means that all elements of these two arrays are set to zero. Coding arrays in this manner brings us much closer to the way we deal with mathematics. In Fortran it is irrelevant whether this is a one-dimensional or multi-dimensional array. In the next next chapter, where we deal with allocation of matrices, we will introduce the numerical library Blitz++ which allows for similar treatments of arrays in C++. By default however, these features are not included in the ANSI C++ standard.

Results

In Table 3.1 we present the results of a *numerical evaluation* for various step sizes for the second derivative of $\exp(x)$ using the approximation $f''_0 = \frac{f_h - 2f_0 + f_{-h}}{h^2}$. The results are compared with the exact ones for various x values. Note well that as the step is decreased we get closer to the exact value. However, if

x	$h = 0.1$	$h = 0.01$	$h = 0.001$	$h = 0.0001$	$h = 0.0000001$	Exact
0.0	1.000834	1.000008	1.000000	1.000000	1.010303	1.000000
1.0	2.720548	2.718304	2.718282	2.718282	2.753353	2.718282
2.0	7.395216	7.389118	7.389057	7.389056	7.283063	7.389056
3.0	20.102280	20.085704	20.085539	20.085537	20.250467	20.085537
4.0	54.643664	54.598605	54.598155	54.598151	54.711789	54.598150
5.0	148.536878	148.414396	148.413172	148.413161	150.635056	148.413159

Table 3.1: Result for numerically calculated second derivatives of $\exp(x)$ as functions of the chosen step size h. A comparison is made with the exact value.

it is further decreased, we run into problems of loss of precision. This is clearly seen for $h = 0.0000001$. This means that even though we could let the computer run with smaller and smaller values of the step, there is a limit for how small the step can be made before we loose precision.

3.2.2 Error analysis

Let us analyze these results in order to see whether we can find a minimal step length which does not lead to loss of precision. Furthermore In Fig. 3.2 we have plotted

$$\epsilon = log_{10}\left(\left|\frac{f''_{\text{computed}} - f''_{\text{exact}}}{f''_{\text{exact}}}\right|\right),$$

as function of $log_{10}(h)$. We used an intial step length of $h = 0.01$ and fixed $x = 10$. For large values of h, that is $-4 < log_{10}(h) < -2$ we see a straight line with a slope close to 2. Close to $log_{10}(h) \approx -4$ the relative error starts increasing and our computed derivative with a step size $log_{10}(h) < -4$, may no longer be reliable.

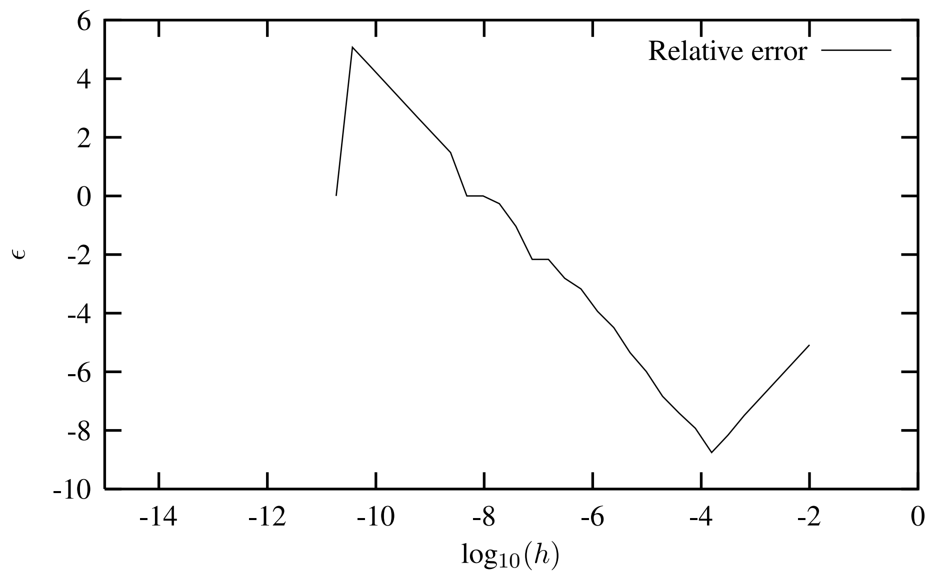


Figure 3.2: Log-log plot of the relative error of the second derivative of e^x as function of decreasing step lengths h. The second derivative was computed for $x = 10$ in the program discussed above. See text for further details

Can we understand this behavior in terms of the discussion from the previous chapter? In chapter 2 we assumed that the total error could be approximated with one term arising from the loss of numerical precision and another due to the truncation or approximation made, that is

$$\epsilon_{\mathrm{tot}} = \epsilon_{\mathrm{approx}} + \epsilon_{\mathrm{ro}}.$$

For the computed second derivative, Eq. (3.4), we have

$$f_0'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} - 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j},$$

and the truncation or approximation error goes like

$$\epsilon_{\mathrm{approx}} \approx \frac{f_0^{(4)}}{12} h^2.$$

If we were not to worry about loss of precision, we could in principle make h as small as possible. However, due to the computed expression in the above program example

$$f_0'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} = \frac{(f_h - f_0) + (f_{-h} - f_0)}{h^2},$$

we reach fairly quickly a limit for where loss of precision due to the subtraction of two nearly equal numbers becomes crucial. If $(f_{\pm h} - f_0)$ are very close, we have $(f_{\pm h} - f_0) \approx \epsilon_M$, where $|\epsilon_M| \leq 10^{-7}$ for single and $|\epsilon_M| \leq 10^{-15}$ for double precision, respectively.

We have then

$$|f_0''| = \left| \frac{(f_h - f_0) + (f_{-h} - f_0)}{h^2} \right| \leq \frac{2\epsilon_M}{h^2}.$$

Our total error becomes

$$|\epsilon_{\mathrm{tot}}| \leq \frac{2\epsilon_M}{h^2} + \frac{f_0^{(4)}}{12}h^2. \tag{3.6}$$

It is then natural to ask which value of h yields the smallest total error. Taking the derivative of $|\epsilon_{\mathrm{tot}}|$ with respect to h results in

$$h = \left(\frac{24\epsilon_M}{f_0^{(4)}}\right)^{1/4}.$$

With double precision and $x = 10$ we obtain

$$h \approx 10^{-4}.$$

Beyond this value, it is essentially the loss of numerical precision which takes over. We note also that the above qualitative argument agrees seemingly well with the results plotted in Fig. 3.2 and Table 3.1. The turning point for the relative error at approximately $h \approx \times 10^{-4}$ reflects most likely the point where roundoff errors take over. If we had used single precision, we would get $h \approx 10^{-2}$. Due to the subtractive cancellation in the expression for f'' there is a pronounced detoriation in accuracy as h is made smaller and smaller.

It is instructive in this analysis to rewrite the numerator of the computed derivative as

$$(f_h - f_0) + (f_{-h} - f_0) = (e^{x+h} - e^x) + (e^{x-h} - e^x),$$

as

$$(f_h - f_0) + (f_{-h} - f_0) = e^x(e^h + e^{-h} - 2),$$

since it is the difference $(e^h + e^{-h} - 2)$ which causes the loss of precision. The results, still for $x = 10$ are shown in the Table 3.2. We note from this table that at $h \approx \times 10^{-8}$ we have essentially lost all leading

h	$e^h + e^{-h}$	$e^h + e^{-h} - 2$
10^{-1}	2.0100083361116070	$1.0008336111607230\times10^{-2}$
10^{-2}	2.0001000008333358	$1.0000083333605581\times10^{-4}$
10^{-3}	2.0000010000000836	$1.0000000834065048\times10^{-6}$
10^{-4}	2.0000000099999999	$1.0000000050247593\times10^{-8}$
10^{-5}	2.0000000001000000	$9.9999897251734637\times10^{-11}$
10^{-6}	2.0000000000010001	$9.9997787827987850\times10^{-13}$
10^{-7}	2.0000000000000098	$9.9920072216264089\times10^{-15}$
10^{-8}	2.0000000000000000	$0.0000000000000000\times10^{0}$
10^{-9}	2.0000000000000000	$1.1102230246251565\times10^{-16}$
10^{-10}	2.0000000000000000	$0.0000000000000000\times10^{0}$

Table 3.2: Result for the numerically calculated numerator of the second derivative as function of the step size h. The calculations have been made with double precision.

digits.

From Fig. 3.2 we can read off the slope of the curve and thereby determine empirically how truncation errors and roundoff errors propagate. We saw that for $-4 < log_{10}(h) < -2$, we could extract a slope close to 2, in agreement with the mathematical expression for the truncation error.

We can repeat this for $-10 < log_{10}(h) < -4$ and extract a slope ≈ -2. This agrees again with our simple expression in Eq. (3.6).

3.3 Exercises and projects

Exercise 3.1: Computing derivatives numerically

We want you to compute the first derivative of

$$f(x) = tan^{-1}(x)$$

for $x = \sqrt{2}$ with step lengths h. The exact answer is $1/3$. We want you to code the derivative using the following two formulae

$$f'_{2c}(x) = \frac{f(x+h) - f(x)}{h} + O(h), \tag{3.7}$$

and

$$f'_{3c} = \frac{f_h - f_{-h}}{2h} + O(h^2), \tag{3.8}$$

with $f_{\pm h} = f(x \pm h)$.

(a) Find mathematical expressions for the total error due to loss of precision and due to the numerical approximation made. Find the step length which gives the smallest value. Perform the analysis with both double and single precision.

(b) Make thereafter a program which computes the first derivative using Eqs. (3.7) and (3.8) as function of various step lengths h and let $h \to 0$. Compare with the exact answer.

Your program should contain the following elements:

- A vector (array) which contains the step lengths. Use dynamic memory allocation.
- Vectors for the computed derivatives of Eqs. (3.7) and (3.8) for both single and double precision.
- A function which computes the derivative and contains call by value and reference (for C++ users only).
- Add a function which writes the results to file.

(c) Compute thereafter

$$\epsilon = log_{10}\left(\left|\frac{f'_{\text{computed}} - f'_{\text{exact}}}{f'_{\text{exact}}}\right|\right),$$

as function of $log_{10}(h)$ for Eqs. (3.7) and (3.8) for both single and double precision. Plot the results and see if you can determine empirically the behavior of the total error as function of h.

Chapter 4

Linear algebra

> In the training of programming for scientific computation the emphasis has historically been on squeezing out every drop of floating point performance for a given algorithm. ...... This practice, however, leads to highly tuned racecarlike software codes: delicate, easily broken and difficult to maintain, but capable of outperforming more user-friendly family cars. *Smith, Bjorstad and Gropp, An introduction to MPI [16]*

4.1 Introduction

In this chapter we deal with basic matrix operations, such as the solution of linear equations, calculate the inverse of a matrix, its determinant etc. The solution of linear equations is an important part of numerical mathematics and arises in many applications in the sciences. Here we focus in particular on so-called direct or elimination methods, which are in principle determined through a finite number of arithmetic operations. Iterative methods will be discussed in connection with eigenvalue problems in chapter 12.

This chapter serves also the purpose of introducing important programming details such as handling memory allocation for matrices and the usage of the libraries which follow these lectures. Classes and pertinent programming aspects are relegated to the appendix.

The algorithms we describe and their original source codes are taken from the widely used software package LAPACK [25], which follows two other popular packages developed in the 1970s, namely EISPACK and LINPACK. The latter was developed for linear equations and least square problems while the former was developed for solving symmetric, unsymmetric and generalized eigenvalue problems. From LAPACK's website `http://www.netlib.org` it is possible to download for free all source codes from this library. Both C++ and Fortran versions are available. Another important library is BLAS [26], which stands for Basic Linear Algebra Subprogram. It contains efficient codes for algebraic operations on vectors, matrices and vectors and matrices. Basically all modern supercomputer include this library, with efficient algorithms. Else, Matlab offers a very efficient programming environment for dealing with matrices. The classic text from where we have taken most of the formalism exposed here is the book on matrix computations by Golub and Van Loan [27]. Good recent introductory texts are Kincaid and Cheney [24] and Datta [28]. For more advanced ones see Trefethen and Bau III [29], Kress [30] and Demmel [31]. Ref. [27] contains an extensive list of textbooks on eigenvalue problems and linear algebra. LAPACK [25] contains also extensive listings to the research literature on matrix computations. For the introduction of the auxiliary library Blitz++ [32], which allows for a very efficient way of handling arrays in C++ we refer to the online manual at `http://www.oonumerics.org` and the appendix.

4.2 Mathematical intermezzo

The matrices we will deal with are primarily square real symmetric or hermitian ones, assuming thereby that an $n \times n$ matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ for a real matrix[1] and $\mathbf{A} \in \mathbb{C}^{n\times n}$ for a complex matrix. For the sake of simplicity, we take a matrix $\mathbf{A} \in \mathbb{R}^{4\times 4}$ and a corresponding identity matrix $\mathbf{I}$

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \qquad \mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{4.1}$$

where $a_{ij} \in \mathbb{R}$. The inverse of a matrix, if it exists, is defined by

$$\mathbf{A}^{-1} \cdot \mathbf{A} = I.$$

In the following discussion, matrices are always two-dimensional arrays while vectors are one-dimensional arrays. In our nomenclature we will restrict boldfaced capitals letters such as **A** to represent a general matrix, which is a two-dimensional array, while a_{ij} refers to a matrix element with row number i and column number j. Similarly, a vector being a one-dimensional array, is labelled **x** and represented as (for a real vector)

$$\mathbf{x} \in \mathbb{R}^n \iff \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix},$$

with pertinent vector elements $x_i \in \mathbb{R}$. Note that this notation implies $x_i \in \mathbb{R}^{4\times 1}$ and that the members of $\mathbf{x}$ are column vectors. The elements of $x_i \in \mathbb{R}^{1\times 4}$ are row vectors.

Table 4.2 lists some essential features of various types of matrices one may encounter. Some of the

Table 4.1: Matrix properties

Relations	Name	matrix elements
$\mathbf{A} = \mathbf{A}^T$	symmetric	$a_{ij} = a_{ji}$
$\mathbf{A} = \left(\mathbf{A}^T\right)^{-1}$	real orthogonal	$\sum_k a_{ik}a_{jk} = \sum_k a_{ki}a_{kj} = \delta_{ij}$
$\mathbf{A} = \mathbf{A}^*$	real matrix	$a_{ij} = a_{ij}^*$
$\mathbf{A} = \mathbf{A}^\dagger$	hermitian	$a_{ij} = a_{ji}^*$
$\mathbf{A} = \left(\mathbf{A}^\dagger\right)^{-1}$	unitary	$\sum_k a_{ik}a_{jk}^* = \sum_k a_{ki}^*a_{kj} = \delta_{ij}$

matrices we will encounter are listed here

1. Diagonal if $a_{ij} = 0$ for $i \neq j$,

[1] A reminder on mathematical symbols may be appropriate here. The symbol $\mathbb{R}$ is the set of real numbers. Correspondingly, $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{C}$ represent the set of natural, integer and complex numbers, respectively. A symbol like $\mathbb{R}^n$ stands for an n-dimensional real Euclidean space, while $C[a, b]$ is the space of real or complex-valued continuous functions on the interval $[a, b]$, where the latter is a closed interval. Similalry, $C^m[a, b]$ is the space of m-times continuously differentiable functions on the interval $[a, b]$. For more symbols and notations, see the main text.

2. Upper triangular if $a_{ij} = 0$ for $i > j$, which for a 4×4 matrix is of the form

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{nn} \end{pmatrix}$$

3. Lower triangular if $a_{ij} = 0$ for $i < j$

$$\begin{pmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

4. Upper Hessenberg if $a_{ij} = 0$ for $i > j + 1$, which is similar to a upper triangular except that it has non-zero elements for the first subdiagonal row

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

5. Lower Hessenberg if $a_{ij} = 0$ for $i < j + 1$

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

6. Tridiagonal if $a_{ij} = 0$ for $|i - j| > 1$

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

There are many more examples, such as lower banded with bandwidth p for $a_{ij} = 0$ for $i > j + p$, upper banded with bandwidth p for $a_{ij} = 0$ for $i < j + p$, block upper triangular, block lower triangular etc.

For a real $n \times n$ matrix $\mathbf{A}$ the following properties are all equivalent

1. If the inverse of $\mathbf{A}$ exists, $\mathbf{A}$ is nonsingular.

2. The equation $\mathbf{Ax} = 0$ implies $\mathbf{x} = 0$.

3. The rows of $\mathbf{A}$ from a basis of $\mathbb{R}^n$.

4. The columns of $\mathbf{A}$ from a basis of $\mathbb{R}^n$.

5. $\mathbf{A}$ is a product of elementary matrices.

6. 0 is not an eigenvalue of $\mathbf{A}$.

The basic matrix operations that we will deal with are addition and subtraction

$$\mathbf{A} = \mathbf{B} \pm \mathbf{C} \Longrightarrow a_{ij} = b_{ij} \pm c_{ij}, \tag{4.2}$$

scalar-matrix multiplication

$$\mathbf{A} = \gamma \mathbf{B} \Longrightarrow a_{ij} = \gamma b_{ij},$$

vector-matrix multiplication

$$\mathbf{y} = \mathbf{A}\mathbf{x} \Longrightarrow y_i = \sum_{j=1}^{n} a_{ij} x_j, \tag{4.3}$$

matrix-matrix multiplication

$$\mathbf{A} = \mathbf{B}\mathbf{C} \Longrightarrow a_{ij} = \sum_{k=1}^{n} b_{ik} c_{kj}, \tag{4.4}$$

transposition

$$\mathbf{A} = \mathbf{B}^T \Longrightarrow a_{ij} = b_{ji},$$

and if $\mathbf{A} \in \mathbb{C}^{n\times n}$, conjugation results in

$$\mathbf{A} = \overline{\mathbf{B}}^T \Longrightarrow a_{ij} = \overline{b}_{ji},$$

where a variable $\overline{z} = x - \imath y$ denotes the complex conjugate of $z = x + \imath y$. In a similar way we have the following basic vector operations, namely addition and subtraction

$$\mathbf{x} = \mathbf{y} \pm \mathbf{z} \Longrightarrow x_i = y_i \pm z_i,$$

scalar-vector multiplication

$$\mathbf{x} = \gamma \mathbf{y} \Longrightarrow x_i = \gamma y_i,$$

vector-vector multiplication (called Hadamard multiplication)

$$\mathbf{x} = \mathbf{y}\mathbf{z} \Longrightarrow x_i = y_i z_i,$$

the inner or so-called dot product

$$c = \mathbf{y}^T \mathbf{z} \Longrightarrow c = \sum_{j=1}^{n} y_j z_j, \tag{4.5}$$

with a c a constant and the outer product, which yields a matrix,

$$\mathbf{A} = \mathbf{y}\mathbf{z}^T \Longrightarrow a_{ij} = y_i z_j, \tag{4.6}$$

Other important operations are vector and matrix norms. A class of vector norms are the so-called p-norms

$$||\mathbf{x}||_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}},$$

where $p \geq 1$. The most important are the 1, 2 and ∞ norms given by

$$||\mathbf{x}||_1 = |x_1| + |x_2| + \cdots + |x_n|,$$

$$||\mathbf{x}||_2 = (|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2)^{\frac{1}{2}} = (\mathbf{x}^T \mathbf{x})^{\frac{1}{2}},$$

and

$$||\mathbf{x}||_\infty = \max |x_i|,$$

for $1 \le i \le n$. From these definitions, one can derive several important relations, of which the so-called Cauchy-Schwartz inequality is of great importance for many algorithms. It reads for any $\mathbf{x}$ and $\mathbf{y}$ in a real or complex inner product space satisfy

$$|\mathbf{x}^T\mathbf{y}| \le ||\mathbf{x}||_2||\mathbf{y}||_2,$$

and the equality is obeyed only if $\mathbf{x}$ and $\mathbf{y}$ are linearly dependent. An important relation which follows from the Cauchy-Schwartz relation is the famous triangle relation, which states that for any $\mathbf{x}$ and $\mathbf{y}$ in a real or complex inner product space satisfy

$$||\mathbf{x}+\mathbf{y}||_2 \le ||\mathbf{x}||_2 + ||\mathbf{y}||_2.$$

Proofs can be found in for example Ref. [27]. As discussed in chapter 2, the analysis of the relative error is important in our studies of loss of numerical precision. Using a vector norm we can define the relative error for the machine representation of a vector $\mathbf{x}$. We assume that $fl(\mathbf{x}) \in \mathbb{R}^n$ is the machine representation of a vector $\mathbf{x} \in \mathbb{R}^n$. If $\mathbf{x} \ne 0$, we define the relative error as

$$\epsilon = \frac{||fl(\mathbf{x}) - \mathbf{x}||}{||\mathbf{x}||}.$$

Using the ∞-norm one can define a relative error that can be translated into a statement on the correct significant digits of $fl(\mathbf{x})$,

$$\frac{||fl(\mathbf{x}) - \mathbf{x}||_\infty}{||\mathbf{x}||_\infty} \approx 10^{-l},$$

where the largest component of $fl(\mathbf{x})$ has roughly l correct significant digits.

We can define similar matrix norms as well. The most frequently used are the Frobenius norm

$$||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2},$$

and the p-norms

$$||\mathbf{A}||_p = \frac{||\mathbf{A}\mathbf{x}||_p}{||\mathbf{x}||_p},$$

assuming that $\mathbf{x} \ne 0$. We refer the reader to the text of Golub and Van Loan [27] for a further discussion of these norms.

The way we implement these operations will be discussed below, as it depends on the programming language we opt for.

4.3 Programming details

Many programming problems arise from improper treatment of arrays. In this section we will discuss some important points such as array declaration, memory allocation and array transfer between functions. We distinguish between two cases: (a) array declarations where the array size is given at compilation time, and (b) where the array size is determined during the execution of the program, so-called dymanic memory allocation. Useful references on C++ programming details, in particular on the use of pointers and memory allocation, are Reek's text [33] on pointers in C, Berryhill's monograph [34] on scientific programming in C++ and finally Franek's text [35] on memory as a programming concept in C and C++. Good allround texts on C++ programming in engineering and science are the books by Flowers [19] and Barton and Nackman [20]. See also the online lecture notes on C++ at `http://heim.ifi.uio.no/~hpl/INF-VERK4830`. For Fortran we recommend the online lectures at `http://folk.uio.no/gunnarw/INF-VERK4820`. These web pages contain extensive references to other C++ and Fortran resources. Both web pages contain enough material, lecture notes and exercises, in order to serve as material for own studies.

Figure 4.1: Segmentation fault, again and again! Alas, this is a situation you must likely will end up in, unless you initialize, access, allocate or deallocate properly your arrays. Many program development environments such as Dev C++ at `www.bloodshed.net` provide debugging possibilities. Another possibility, discussed in appendix A is to use the debugger GDB within the text editor emacs. Beware however that there may be segmentation errors which occur due to errors in libraries of the operating system. (Drawing: courtesy by Victoria Popsueva 2003.)

4.3.1 Declaration of fixed-sized vectors and matrices

Table 4.2 presents a small program which treats essential features of vector and matrix handling where the dimensions are declared in the program code.

In **line a** we have a standard C++ declaration of a vector. The compiler reserves memory to store five integers. The elements are vec[0], vec[1],....,vec[4]. Note that the numbering of elements starts with zero. Declarations of other data types are similar, including structure data.

The symbol vec is an element in memory containing the address to the first element vec[0] and is a pointer to a vector of five integer elements.

In **line b** we have a standard fixed-size C++ declaration of a matrix. Again the elements start with zero, matr[0][0], matr[0][1], ..., matr[0][4], matr[1][0],.... . This sequence of elements also shows how data are stored in memory. For example, the element matr[1][0] follows matr[0][4]. This is important in order to produce an efficient code and avoid memory stride.

There is one further important point concerning matrix declaration. In a similar way as for the symbol **vec**, **matr** is an element in memory which contains an address to a vector of three elements, but now these elements are not integers. Each element is a vector of five integers. This is the correct way to understand the declaration in **line b**. With respect to pointers this means that matr is *pointer-to-a-pointer-to-an-integer* which we can write $**$matr. Furthermore $*$matr is *a-pointer-to-a-pointer* of five integers. This interpretation is important when we want to transfer vectors and matrices to a function.

In **line c** we transfer vec[] and matr[][] to the function sub_1(). To be specific, we transfer the addresses of vec[] and matr[][] to sub_1().

In **line d** we have the function definition of sub_1(). The **int** vec[] is a pointer to an integer. Alternatively we could write **int** $*$vec. The first version is better. It shows that it is a vector of several integers, but not how many. The second version could equally well be used to transfer the address to a single integer element. Such a declaration does not distinguish between the two cases.

The next definition is **int** matr[][5]. This is a pointer to a vector of five elements and the compiler must be told that each vector element contains five integers. Here an alternative version could be int ($*$matr)[5] which clearly specifies that matr is a pointer to a vector of five integers.

There is at least one drawback with such a matrix declaration. If we want to change the dimension of the matrix and replace 5 by something else we have to do the same change in all functions where this matrix occurs.

There is another point to note regarding the declaration of variables in a function which includes vectors and matrices. When the execution of a function terminates, the memory required for the variables is released. In the present case memory for all variables in main() are reserved during the whole program execution, but variables which are declared in sub_1() are released when the execution returns to main().

4.3.2 Runtime declarations of vectors and matrices in C++

As mentioned in the previous subsection a fixed size declaration of vectors and matrices before compilation is in many cases bad. You may not know beforehand the actually needed sizes of vectors and matrices. In large projects where memory is a limited factor it could be important to reduce memory requirement for matrices which are not used any more. In C an C++ it is possible and common to postpone size declarations of arrays untill you really know what you need and also release memory reservations when it is not needed any more. The details are shown in Table 4.3.

In **line a** we declare a pointer to an integer which later will be used to store an address to the first element of a vector. Similarily, **line b** declares a pointer-to-a-pointer which will contain the address to a pointer of row vectors, each with col integers. This will then become a matrix[col][col]

In **line c** we read in the size of vec[] and matr[][] through the numbers row and col.

Next we reserve memory for the vector in **line d**. In **line e** we use a user-defined function to reserve necessary memory for matrix[row][col] and again matr contains the address to the reserved memory location.

The remaining part of the function main() are as in the previous case down to **line f**. Here we have a call to a user-defined function which releases the reserved memory of the matrix. In this case this is not done automatically.

Table 4.2: Matrix handling program where arrays are defined at compilation time

```
int main()
{
    int k,m, row = 3, col = 5;
    int vec[5];                                            // line a
    int matr[3][5];                                        // line b

    for(k = 0; k < col; k++) vec[k] = k;                   // data into vector[]
    for(m = 0; m < row; m++) {                             // data into matr[][]
        for(k = 0; k < col ; k++) matr[m][k] = m + 10 * k;
    }
    printf("\n\nVector data in main():\n");                // print vector data
    for(k = 0; k < col; k++) printf("vector[%d] = %d ",k, vec[k]);
     printf("\n\nMatrix data in main():");
    for(m = 0; m < row; m++) {
        printf("\n");
        for(k = 0; k < col; k++)
            printf("matr[%d][[%d] = %d ",m,k,matr[m][k]);
        }
    }
    printf("\n");
    sub_1(row, col, vec, matr);                            // line c
    return 0;
}  // End: function main()

void sub_1(int row, int col, int vec[], int matr[][5])     // line d
{
    int k,m;

    printf("\n\nVector data in sub_1():\n");               // print vector data
    for(k = 0; k < col; k++) printf("vector[%d] = %d ",k, vec[k]);
     printf("\n\nMatrix data in sub_1():");
    for(m = 0; m < row; m++) {
        printf("\n");
        for(k = 0; k < col; k++) {
            printf("matr[%d][[%d] = %d ",m, k, matr[m][k]);
        }
    }
    printf("\n");
}  // End: function sub_1()
```

Table 4.3: Matrix handling program with dynamic array allocation.

```
int main()
{
    int *vec;                                                        // line a
    int **matr;                                                      // line b
    int m, k, row, col, total = 0;

    printf("\n\nRead in number of rows = ");                         // line c
    scanf("%d",&row);
    printf("\n\nRead in number of column = ");
    scanf("%d", &col);

    vec = new int [col];                                             // line d
    matr = (int **)matrix(row, col, sizeof(int));                    // line e
    for(k = 0; k < col; k++) vec[k] = k;                   // store data in vector[]
    for(m = 0; m < row; m++) {                             // store data in array[][]
        for(k = 0; k < col; k++) matr[m][k] = m + 10 * k;
    }
    printf("\n\nVector data in main():\n");                // print vector data
    for(k = 0; k < col; k++) printf("vector[%d] = %d ",k,vec[k]);
    printf("\n\nArray data in main():");
    for(m = 0; m < row; m++) {
        printf("\n");
        for(k = 0; k < col; k++) {
            printf("matrix[%d][[%d] = %d ",m, k, matr[m][k]);
        }
    }
    printf("\n");
    for(m = 0; m < row; m++) {                             // access the array
        for(k = 0; k < col; k++) total += matr[m][k];
    }
    printf("\n\nTotal = %d\n",total);
    sub_1(row, col, vec, matr);
    free_matrix((void **)matr);                                      // line f
    delete [] vec;                                                   // line g
    return 0;
} // End: function main()

void sub_1(int row, int col, int vec[], int **matr)                  // line h
{
    int k,m;

    printf("\n\nVector data in sub_1():\n");               // print vector data
    for(k = 0; k < col; k++) printf("vector[%d] = %d ",k, vec[k]);
    printf("\n\nMatrix data in sub_1():");
    for(m = 0; m < row; m++) {
        printf("\n");
        for(k = 0; k < col; k++) {
            printf("matrix[%d][[%d] = %d ",m,k,matr[m][k]);
        }
    }
    printf("\n");
} // End: function sub_1()
```

In **line g** the same procedure is performed for vec[]. In this case the standard C++ library has the necessary function.

Next, in **line h** an important difference from the previous case occurs. First, the vector declaration is the same, but the matr declaration is quite different. The corresponding parameter in the call to sub_1[] in **line g** is a double pointer. Consequently, matr in **line h** must be a double pointer.

Except for this difference sub_1() is the same as before. The new feature in Table 4.3 is the call to the user-defined functions **matrix** and **free_matrix**. These functions are defined in the library file **lib.cpp**. The code for the dynamic memory allocation is given below.

`http://www.fys.uio.no/compphys/cp/programs/FYS3150/cpp/cpluspluslibrary/lib.cpp`

```
 /*
  * The function
  *       void  **matrix()
  * reserves dynamic memory for a two-dimensional matrix
  * using the C++ command new . No initialization of the elements.
  * Input data:
  *  int row      - number of  rows
  *  int col      - number of columns
  *  int num_bytes- number of bytes for each
  *                 element
  * Returns a void  **pointer to the reserved memory location.
  */

void **matrix(int row, int col, int num_bytes)
 {
 int      i, num;
 char     **pointer, *ptr;

 pointer = new(nothrow) char* [row];
 if(!pointer) {
   cout << "Exception handling: Memory allocation failed";
   cout << " for "<< row << "row addresses !" << endl;
   return NULL;
 }
 i = (row * col * num_bytes)/sizeof(char);
 pointer[0] = new(nothrow) char [i];
 if(!pointer[0]) {
   cout << "Exception handling: Memory allocation failed";
   cout << " for address to " << i << " characters !" << endl;
   return NULL;
 }
 ptr = pointer[0];
 num = col * num_bytes;
 for(i = 0; i < row; i++, ptr += num )   {
   pointer[i] = ptr;
 }
 return  (void **)pointer;
 } // end: function void **matrix()
```

As an alternative, you could write your own allocation and deallocation of matrices. This can be done rather straightforwardly with the following statements. Recall first that a matrix is represented by a double pointer that points to a contiguous memory segment holding a sequence of double* pointers in

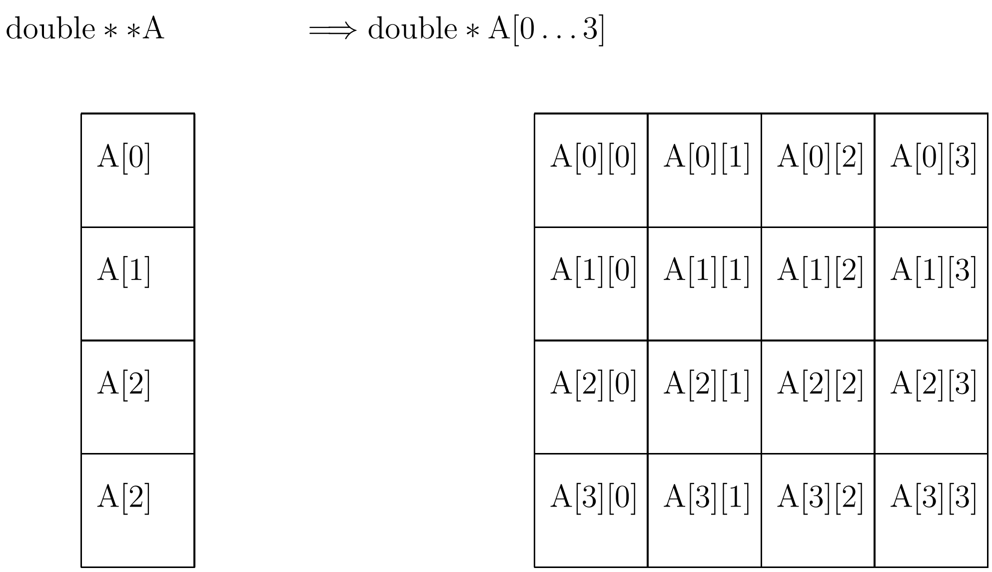


Figure 4.2: Conceptual representation of the allocation of a matrix in C++.

case our matrix is a double precision variable. Then each double* pointer points to a row in the matrix. A declaration like **double**$**$ A; means that A$[i]$ is a pointer to the $i+1$-th row A$[i]$ and A$[i][j]$ is matrix entry (i,j). The way we would allocate memory for such a matrix of dimensionality $n\times n$ is for example using the following piece of code

```
int n;
double **  A;

A = new double*[n]
for ( i = 0; i < n; i++)
    A[i] = new double[N];
```

When we declare a matrix (a two-dimensional array) we must first declare an array of double variables. To each of this variables we assign an allocation of a single-dimensional array. A conceptual picture on how a matrix **A** is stored in memory is shown in Fig. 4.2.

Allocated memory should always be deleted when it is no longer needed. We free memory using the statements

```
for ( i = 0; i < n; i++)
    delete[] A[i];
delete[] A;
```

delete [] A;, which frees an array of pointers to matrix rows.

However, including a library like Blitz++ `http://www.oonumerics.org` makes life much easier when dealing with matrices. This is discussed in the appendix.

4.3.3 Matrix operations and C++ and Fortran features of matrix handling

Many program libraries for scientific computing are written in Fortran, often also in older version such Fortran 77. When using functions from such program libraries, there are some differences between C++ and Fortran encoding of matrices and vectors worth noticing. Here are some simple guidelines in order to avoid some of the most common pitfalls.

First of all, when we think of an $n \times n$ matrix in Fortran and C++, we typically would have a mental picture of a two-dimensional block of stored numbers. The computer stores them however as sequential strings of numbers. The latter could be stored as row-major order or column-major order. What do we mean by that? Recalling that for our matrix elements a_{ij}, i refers to rows and j to columns, we could store a matrix in the sequence $a_{11}a_{12}\dots a_{1n}a_{21}a_{22}\dots a_{2n}\dots a_{nn}$ if it is row-major order (we go along a given row i and pick up all column elements j) or it could be stored in column-major order $a_{11}a_{21}\dots a_{n1}a_{12}a_{22}\dots a_{n2}\dots a_{nn}$.

Fortran stores matrices in the latter way, i.e., by column-major, while C++ stores them by row-major. It is crucial to keep this in mind when we are dealing with matrices, because if we were to organize the matrix elements in the wrong way, important properties like the transpose of a real matrix or the inverse can be wrong, and obviously yield wrong physics. Fortran subscripts begin typically with 1, although it is no problem in starting with zero, while C++ starts with 0 for the first element. This means that $A(1,1)$ in Fortran is equivalent to $A[0][0]$ in C++. Moreover, since the sequential storage in memory means that nearby matrix elements are close to each other in the memory locations (and thereby easier to fetch) , operations involving e.g., additions of matrices may take more time if we do not respect the given ordering.

To see this, consider the following coding of matrix addition in C++ and Fortran. We have $n \times n$ matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ and we wish to evaluate $\mathbf{A} = \mathbf{B} + \mathbf{C}$ according to Eq. (4.2). In C++ this would be coded like

```
for(i=0 ; i < n ; i++) {
   for(j=0 ; j < n ; j++) {
      a[i][j]=b[i][j]+c[i][j]
   }
}
```

while in Fortran we would have

```
DO  j=1,  n
   DO i=1, n
      a(i,j)=b(i,j)+c(i,j)
   ENDDO
ENDDO
```

Fig. 4.3 shows how a 3×3 matrix $\mathbf{A}$ is stored in both row-major and column-major ways.

Interchanging the order of i and j can lead to a considerable enhancement in process time. In Fortran we write the above statements in a much simpler way a=b+c. However, the addition still involves $\sim n^2$ operations. Matrix multiplication or taking the inverse requires $\sim n^3$ operations. The matrix multiplication of Eq. (4.4) of two matrices $\mathbf{A} = \mathbf{BC}$ could then take the following form in C++

```
for(i=0 ; i < n ; i++) {
   for(j=0 ; j < n ; j++) {
      for(k=0 ; k < n ; k++) {
         a[i][j]+=b[i][k]*c[k][j]
      }
```

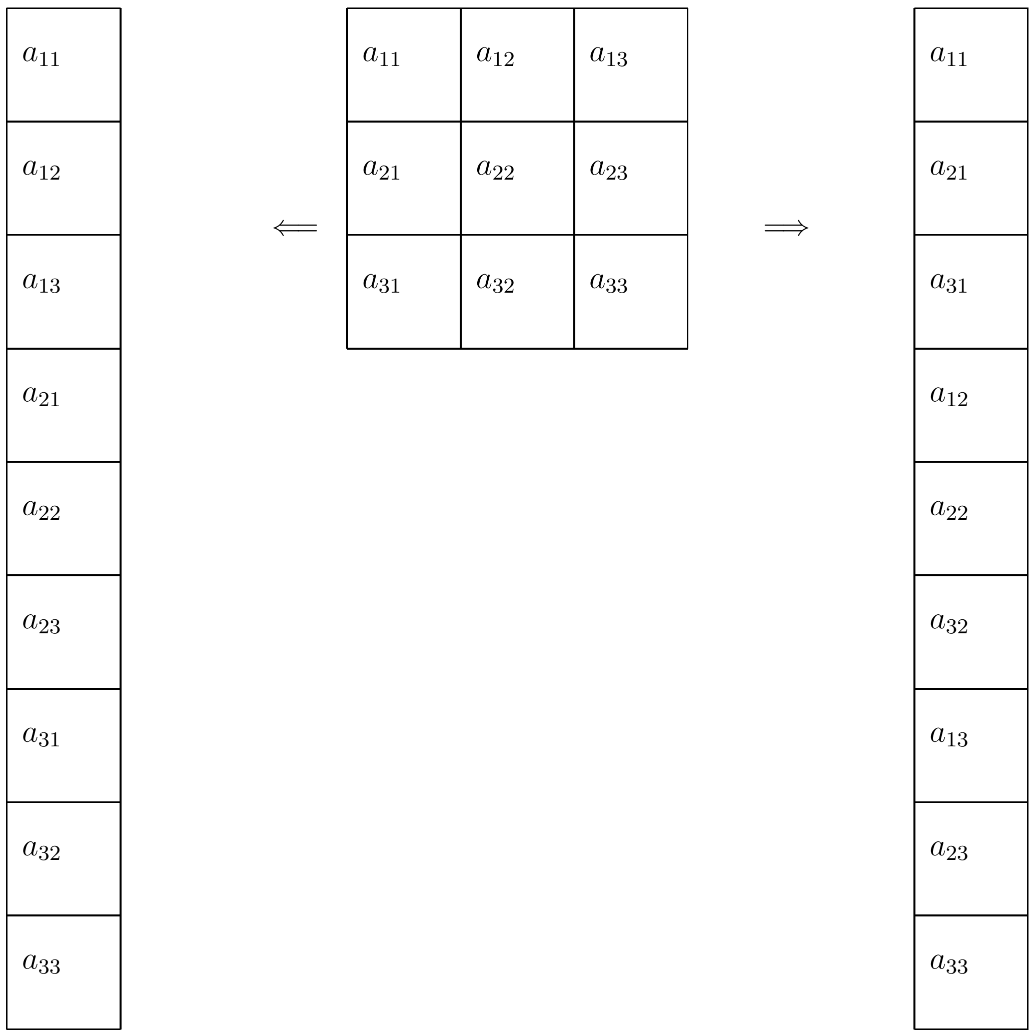


Figure 4.3: Row-major storage of a matrix to the left (C++ way) and column-major to the right (Fortran way).

```
    }
}
```

and in Fortran we have

```
DO   j=1,   n
   DO  i=1,  n
      DO k  =  1,  n
         a(i,j)=a(i,j)+b(i,k)*c(i,j)
      ENDDO
   ENDDO
ENDDO
```

However, Fortran has an intrisic function called MATMUL, and the above three loops can be coded in a single statement a=MATMUL(b,c). Fortran contains several array manipulation statements, such as dot product of vectors, the transpose of a matrix etc etc. The outer product of two vectors is however not included in Fortran. The coding of Eq. (4.6) takes then the following form in C++

```
for(i=0 ;  i  < n ;  i++) {
   for(j=0 ;  j  < n ;  j++) {
      a[i][j]+=x[i]*y[j]
   }
}
```

and in Fortran we have

```
DO   j=1,   n
   DO  i=1,  n
      a(i,j)=a(i,j)+x(j)*y(i)
   ENDDO
ENDDO
```

A matrix-matrix multiplication of a general $n \times n$ matrix with

$$a(i,j) = a(i,j) + b(i,k) * c(i,j),$$

in its inner loops requires a multiplication and an addition. We define now a flop (floating point operation) as one of the following floating point arithmetic operations, viz addition, subtraction, multiplication and division. The above two floating point operations (flops) are done n^3 times meaning that a general matrix multiplication requires $2n^3$ flops if we have a square matrix. If we assume that our computer performs 10^9 flops per second, then to perform a matrix multiplication of a 1000×1000 case should take two seconds. This can be reduced if we multiply two matrices which are upper triangular such as

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{44} \end{pmatrix}.$$

The multiplication of two upper triangular matrices $\mathbf{BC}$ yields another upper triangular matrix $\mathbf{A}$, resulting in the following C++ code

```
for(i=0 ;  i  < n ;  i++) {
   for(j=i ;  j  < n ;  j++) {
      for(k=i ;  k  < j ;  k++) {
```

```
          a[i][j]+=b[i][k]*c[k][j]
        }
      }
    }
```

The fact that we have the constraint $i \leq j$ leads to the requirement for the computation of a_{ij} of $2(j-i+1)$ flops. The total number of flops is then

$$\sum_{i=1}^{n}\sum_{j=1}^{n}2(j-i+1) = \sum_{i=1}^{n}\sum_{j=1}^{n-i+1}2j \approx \sum_{i=1}^{n}\frac{2(n-i+1)^2}{2},$$

where we used that $\sum_{j=1}^{n} j = n(n+1)/2 \approx n^2/2$ for large n values. Using in addition that $\sum_{j=1}^{n} j^2 \approx n^3/3$ for large n values, we end up with approximately $n^3/3$ flops for the multiplication of two upper triangular matrices. This means that if we deal with matrix multiplication of upper triangular matrices, we reduce the number of flops by a factor six if we code our matrix multiplication in an efficient way.

It is also important to keep in mind that computers are finite, we can thus not store infinitely large matrices. To calculate the space needed in memory for an $n \times n$ matrix with double precision, 64 bits or 8 bytes for every matrix element, one needs simply compute $n \times n \times 8$ bytes . Thus, if $n = 10000$, we will need close to 1GB of storage. Decreasing the precision to single precision, only halves our needs.

A further point we would like to stress, is that one should in general avoid fixed (at compilation time) dimensions of matrices. That is, one could always specify that a given matrix $\mathbf{A}$ should have size $A[100][100]$, while in the actual execution one may use only $A[10][10]$. If one has several such matrices, one may run out of memory, while the actual processing of the program does not imply that. Thus, we will always recommend that you use dynamic memory allocation, and deallocation of arrays when they are no longer needed. In Fortran one uses the intrisic functions **ALLOCATE** and **DEALLOCATE**, while C++ employs the functions **new** and **delete**.

Fortran allocate statement and mathematical operations on arrays

An array is declared in the declaration section of a program, module, or procedure using the dimension attribute. Examples include

```
REAL, DIMENSION (10) :: x,y
REAL, DIMENSION (1:10) :: x,y
INTEGER, DIMENSION (-10:10) :: prob
INTEGER, DIMENSION (10,10) :: spin
```

The default value of the lower bound of an array is 1. For this reason the first two statements are equivalent to the first. The lower bound of an array can be negative. The last two statements are examples of two-dimensional arrays.

Rather than assigning each array element explicitly, we can use an array constructor to give an array a set of values. An array constructor is a one-dimensional list of values, separated by commas, and delimited by "(/" and "/)". An example is

```
a(1:3) = (/ 2.0, -3.0, -4.0 /)
```

is equivalent to the separate assignments

```
a(1) = 2.0
a(2) = -3.0
a(3) = -4.0
```

One of the better features of Fortran is dynamic storage allocation. That is, the size of an array can be changed during the execution of the program. To see how the dynamic allocation works in Fortran, consider the following simple example where we set up a 4×4 unity matrix.

```
        ......
        IMPLICIT NONE
!       The definition of the matrix, using dynamic allocation
        REAL, ALLOCATABLE, DIMENSION(:,:) :: unity
!       The size of the matrix
        INTEGER :: n
!       Here we set the dim n=4
        n=4
!   Allocate now place in memory for the matrix
        ALLOCATE ( unity(n,n) )
!   all elements are set equal zero
        unity=0.
!       setup identity matrix
        DO i=1,n
           unity(i,i)=1.
        ENDDO
        DEALLOCATE ( unity)
        .......
```

We always recommend to use the deallocation statement, since this frees space in memory. If the matrix is transferred to a function from a calling program, one can transfer the dimensionality n of that matrix with the call. Another possibility is to determine the dimensionality with the `SIZE` function. Writing a statement like n=SIZE(unity,DIM=1) gives the number of rows, while using DIM=2 gives the number of columns. Note however that this involves an extra call to a function. If speed matters, one should avoid such calls.

4.4 Linear Systems

In this section we outline some of the most used algorithms to solve sets of linear equations. These algorithms are based on Gaussian elimination [27, 30] and will allow us to catch several birds with a stone. We will show how to rewrite a matrix $\mathbf{A}$ in terms of an upper and a lower triangular matrix, from which we easily can solve linear equation, compute the inverse of $\mathbf{A}$ and obtain the determinant. We start with Gaussian elimination, move to the more efficient LU-algorithm, which forms the basis for many linear algebra applications, and end the discussion with special cases such as the Cholesky decomposition and linear system of equations with a tridiagonal matrix.

We begin however with an example which demonstrates the importance of being able to solve linear equations. Suppose we want to solve the following boundary value equation

$$-\frac{d^2u(x)}{dx^2} = f(x, u(x)),$$

with $x \in (a, b)$ and with boundary conditions $u(a) = u(b) = 0$. We assume that f is a continuous function in the domain $x \in (a, b)$. Since, except the few cases where it is possible to find analytic solutions, we will seek after approximate solutions, we choose to represent the approximation to the second derivative from the previous chapter

$$f'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} + O(h^2).$$

We subdivide our interval $x \in (a, b)$ into n subintervals by setting $x_i = ih$, with $i = 0, 1, \ldots, n+1$. The step size is then given by $h = (b-a)/(n+1)$ with $n \in \mathbb{N}$. For the internal grid points $i = 1, 2, \ldots n$ we replace the differential operator with the above formula resulting in

$$u''(x_i) \approx \frac{u(x_i + h) - 2u(x_i) + u(x_i - h)}{h^2},$$

which we rewrite as

$$u_i^{''} \approx \frac{u_{i+1} - 2u_i + u_{i-i}}{h^2}.$$

We can rewrite our original differential equation in terms of a discretized equation with approximations to the derivatives as

$$-\frac{u_{i+1} - 2u_i + u_{i-i}}{h^2} = f(x_i, u(x_i)),$$

with $i = 1, 2, \ldots, n$. We need to add to this system the two boundary conditions $u(a) = u_0$ and $u(b) = u_{n+1}$. If we define a matrix

$$\mathbf{A} = \frac{1}{h^2}\begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & \dots & \dots & \dots & \dots & \dots \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{pmatrix}$$

and the corresponding vectors $\mathbf{u} = (u_1, u_2, \ldots, u_n)^T$ and $\mathbf{f}(\mathbf{u}) = f(x_1, x_2, \ldots, x_n, u_1, u_2, \ldots, u_n)^T$ we can rewrite the differential equation including the boundary conditions as a system of linear equations with a large number of unknowns

$$\mathbf{Au} = \mathbf{f}(\mathbf{u}). \tag{4.7}$$

We assume that the solution u exists and is unique for the exact differential equation, viz that the boundary value problem has a solution. But the discretization of the above differential equation leads to several questions, such as how well does the approximate solution resemble the exact one as $h \to 0$, or does a given small value of h allow us to establish existence and uniqueness of the solution.

Here we specialize to two particular cases. Assume first that the function f does not depend on $u(x)$. Then our linear equation reduces to

$$\mathbf{Au} = \mathbf{f}, \tag{4.8}$$

which is nothing but a simple linear equation with a tridiagonal matrix $\mathbf{A}$. We will solve such a system of equations in subsection 4.4.3.

If we assume that our boundary value problem is that of a quantum mechanical particle confined by a harmonic oscillator potential, then our function f takes the form (assuming that all constants $m = \hbar =$

$\omega = 1)$ $f(x_i, u(x_i)) = -x_i^2 u(x_i) + 2\lambda u(x_i)$ with λ being the eigenvalue. Inserting this into our equation, we define first a new matrix $\mathbf{A}$ as

$$\mathbf{A} = \begin{pmatrix} \frac{2}{h^2}+x_1^2 & -\frac{1}{h^2} & & & & \\ -\frac{1}{h^2} & \frac{2}{h^2}+x_2^2 & -\frac{1}{h^2} & & & \\ & -\frac{1}{h^2} & \frac{2}{h^2}+x_3^2 & -\frac{1}{h^2} & & \\ & \dots & \dots & \dots & \dots & \dots \\ & & & -\frac{1}{h^2} & \frac{2}{h^2}+x_{n-1}^2 & -\frac{1}{h^2} \\ & & & & -\frac{1}{h^2} & \frac{2}{h^2}+x_n^2 \end{pmatrix}, \tag{4.9}$$

which leads to the following eigenvalue problem

$$\begin{pmatrix} \frac{2}{h^2}+x_1^2 & -\frac{1}{h^2} & & & & \\ -\frac{1}{h^2} & \frac{2}{h^2}+x_2^2 & -\frac{1}{h^2} & & & \\ & -\frac{1}{h^2} & \frac{2}{h^2}+x_3^2 & -\frac{1}{h^2} & & \\ & \dots & \dots & \dots & \dots & \dots \\ & & & -\frac{1}{h^2} & \frac{2}{h^2}+x_{n-1}^2 & -\frac{1}{h^2} \\ & & & & -\frac{1}{h^2} & \frac{2}{h^2}+x_n^2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \\ \\ \\ u_n \end{pmatrix} = 2\lambda \begin{pmatrix} u_1 \\ u_2 \\ \\ \\ \\ u_n \end{pmatrix}.$$

We will solve this type of equations in chapter 12. These lecture notes contain however several other examples of rewriting mathematical expressions into matrix problems. In chapter 7 we show how a set of linear integral equation when discretized can be transformed into a simple matrix inversion problem. The specific example we study in that chapter is the rewriting of Schrödinger's equation for scattering problems. Other examples of linear equations will appear in our discussion of ordinary and partial differential equations.

4.4.1 Gaussian elimination

Any discussion on the solution of linear equations should start with Gaussian elimination. This text is no exception. We start with the linear set of equations

$$\mathbf{Ax} = \mathbf{w}.$$

We assume also that the matrix $\mathbf{A}$ is non-singular and that the matrix elements along the diagonal satisfy $a_{ii} \neq 0$. We discuss later how to handle such cases. In the discussion we limit ourselves again to a matrix $\mathbf{A} \in \mathbb{R}^{4\times 4}$, resulting in a set of linear equations of the form

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix}.$$

or

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= \quad w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= \quad w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= \quad w_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= \quad w_4. \end{aligned}$$

The basic idea of Gaussian elimination is to use the first equation to eliminate the first unknown x_1 from the remaining $n-1$ equations. Then we use the new second equation to eliminate the second unknown x_2

from the remaining $n-2$ equations. With $n-1$ such eliminations we obtain a so-called upper triangular set of equations of the form

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= \quad y_1 \\ b_{22}x_2 + b_{23}x_3 + b_{24}x_4 &= \quad y_2 \\ b_{33}x_3 + b_{34}x_4 &= \quad y_3 \\ b_{44}x_4 &= \quad y_4. \end{aligned}$$

We can solve this system of equations recursively starting from x_n (in our case x_4) and proceed with what is called a backward substitution. This process can be expressed mathematically as

$$x_m = \frac{1}{b_{mm}}\left(y_m - \sum_{k=m+1}^{n} b_{mk}x_k\right) \qquad m = n-1, n-2, \dots, 1.$$

To arrive at such an upper triangular system of equations, we start by eliminating the unknown x_1 for $j = 2, n$. We achieve this by multiplying the first equation by a_{j1}/a_{11} and then subtract the result from the jth equation. We assume obviously that $a_{11} \neq 0$ and that $\mathbf{A}$ is not singular. We will come back to this problem below.

Our actual 4×4 example reads after the first operation

$$\left(\begin{array}{cccc} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & (a_{22} - \frac{a_{21}a_{12}}{a_{11}}) & (a_{23} - \frac{a_{21}a_{13}}{a_{11}}) & (a_{24} - \frac{a_{21}a_{14}}{a_{11}}) \\ 0 & (a_{32} - \frac{a_{31}a_{12}}{a_{11}}) & (a_{33} - \frac{a_{31}a_{13}}{a_{11}}) & (a_{34} - \frac{a_{31}a_{14}}{a_{11}}) \\ 0 & (a_{42} - \frac{a_{41}a_{12}}{a_{11}}) & (a_{43} - \frac{a_{41}a_{13}}{a_{11}}) & (a_{44} - \frac{a_{41}a_{14}}{a_{11}}) \end{array}\right) \left(\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array}\right) = \left(\begin{array}{c} y_1 \\ w_2^{(2)} \\ w_3^{(2)} \\ w_4^{(2)} \end{array}\right).$$

or

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= \quad y_1 \\ a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + a_{24}^{(2)}x_4 &= \quad w_2^{(2)} \\ a_{32}^{(2)}x_2 + a_{33}^{(2)}x_3 + a_{34}^{(2)}x_4 &= \quad w_3^{(2)} \\ a_{42}^{(2)}x_2 + a_{43}^{(2)}x_3 + a_{44}^{(2)}x_4 &= \quad w_4^{(2)}, \end{aligned} \tag{4.10}$$

with the new coefficients

$$b_{1k} = a_{1k}^{(1)} \; k = 1, \dots, n,$$

where each $a_{1k}^{(1)}$ is equal to the original a_{1k} element. The other coefficients are

$$a_{jk}^{(2)} = a_{jk}^{(1)} - \frac{a_{j1}^{(1)}a_{1k}^{(1)}}{a_{11}^{(1)}} \; j, k = 2, \dots, n,$$

with a new right-hand side given by

$$y_1 = w_1^{(1)}, \; w_j^{(2)} = w_j^{(1)} - \frac{a_{j1}^{(1)}w_1^{(1)}}{a_{11}^{(1)}} \; j = 2, \dots, n.$$

We have also set $w_1^{(1)} = w_1$, the original vector element. We see that the system of unknowns $x_1, \dots, x_n$ is transformed into an $(n-1) \times (n-1)$ problem.

This step is called forward substitution. Proceeding with these substitutions, we obtain the general expressions for the new coefficients

$$a_{jk}^{(m+1)} = a_{jk}^{(m)} - \frac{a_{jm}^{(m)} a_{mk}^{(m)}}{a_{mm}^{(m)}} \; j,k = m+1,\dots,n,$$

with $m = 1,\dots,n-1$ and a right-hand side given by

$$w_j^{(m+1)} = w_j^{(m)} - \frac{a_{jm}^{(m)} w_m^{(m)}}{a_{mm}^{(m)}} \; j = m+1,\dots,n.$$

This set of $n-1$ elimations leads us to Eq. (4.10), which is solved by back substitution. If the arithmetics is exact and the matrix $\mathbf{A}$ is not singular, then the computed answer will be exact. However, as discussed in the two preceeding chapters, computer arithmetics is not necessarily exact. We will always have to cope with truncations and possible losses of precision. Even though the matrix elements along the diagonal are not zero, numerically small numbers may appear and subsequent divisions may lead to large numbers, which, if added to a small number may yield losses of precision. Suppose for example that our first division in $(a_{22} - a_{21}a_{12}/a_{11})$ results in -10^{-7} and that a_{22} is one. one. We are then adding $10^7 + 1$. With single precision this results in 10^7. Already at this stage we see the potential for producing wrong results.

The solution to this set of problems is called pivoting, and we distinguish between partial and full pivoting. Pivoting means that if small values (especially zeros) do appear on the diagonal we remove them by rearranging the matrix and vectors by permuting rows and columns. As a simple example, let us assume that at some stage during a calculation we have the following set of linear equations

$$\begin{pmatrix} 1 & 3 & 4 & 6 \\ 0 & 10^{-8} & 198 & 19 \\ 0 & -91 & 51 & 9 \\ 0 & 7 & 76 & 541 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}.$$

The element at row $i = 2$ and column 2 is 10^{-8} and may cause problems for us in the next forward substitution. The element $i = 2, j = 3$ is the largest in the second row and the element $i = 3, j = 2$ is the largest in the third row. The small element can be removed by rearranging the rows and/or columns to bring a larger value into the $i = 2, j = 2$ element.

In partial or column pivoting, we rearrange the rows of the matrix and the right-hand side to bring the numerically largest value in the column onto the diagonal. For our example matrix the largest value of column two is in element $i = 3, j = 2$ and we interchange rows 2 and 3 to give

$$\begin{pmatrix} 1 & 3 & 4 & 6 \\ 0 & -91 & 51 & 9 \\ 0 & 10^{-8} & 198 & 19 \\ 0 & 7 & 76 & 541 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_3 \\ y_2 \\ y_4 \end{pmatrix}.$$

Note that our unknown variables x_i remain in the same order which simplifies the implementation of this procedure. The right-hand side vector, however, has been rearranged. Partial pivoting may be implemented for every step of the solution process, or only when the diagonal values are sufficiently small as to potentially cause a problem. Pivoting for every step will lead to smaller errors being introduced through numerical inaccuracies, but the continual reordering will slow down the calculation.

The philosophy behind full pivoting is much the same as that behind partial pivoting. The main difference is that the numerically largest value in the column or row containing the value to be replaced. In our example above the magnitude of element $i = 2, j = 3$ is the greatest in row 2 or column 2. We could rearrange the columns in order to bring this element onto the diagonal. This will also entail a rearrangement of the solution vector x. The rearranged system becomes, interchanging columns two and three,

$$\begin{pmatrix} 1 & 6 & 3 & 4 \\ 0 & 198 & 10^{-8} & 19 \\ 0 & 51 & -91 & 9 \\ 0 & 76 & 7 & 541 \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_2 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}.$$

The ultimate degree of accuracy can be provided by rearranging both rows and columns so that the numerically largest value in the submatrix not yet processed is brought onto the diagonal. This process may be undertaken for every step, or only when the value on the diagonal is considered too small relative to the other values in the matrix. In our case, the matrix element at $i = 4, j = 4$ is the largest. We could here interchange rows two and four and then columns two and four to bring this matrix element at the diagonal position $i = 2, j = 2$. When interchanging columns and rows, one needs to keep track of all permutations performed. Partial and full pivoting are discussed in most texts on numerical linear algebra. For an in depth discussion we recommend again the text of Golub and Van Loan [27], in particular chapter three. See also the discussion of chapter two in Ref. [36]. The library functions you end up using, be it via Matlab, the library included with this text or other ones, do all include pivoting.

If it is not possible to rearrange the columns or rows to remove a zero from the diagonal, then the matrix A is singular and no solution exists.

Gaussian elimination requires however many floating point operations. An $n \times n$ matrix requires for the simultaneous solution of a set of r different right-hand sides, a total of $n^3/3 + rn^2 - n/3$ multiplications. Adding the cost of additions, we end up with $2n^3/3 + O(n^2)$ floating point operations, see Kress [30] for a proof. An $n \times n$ matrix of dimensionalty $n = 10^3$ requires, on a modern PC with a processor that allows for something like 10^9 floating point operations per second (flops), approximately one second. If you increase the size of the matrix to $n = 10^4$ you need 1000 seconds, or roughly 16 minutes.

Although the direct Gaussian elmination algorithm allows you to compute the determinant of $\mathbf{A}$ via the product of the diagonal matrix of the triangular matrix, it is seldomly used in normal applications. The more practical elimination is provided by what is called lower and upper decomposition. Once decomposed, one can use this matrix to solve many other linear systems which use the same matrix $\mathbf{A}$, viz with different right-hand sides. With an LU decomposed matrix, the number of floating point operations for solving a set of linear equations scales as $O(n^2)$. One should however remember that to obtain the LU decompsed matrix requires roughly $O(n^3)$ floating point operations. Finally, LU decomposition allows for an efficient computation of the inverse of $\mathbf{A}$.

4.4.2 LU decomposition of a matrix

A frequently used form of Gaussian elimination is L(ower)U(pper) factorisation also known as LU Decomposition or Crout or Dolittle factorisation. In this section we describe how one can decompose a matrix A in terms of a matrix L with elements only below the diagonal (and thereby the naming lower) and a matrix U which contains both the diagonal and matrix elements above the diagonal (leading to the labelling upper). Consider again the matrix $\mathbf{A}$ given in Eq. (4.1). The LU decomposition method means that we can rewrite this matrix as the product of two matrices **L** and **U** where

$$\mathbf{A} = \mathbf{LU} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix}. \quad (4.11)$$

LU decomposition forms the backbone of other algorithms in linear algebra, such as the solution of linear equations given by

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 = &\quad w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 = &\quad w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 = &\quad w_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 = &\quad w_4. \end{aligned}$$

The above set of equations is conveniently solved by using LU decomposition as an intermediate step see the next subsection for more details on how to solve linear equations with an LU decomposed matrix.

The matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ has an LU factorization if the determinant is different from zero. If the LU factorization exists and **A** is non-singular, then the LU factorization is unique and the determinant is given by

$$det\{\mathbf{A}\} = u_{11}u_{22}\dots u_{nn}.$$

For a proof of this statement, see chapter 3.2 of Ref. [27].

The algorithm for obtaining L and U is actually quite simple. We start always with the first column. In our simple (4×4) case we obtain then the following equations for the first column

$$\begin{aligned} a_{11} &= u_{11} \\ a_{21} &= l_{21}u_{11} \\ a_{31} &= l_{31}u_{11} \\ a_{41} &= l_{41}u_{11}, \end{aligned}$$

which determine the elements u_{11}, l_{21}, l_{31} and u_{41} in **L** and **U**. Writing out the equations for the second column we get

$$\begin{aligned} a_{12} &= u_{12} \\ a_{22} &= l_{21}u_{12} + u_{22} \\ a_{32} &= l_{31}u_{12} + l_{32}u_{22} \\ a_{42} &= l_{41}u_{12} + l_{42}u_{22}. \end{aligned}$$

Here the unknowns are u_{12}, u_{22}, l_{32} and l_{42} which all can be evaluated by means of the results from the first column and the elements of **A**. Note an important feature. When going from the first to the second column we do not need any further information from the matrix elements a_{i1}. This is a general property throughout the whole algorithm. Thus the memory locations for the matrix **A** can be used to store the calculated matrix elements of **L** and **U**. This saves memory.

We can generalize this procedure into three equations

$$\begin{aligned} i<j: &\quad l_{i1}u_{1j} + l_{i2}u_{2j} + \dots + l_{ii}u_{ij} = &\quad a_{ij} \\ i=j: &\quad l_{i1}u_{1j} + l_{i2}u_{2j} + \dots + l_{ii}u_{jj} = &\quad a_{ij} \\ i>j: &\quad l_{i1}u_{1j} + l_{i2}u_{2j} + \dots + l_{ij}u_{jj} = &\quad a_{ij} \end{aligned}$$

which gives the following algorithm:
Calculate the elements in **L** and **U** columnwise starting with column one. For each column (j):

– Compute the first element u_{1j} by

$$u_{1j} = a_{1j}.$$

– Next, we calculate all elements $u_{ij}, i = 2, \dots, j-1$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}.$$

– Then calculate the diagonal element u_{jj}

$$u_{jj} = a_{jj} - \sum_{k=1}^{j-1} l_{jk} u_{kj}. \tag{4.12}$$

– Finally, calculate the elements $l_{ij}, i > j$

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right), \tag{4.13}$$

The algorithm is known as Doolittle's algorithm since the diagonal matrix elements of $\mathbf{L}$ are 1. For the case where the diagonal elements of $\mathbf{U}$ are 1, we have what is called Crout's algorithm. For the case where $\mathbf{U} = \mathbf{L}^T$ so that $u_{ii} = l_{ii}$ for $1 \le i \le n$ we can use what is called the Cholesky factorization algorithm. In this case the matrix $\mathbf{A}$ has to fulfil several features; namely, it should be real, symmetric and positive definite. A matrix is positive definite if the quadratic form $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$. Establishing this feature is not easy since it implies the use of an arbitrary vector $\mathbf{x} \neq 0$. If the matrix is positive definite and symmetric, its eigenvalues are always real and positive. We discuss the Cholesky factorization below.

A crucial point in the LU decomposition is obviously the case where u_{jj} is close to or equals zero, a case which can lead to serious problems. Consider the following simple 2×2 example taken from Ref. [29]

$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

The algorithm discussed above fails immediately, the first step simple states that $u_{11} = 0$. We could change slightly the above matrix by replacing 0 with 10^{-20} resulting in

$$\mathbf{A} = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix},$$

yielding

$$\begin{aligned} u_{11} &= 10^{-20} \\ l_{21} &= 10^{20} \end{aligned}$$

and $u_{12} = 1$ and

$$u_{11} = a_{11} - l_{21} = 1 - 10^{20},$$

we obtain

$$\mathbf{L} = \begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix},$$

and

$$\mathbf{U} = \begin{pmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{pmatrix},$$

With the change from 0 to a small number like 10^{-20} we see that the LU decomposition is now stable, but it is not backward stable. What do we mean by that? First we note that the matrix $\mathbf{U}$ has an element $u_{22} = 1 - 10^{20}$. Numerically, since we do have a limited precision, which for double precision is approximately $\epsilon_M \sim 10^{-16}$ it means that this number is approximated in the machine as $u_{22} \sim -10^{20}$ resulting in a machine representation of the matrix as

$$\mathbf{U} = \left(\begin{array}{cc} 10^{-20} & 1 \\ 0 & -10^{20} \end{array} \right).$$

If we multiply the matrices $\mathbf{LU}$ we have

$$\left(\begin{array}{cc} 1 & 0 \\ 10^{20} & 1 \end{array} \right) \left(\begin{array}{cc} 10^{-20} & 1 \\ 0 & -10^{20} \end{array} \right) = \left(\begin{array}{cc} 10^{-20} & 1 \\ 1 & 0 \end{array} \right) \neq \mathbf{A}.$$

We do not get back the original matrix $\mathbf{A}$!

The solution is pivoting (interchanging rows in this case) around the largest element in a column j. Then we are actually decomposing a rowwise permutation of the original matrix $\mathbf{A}$. The key point to notice is that Eqs. (4.12) and (4.13) are equal except for the case that we divide by u_{jj} in the latter one. The upper limits are always the same $k = j - 1 (= i - 1)$. This means that we do not have to choose the diagonal element u_{jj} as the one which happens to fall along the diagonal in the first instance. Rather, we could promote one of the undivided l_{ij}'s in the column $i = j + 1, \dots N$ to become the diagonal of U. The partial pivoting in Crout's or Doolittle's methods means then that we choose the largest value for u_{jj} (the pivot element) and then do the divisions by that element. Then we need to keep track of all permutations performed. For the above matrix $\mathbf{A}$ it would have sufficed to interchange the two rows and start the LU decomposition with

$$\mathbf{A} = \left(\begin{array}{cc} 1 & 1 \\ 0 & 1 \end{array} \right).$$

The error which is done in the LU decomposition of an $n \times n$ matrix if no zero pivots are encountered is given by, see chapter 3.3 of Ref. [27],

$$\mathbf{LU} = \mathbf{A} + \mathbf{H},$$

with

$$|\mathbf{H}| \leq 3(n-1)\mathbf{u}\left(|\mathbf{A}| + |\mathbf{L}||\mathbf{U}|\right) + O(\mathbf{u}^2),$$

with $|\mathbf{H}|$ being the absolute value of a matrix and $\mathbf{u}$ is the error done in representing the matrix elements of the matrix $\mathbf{A}$ as floating points in a machine with a given precision ϵ_M, viz. every matrix element of $\mathbf{u}$ is

$$|fl(a_{ij}) - a_{ij}| \leq u_{ij},$$

with $|u_{ij}| \leq \epsilon_M$ resulting in

$$|fl(\mathbf{A}) - \mathbf{A}| \leq \mathbf{u}|\mathbf{A}|.$$

The programs which perform the above described LU decomposition are called as follows

C++: ludcmp(double $**$a, int n, int $*$indx, double $*$d)
Fortran: CALL lu_decompose(a, n, indx, d)

Both the C++ and Fortran 90/95 programs receive as input the matrix to be LU decomposed. In C++ this is given by the double pointer $**$a. Further, both functions need the size of the matrix n. It returns the variable d, which is ± 1 depending on whether we have an even or odd number of row interchanges, a pointer $indx$ that records the row permutation which has been effected and the LU decomposed matrix. Note that the original matrix is destroyed.

Cholesky's factorization

If the matrix A is real, symmetric and positive definite, then it has a unique factorization (called Cholesky factorization)

$$A = LU = LL^T$$

where L^T is the upper matrix, implying that

$$L^T_{ij} = L_{ji}.$$

The algorithm for the Cholesky decomposition is a special case of the general LU-decomposition algorithm. The algorithm of this decomposition is as follows

- Calculate the diagonal element L_{ii} by setting up a loop for $i = 0$ to $i = n-1$ (C++ indexing of matrices and vectors)

$$L_{ii} = \left(A_{ii} - \sum_{k=0}^{i-1} L^2_{ik}\right)^{1/2}.$$

- within the loop over i, introduce a new loop which goes from $j = i+1$ to $n-1$ and calculate

$$L_{ji} = \frac{1}{L_{ii}}\left(A_{ij} - \sum_{k=0}^{i-1} L_{ik}l_{jk}\right).$$

For the Cholesky algorithm we have always that $L_{ii} > 0$ and the problem with exceedingly large matrix elements does not appear and hence there is no need for pivoting.

To decide whether a matrix is positive definite or not needs some careful analysis. To find criteria for positive definiteness, one needs two statements from matrix theory, see Golub and Van Loan [27] for examples. First, the leading principal submatrices of a positive definite matrix are positive definite and non-singular and secondly a matrix is positive definite if and only if it has an $\mathbf{LDL}^T$ factorization with positive diagonal elements only in the diagonal matrix $\mathbf{D}$. A positive definite matrix has to be symmetric and have only positive eigenvalues.

The easiest way therefore to test whether a matrix is positive definite or not is to solve the eigenvalue problem $\mathbf{Ax} = \lambda\mathbf{x}$ and check that all eigenvalues are positive.

4.4.3 Solution of linear systems of equations

With the LU decomposition it is rather simple to solve a system of linear equations

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= w_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= w_2 \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= w_3 \\
a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= w_4.
\end{aligned}$$

This can be written in matrix form as

$$\mathbf{Ax} = \mathbf{w}.$$

where $\mathbf{A}$ and $\mathbf{w}$ are known and we have to solve for $\mathbf{x}$. Using the LU dcomposition we write

$$\mathbf{Ax} \equiv \mathbf{LUx} = \mathbf{w}. \tag{4.14}$$

This equation can be calculated in two steps

$$\mathbf{L}\mathbf{y} = \mathbf{w}; \qquad \mathbf{U}\mathbf{x} = \mathbf{y}. \tag{4.15}$$

To show that this is correct we use to the LU decomposition to rewrite our system of linear equations as

$$\mathbf{L}\mathbf{U}\mathbf{x} = \mathbf{w},$$

and since the determinat of $\mathbf{L}$ is equal to 1 (by construction since the diagonals of $\mathbf{L}$ equal 1) we can use the inverse of $\mathbf{L}$ to obtain

$$\mathbf{U}\mathbf{x} = \mathbf{L}^{-\mathbf{1}}\mathbf{w} = \mathbf{y},$$

which yields the intermediate step

$$\mathbf{L}^{-\mathbf{1}}\mathbf{w} = \mathbf{y}$$

and multiplying with $\mathbf{B}$ on both sides we reobtain Eq. (4.15). As soon as we have $\mathbf{y}$ we can obtain $\mathbf{x}$ through $\mathbf{U}\mathbf{x} = \mathbf{y}$.

For our four-dimentional example this takes the form

$$\begin{aligned}
y_1 &= w_1 \\
l_{21}y_1 + y_2 &= w_2 \\
l_{31}y_1 + l_{32}y_2 + y_3 &= w_3 \\
l_{41}y_1 + l_{42}y_2 + l_{43}y_3 + y_4 &= w_4.
\end{aligned}$$

and

$$\begin{aligned}
u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + u_{14}x_4 &= y_1 \\
u_{22}x_2 + u_{23}x_3 + u_{24}x_4 &= y_2 \\
u_{33}x_3 + u_{34}x_4 &= y_3 \\
u_{44}x_4 &= y_4
\end{aligned}$$

This example shows the basis for the algorithm needed to solve the set of n linear equations. The algorithm goes as follows

- Set up the matrix **A** and the vector **w** with their correct dimensions. This determines the dimensionality of the unknown vector **x**.
- Then LU decompose the matrix **A** through a call to the function

C++:	ludcmp(double a, int n, int indx, double &d)
Fortran:	CALL lu_decompose(a, n, indx, d)

  This functions returns the LU decomposed matrix **A**, its determinant and the vector indx which keeps track of the number of interchanges of rows. If the determinant is zero, the solution is malconditioned.
- Thereafter you call the function

C++:	lubksb(double a, int n, int indx, double w)
Fortran:	CALL lu_linear_equation(a, n, indx, w)

  which uses the LU decomposed matrix **A** and the vector **w** and returns **x** in the same place as **w**. Upon exit the original content in **w** is destroyed. If you wish to keep this information, you should make a backup of it in your calling function.

4.4.4 Inverse of a matrix and the determinant

The basic definition of the determinant of **A** is

$$det\{\mathbf{A}\} = \sum_p (-)^p a_{1p_1} \cdot a_{2p_2} \cdots a_{np_n},$$

where the sum runs over all permutations p of the indices $1, 2, \ldots, n$, altogether $n!$ terms. Also to calculate the inverse of **A** is a formidable task. Here we have to calculate *the complementary cofactor* a^{ij} of each element a_ij which is the $(n-1)$determinant obtained by striking out the row i and column j in which the element a_{ij} appears. The inverse of **A** is then constructed as the transpose a matrix with the elements $(-)^{i+j}a^{ij}$. This involves a calculation of n^2 determinants using the formula above. A simplified method is highly needed.

With the LU decomposed matrix **A** in Eq. (4.11) it is rather easy to find the determinant

$$det\{\mathbf{A}\} = det\{\mathbf{L}\} \times det\{\mathbf{U}\} = det\{\mathbf{U}\},$$

since the diagonal elements of **L** equal 1. Thus the determinant can be written

$$det\{\mathbf{A}\} = \prod_{k=1}^{N} u_{kk}.$$

The inverse is slightly more difficult to obtain from the LU decomposition. It is formally defined as

$$\mathbf{A}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1}.$$

We use this form since the computation of the inverse goes through the inverse of the matrices **L** and **U**. The reason is that the inverse of a lower (upper) triangular matrix is also a lower (upper) triangular matrix. If we call **D** for the inverse of **L**, we can determine the matrix elements of **D** through the equation

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ d_{21} & 1 & 0 & 0 \\ d_{31} & d_{32} & 1 & 0 \\ d_{41} & d_{42} & d_{43} & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

which gives the following general algorithm

$$d_{ij} = -l_{ij} - \sum_{k=j+1}^{i-1} l_{ik} d_{kj}, \tag{4.16}$$

which is valid for $i > j$. The diagonal is 1 and the upper matrix elements are zero. We solve this equation column by column (increasing order of j). In a similar way we can define an equation which gives us the inverse of the matrix **U**, labelled **E** in the equation below. This contains only non-zero matrix elements in the upper part of the matrix (plus the diagonal ones)

$$\begin{pmatrix} e_{11} & e_{12} & e_{13} & e_{14} \\ 0 & e_{22} & e_{23} & e_{24} \\ 0 & 0 & e_{33} & e_{34} \\ 0 & 0 & 0 & e_{44} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

with the following general equation

$$e_{ij} = -\frac{1}{u_{jj}} \sum_{k=1}^{j-1} e_{ik} u_{kj}. \tag{4.17}$$

for $i \leq j$.

A calculation of the inverse of a matrix could then be implemented in the following way:

- Set up the matrix to be inverted.
- Call the LU decomposition function.
- Check whether the determinant is zero or not.
- Then solve column by column Eqs. (4.16, 4.17).

The following codes compute the inverse of a matrix using either C++ or Fortran as programming languages. They are both included in the library packages, but we include them explicitely here as well as two distinct programs. We list first the C++ code

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter04/cpp/program1.cpp

```
/*   The  function
**                    inverse()
```

```
** perform a mtx inversion of the input matrix a[][] with
** dimension n.
*/
void inverse(double **a, int n)
{
  int          i,j, *indx;
  double       d, *col, **y;

  // allocate space in memory
  indx = new int[n];
  col  = new double[n];
  y    = (double **) matrix(n, n, sizeof(double));
  // first we need to LU decompose the matrix
  ludcmp(a, n, indx, &d);
  // find inverse of a[][] by columns
  for(j = 0; j < n; j++) {
    // initialize right-side of linear equations
    for(i = 0; i < n; i++) col[i] = 0.0;
    col[j] = 1.0;
    lubksb(a, n, indx, col);
    // save result in y[][]
    for(i = 0; i < n; i++) y[i][j] = col[i];
  }
  // return the inverse matrix in a[][]

  for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) a[i][j] = y[i][j];
  }
  free_matrix((void **) y);      // release local memory
  delete [] col;
  delete []indx;

}  // End: function inverse()
```

We first need to LU decompose the matrix. Thereafter we solve Eqs. (4.16) and (4.17) by using the back substitution method calling the function **lubksb** and obtain finally the inverse matrix.

An example of a C++ function which calls this function is also given in the program and reads

`http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter04/cpp/program1.cpp`

```
//  Simple matrix inversion example
#include <iostream>
#include <new>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
#include    "lib.h"

using namespace std;

/* function declarations */

void inverse(double **, int);
```

```
/*
** This program sets up a simple 3x3 symmetric matrix
** and finds its determinant and inverse
*/

int main()
{
  int          i, j, k, result, n = 3;
  double       **matr, sum,
    a[3][3]    = { {1.0, 3.0, 4.0},
                   {3.0, 4.0, 6.0},
                   {4.0, 6.0, 8.0}};
  // memory for  inverse matrix
  matr = (double **) matrix(n, n, sizeof(double));
  // various print statements in the original code are omitted

  inverse(matr, n);     // calculate and return inverse matrix
  ....
  return 0;
} // End: function main()
```

In order to use the program library you need to include the **lib.h** file using the **#include** `"lib.h"` statement. This function utilizes the library function **matrix** and **free_matrix** to allocate and free memory during execution. The matrix $a[3][3]$ is set at compilation time. The corresponding Fortran program for the inverse of a matrix reads

`http://www.fys.uio.no/compphys/cp/programs/FYS3150/f90library/f90lib.f90`

```
!
!           Routines to do mtx inversion, from Numerical
!           Recipes, Teukolsky et al. Routines included
!           below are MATINV, LUDCMP and LUBKSB. See chap 2
!           of Numerical Recipes for further details
!
SUBROUTINE matinv(a,n, indx, d)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  INTEGER :: i, j
  REAL(DP), DIMENSION(n,n), INTENT(INOUT)  :: a
  REAL(DP), ALLOCATABLE :: y(:,:)
  REAL(DP) :: d
  INTEGER, , INTENT(INOUT) :: indx(n)

  ALLOCATE (y( n, n))
  y=0.
  !      setup identity matrix
  DO i=1,n
     y(i,i)=1.
  ENDDO
  !     LU decompose the matrix just once
  CALL  lu_decompose(a,n,indx,d)

  !      Find inverse by columns
  DO j=1,n
```

```
      CALL lu_linear_equation(a,n,indx,y(:,j))
   ENDDO
!    The original matrix a was destroyed, now we equate it with the
      inverse y
   a=y
   DEALLOCATE ( y )

END SUBROUTINE matinv
```

The Fortran program **matinv** receives as input the same variables as the C++ program and calls the function for LU decomposition **lu_decompose** and the function to solve sets of linear equations **lu_linear_equation**. The program listed under programs/chapter4/program1.f90 performs the same action as the C++ listed above. In order to compile and link these programs it is convenient to use a so-called **makefile**. Examples of these are found under the same catalogue as the above programs.

Inverse of the Vandermonde matrix

In chapter 6 we discuss how to interpolate a function f which is known only at $n+1$ points $x_0, x_1, x_2, \dots, x_n$ with corresponding values $f(x_0), f(x_1), f(x_2), \dots, f(x_n)$. The latter is often a typical outcome of a large scale computation or from an experiment. In most cases in the sciences we do not have a closed form expressions for a function f. The function is only known at specific points.

We seek a functional form for a function f which passes through the above pairs of values

$$(x_0, f(x_0)), (x_1, f(x_1)), (x_2, f(x_2)), \dots, (x_n, f(x_n)).$$

This is normally achieved by expanding the function $f(x)$ in terms of well-known polynomials $\phi_i(x)$, such as Legendre, Chebyshev, Laguerre etc. The function is then approximated by a polynomial of degree n $p_n(x)$

$$f(x) \approx p_n(x) = \sum_{i=0}^{n} a_i \phi_i(x),$$

where a_i are unknown coefficients and $\phi_i(x)$ are a priori well-known functions. The simplest possible case is to assume that $\phi_i(x) = x^i$, resulting in an approximation

$$f(x) \approx a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

Our function is known at the points $n+1$ points $x_0, x_1, x_2, \dots, x_n$, leading to $n+1$ equations of the type

$$f(x_i) \approx a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n.$$

We can then obtain the unknown coefficients by rewriting our problem as

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & \dots & x_2^n \\ 1 & x_3 & x_3^2 & \dots & \dots & x_3^n \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_n & x_n^2 & \dots & \dots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \dots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ f(x_3) \\ \dots \\ f(x_n) \end{pmatrix},$$

an expression which can be rewritten in a more compact form as

$$\mathbf{X}\mathbf{a} = \mathbf{f},$$

with

$$
\mathbf{X} = \begin{pmatrix}
1 & x_0 & x_0^2 & \dots & \dots & x_0^n \\
1 & x_1 & x_1^2 & \dots & \dots & x_1^n \\
1 & x_2 & x_2^2 & \dots & \dots & x_2^n \\
1 & x_3 & x_3^2 & \dots & \dots & x_3^n \\
\dots & \dots & \dots & \dots & \dots & \dots \\
1 & x_n & x_n^2 & \dots & \dots & x_n^n
\end{pmatrix}.
$$

. This matrix is called a Vandermonde matrix and is by definition non-singular since all points x_i are different. The inverse exists and we can obtain the unknown coefficients by inverting $\mathbf{X}$, resulting in

$$
\mathbf{a} = \mathbf{X}^{-1}\mathbf{f}.
$$

Although this algorithm for obtaining an interpolating polynomial which approximates our data set looks very simple, it is an inefficient algorithm since the computation of the inverse requires $O(n^3)$ flops. The methods we will discuss in chapter 6 are much more effective from a numerical point of view. There is also another subtle point. Although we have a data set with $n+1$ points, this does not necessarily mean that our function $f(x)$ is well represented by a polynomial of degree n. On the contrary, our function $f(x)$ may be a parabola (second-order in n), meaning that we have a large excess of data points. In such cases a least-square fit or a spline interpolation may be better approaches to represent the function. These techniques are discussed in chapter 6.

4.4.5 Tridiagonal systems of linear equations

We start with the linear set of equations from Eq. (4.8), viz

$$
\mathbf{A}\mathbf{u} = \mathbf{f},
$$

where $\mathbf{A}$ is a tridiagonal matrix which we rewrite as

$$
\mathbf{A} = \begin{pmatrix}
b_1 & c_1 & 0 & \dots & \dots & \dots \\
a_2 & b_2 & c_2 & \dots & \dots & \dots \\
 & a_3 & b_3 & c_3 & \dots & \dots \\
 & \dots & \dots & \dots & \dots & \dots \\
 & & & a_{n-2} & b_{n-1} & c_{n-1} \\
 & & & & a_n & b_n
\end{pmatrix}
$$

where a, b, c are one-dimensional arrays of length $1:n$. In the example of Eq. (4.8) the arrays a and c are equal, namely $a_i = c_i = -1/h^2$. We can rewrite Eq. (4.8) as

$$
\mathbf{A} = \begin{pmatrix}
b_1 & c_1 & 0 & \dots & \dots & \dots \\
a_2 & b_2 & c_2 & \dots & \dots & \dots \\
 & a_3 & b_3 & c_3 & \dots & \dots \\
 & \dots & \dots & \dots & \dots & \dots \\
 & & & a_{n-2} & b_{n-1} & c_{n-1} \\
 & & & & a_n & b_n
\end{pmatrix}
\begin{pmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_n \end{pmatrix}
=
\begin{pmatrix} f_1 \\ f_2 \\ \dots \\ \dots \\ \dots \\ f_n \end{pmatrix}.
$$

A tridiagonal matrix is a special form of banded matrix where all the elements are zero except for those on and immediately above and below the leading diagonal. The above tridiagonal system can be written as

$$
a_i u_{i-1} + b_i u_i + c_i u_{i+1} = f_i,
$$

for $i = 1, 2, \ldots, n$. We see that u_{-1} and u_{n+1} are not required and we can set $a_1 = c_n = 0$. In many applications the matrix is symmetric and we have $a_i = c_i$. The algorithm for solving this set of equations is rather simple and requires two steps only, a forward substitution and a backward substitution. These steps are also common to the algorithms based on Gaussian elimination that we will discussed previously. However, due to its simplicity, the number of floating point operations is in this case proportional with $O(n)$ while Gaussian elimination requires $2n^3/3 + O(n^2)$ floating point operations. In case your system of equations leads to a tridiagonal matrix, it is clearly an overkill to employ Gaussian elimination or the standard LU decomposition. You will encounter several applications involving tridiagonal matrices in our discussion of partial differential equations in chapter 15.

Our algorithm starts with forward substitution with a loop over of the elements i and can be expressed via the following piece of code taken from the Numerical Recipe text of Teukolsky *et al* [36]

```
btemp = b[1];
u[1] = f[1]/btemp;
for(i=2 ; i <= n ; i++) {
   temp[i] = c[i-1]/btemp;
   btemp = b[i]-a[i]*temp[i];
   u[i] = (f[i] - a[i]*u[i-1])/btemp;
}
```

Note that you should avoid cases with $b_1 = 0$. If that is the case, you should rewrite the equations as a set of order $n-1$ with u_2 eliminated. Finally we perform the backsubstitution leading to the following code

```
for(i=n-1 ; i >= 1 ; i--) {
   u[i] -= temp[i+1]*u[i+1];
}
```

Note that our sums start with $i = 1$ and that one should avoid cases with $b_1 = 0$. If that is the case, you should rewrite the equations as a set of order $n-1$ with u_2 eliminated. However, a tridiagonal matrix problem is not a guarantee that we can find a solution. The matrix $\mathbf{A}$ which rephrases a second derivative in a discretized form

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix},$$

fulfills the condition of a weak dominance of the diagonal, with $|b_1| > |c_1|$, $|b_n| > |a_n|$ and $|b_k| \geq |a_k| + |c_k|$ for $k = 2, 3, \ldots, n-1$. This is a relevant but not sufficient condition to guarantee that the matrix $\mathbf{A}$ yields a solution to a linear equation problem. The matrix needs also to be irreducible. A tridiagonal irreducible matrix means that all the elements a_i and c_i are non-zero. If these two conditions are present, then $\mathbf{A}$ is nonsingular and has a unique LU decomposition.

We can obviously extend our boundary value problem to include a first derivative as well

$$-\frac{d^2u(x)}{dx^2} + g(x)\frac{du(x)}{dx} + h(x)u(x) = f(x),$$

with $x \in [a, b]$ and with boundary conditions $u(a) = u(b) = 0$. We assume athat f, g and h are continuous functions in the domain $x \in [a, b]$ and that $h(x) \geq 0$. Then the differential equation has a unique solution. We subdivide our interval $x \in [a, b]$ into n subintervals by setting $x_i = ih$, with

$i = 0, 1, \ldots, n+1$. The step size is then given by $h = (b-a)/(n+1)$ with $n \in \mathbb{N}$. For the internal grid points $i = 1, 2, \ldots n$ we replace the differential operators with

$$u_i^{''} \approx \frac{u_{i+1} - 2u_i + u_{i-i}}{h^2}.$$

for the second derivative while the first derivative is given by

$$u_i^{'} \approx \frac{u_{i+1} - u_{i-i}}{2h}.$$

We rewrite our original differential equation in terms of a discretized equation as

$$-\frac{u_{i+1} - 2u_i + u_{i-i}}{h^2} + g_i \frac{u_{i+1} - u_{i-i}}{2h} + h_i u_i = f_i,$$

with $i = 1, 2, \ldots, n$. We need to add to this system the two boundary conditions $u(a) = u_0$ and $u(b) = u_{n+1}$. This equation can again be rewritten as a tridiagonal matrix problem. We leave it as an exercise to the reader to find the matrix elements, find the conditions for having weakly dominant diagonal elements and that the matrix is irreducible.

4.5 Singular value decomposition

Finalized end fall 2008.

4.6 Exercises and projects

Exercise 2.1: Writing your own Gaussian elimination code

(a) Consider the linear system of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= w_3. \end{aligned}$$

This can be written in matrix form as

$$\mathbf{Ax} = \mathbf{w}.$$

We specialize here to the following case

$$\begin{aligned} -x_1 + x_2 - 4x_3 &= 0 \\ 2x_1 + 2x_2 &= 1 \\ 3x_1 + 3x_2 + 2x_3 &= \tfrac{1}{2}. \end{aligned}$$

Obtain the solution (by hand) of this system of equations by doing Gaussian elimination.

(b) Write therafter a program which implements Gaussian elimination (with pivoting) and solve the above system of linear equations. How many floating point operations are involved in the solution via Gaussian elimination without pivoting? Can you estimate the number of floating point operations with pivoting?

Exercise 4.2: Cholesky factorization

If the matrix A is real, symmetric and positive definite, then it has a unique factorization (called Cholesky factorization)

$$A = LU = LL^T$$

where L^T is the upper matrix, implying that

$$L^T_{ij} = L_{ji}.$$

The algorithm for the Cholesky decomposition is a special case of the general LU-decomposition algorithm. The algorithm of this decomposition is as follows

– Calculate the diagonal element L_{ii} by setting up a loop for $i = 0$ to $i = n-1$ (C++ indexing of matrices and vectors)

$$L_{ii} = \left(A_{ii} - \sum_{k=0}^{i-1} L_{ik}^2 \right)^{1/2}. \tag{4.18}$$

– within the loop over i, introduce a new loop which goes from $j = i+1$ to $n-1$ and calculate

$$L_{ji} = \frac{1}{L_{ii}} \left(A_{ij} - \sum_{k=0}^{i-1} L_{ik} l_{jk} \right). \tag{4.19}$$

For the Cholesky algorithm we have always that $L_{ii} > 0$ and the problem with exceedingly large matrix elements does not appear and hence there is no need for pivoting. Write a function which performs the Cholesky decomposition. Test your program against the standard LU decomposition by using the matrix

$$\mathbf{A} = \begin{pmatrix} 6 & 3 & 2 \\ 3 & 2 & 1 \\ 2 & 1 & 1 \end{pmatrix} \tag{4.20}$$

Finally, use the Cholesky method to solve

$$\begin{aligned} 0.05x_1 + 0.07x_2 + 0.06x_3 + 0.05x_4 &= 0.23 \\ 0.07x_1 + 0.10x_2 + 0.08x_3 + 0.07x_4 &= 0.32 \\ 0.06x_1 + 0.08x_2 + 0.10x_3 + 0.09x_4 &= 0.33 \\ 0.05x_1 + 0.07x_2 + 0.09x_3 + 0.10x_4 &= 0.31 \end{aligned}$$

You can also use the LU codes for linear equations to check the results.

Project 4.1: The one-dimensional Poisson equation

(a) We are going to solve the one-dimensional Poisson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations.

The three-dimensional Poisson equation is a partial differential equation,

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2} = -\frac{\rho(x,y,z)}{\epsilon_0},$$

whose solution we will discuss in chapter 15. The function $\rho(x, y, z)$ is the charge density and ϕ is the electrostatic potential. In this project we consider the one-dimensional case since there are a few situations, possessing a high degree of symmetry, where it is possible to find analytic solutions. Let us discuss some of these solutions.

Suppose, first of all, that there is no variation of the various quantities in the y- and z-directions. In this case, Poisson's equation reduces to an ordinary differential equation in x, the solution of which is relatively straightforward. Consider for example a vacuum diode, in which electrons are emitted from a hot cathode and accelerated towards an anode. The anode is held at a large positive potential V_0 with respect to the cathode. We can think of this as an essentially one-dimensional problem. Suppose that the cathode is at $x = 0$ and the anode at $x = d$. Poisson's equation takes the form

$$\frac{d^2\phi}{dx^2} = -\frac{\rho(x)}{\epsilon_0},$$

where $\phi(x)$ satisfies the boundary conditions $\phi(0) = 0$ and $\phi(d) = V_0$. By energy conservation, an electron emitted from rest at the cathode has an x-velocity $v(x)$ which satisfies

$$\frac{1}{2}m_e v^2(x) - e\phi(x) = 0.$$

Furthermore, we assume that the current I is independent of x between the anode and cathode, otherwise, charge will build up at some points. From electromagnetism one can then show that the current I is given by $I = -\rho(x)v(x)A$, where A is the cross-sectional area of the diode. The previous equations can be combined to give

$$\frac{d^2\phi}{dx^2} = \frac{I}{\epsilon_0 A}\left(\frac{m_e}{2e}\right)^{1/2}\phi^{-1/2}.$$

The solution of the above equation which satisfies the boundary conditions is

$$\phi = V_0\left(\frac{x}{d}\right)^{4/3},$$

with

$$I = \frac{4}{9}\frac{\epsilon_0 A}{d^2}\left(\frac{2e}{m_e}\right)^{1/2} V_0^{3/2}.$$

This relationship between the current and the voltage in a vacuum diode is called the Child-Langmuir law.

Another physics example in one dimension is the famous Thomas-Fermi model, widely used as a mean-field model in simulations of quantum mechanical systems [37, 38], see Lieb for a newer and updated discussion [39]. Thomas and Fermi assumed the existence of an energy functional, and derived an expression for the kinetic energy based on the density of electrons, $\rho(r)$ in an infinite potential well. For a large atom or molecule with a large number of electrons. Schrödinger's equation, which would give the exact density and energy, cannot be easily handled for large numbers of interacting particles. Since the Poisson equation connects the electrostatic potential with the charge density, one can derive the following equation for potential V

$$\frac{d^2V}{dx^2} = \frac{V^{3/2}}{\sqrt{x}},$$

with $V(0) = 1$.

In our case we will rewrite Poisson's equation in terms of dimensionless variables. We can then rewrite the equation as

$$-u''(x) = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0.$$

and we define the discretized approximation to u as v_i with grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$. The step length or spacing is defined as $h = 1/(n+1)$. We have then the boundary conditions $v_0 = v_{n+1} = 0$. We approximate the second derivative of u with

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, \dots, n,$$

where $f_i = f(x_i)$. Show that you can rewrite this equation as a linear set of equations of the form

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}},$$

where $\mathbf{A}$ is an $n \times n$ tridiagonal matrix which we rewrite as

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{pmatrix}$$

and $\tilde{b}_i = h^2 f_i$.

In our case we will assume that $f(x) = (3x + x^2)e^x$, and keep the same interval and boundary conditions. Then the above differential equation has an analytic solution given by $u(x) = x(1-x)e^x$ (convince yourself that this is correct by inserting the solution in the Poisson equation). We will compare our numerical solution with this analytic result in the next exercise.

(b) We can rewrite our matrix $\mathbf{A}$ in terms of one-dimensional vectors a, b, c of length $1:n$. Our linear equation reads

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{pmatrix}.$$

A tridiagonal matrix is a special form of banded matrix where all the elements are zero except for those on and immediately above and below the leading diagonal. The above tridiagonal system can be written as

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i,$$

for $i = 1, 2, \dots, n$. The algorithm for solving this set of equations is rather simple and requires two steps only, a decomposition and forward substitution and finally a backward substitution.

Your first task is to set up the algorithm for solving this set of linear equations. Find also the number of operations needed to solve the above equations. Show that they behave like $O(n)$ with n the dimensionality of the problem. Compare this with standard Gaussian elimination.

Then you should code the above algorithm and solve the problem for matrices of the size 10×10, 100×100 and 1000×1000. That means that you choose $n = 10$, $n = 100$ and $n = 1000$ grid points.

Compare your results (make plots) with the analytic results for the different number of grid points in the interval $x \in (0, 1)$. The different number of grid points corresponds to different step lengths h.

Compute also the maximal relative error in the data set $i = 1, \ldots, n$,by setting up

$$\epsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right),$$

as function of $log_{10}(h)$ for the function values u_i and v_i. For each step length extract the max value of the relative error. Try to increase n to $n = 10000$ and $n = 10^5$. Comment your results.

(c) Compare your results with those from the LU decomposition codes for the matrix of size 1000×1000. Use for example the unix function *time* when you run your codes and compare the time usage between LU decomposition and your tridiagonal solver. Can you run the standard LU decomposition for a matrix of the size $10^5 \times 10^5$? Comment your results.

Solution to exercise b)

The program listed below encodes a possible solution to part b) of the above project. Note that we have employed Blitz++ as library and that the range of the various vectors are now shifted from their default ranges $(0 : n-1)$ to $(1 : n)$ and that we access vector elements as $a(i)$ instead of the standard C++ declaration $a[i]$.

The program reads from screen the name of the ouput file and the dimension of the problem, which in our case corresponds to the number of mesh points as well, in addition to the two endpoints. The function $f(x) = (3x + x^2)\exp{(x)}$ is included explicitely in the code. An obvious change is to define a separate function, allowing thereby for a generalization to other function $f(x)$.

```
/*
    Program to solve the one-dimensional Poisson equation
    -u''(x) = f(x)  rewritten as a set of linear equations
    A u = f   where A is an n x n matrix, and u and f are 1 x n vectors
    In this problem f(x) = (3x+x*x)exp(x)  with solution u(x) = x(1-x)exp(x)
    The program reads  from screen the name of the output file.
    Blitz++ is used here, with arrays starting from 1 to n
*/
#include <iomanip>
#include <fstream>
#include <blitz/array.h>
#include <iostream>
using namespace std;
using namespace blitz;

ofstream ofile;
//  Main program only, no other functions
```

```
int main(int argc, char* argv[])
{
  char *outfilename;
  int i, j, n;
  double h, btemp;
  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  cout << "Read in number of mesh points" << endl;
  cin >> n;
  h =  1.0/( (double) n+1);
  //  Use Blitz to allocate arrays
  //  Use range to change default arrays from 0:n-1 to 1:n
  Range r(1,n);
  Array<double,1> a(r), b(r), c(r), y(r), f(r), temp(r);
  //  set up the matrix defined by three arrays, diagonal, upper and lower
     diagonal band
  b = 2.0;   a = -1.0   ; c = -1.0;
  // Then define the value of the right hand side f (multiplied by h*h)
  for(i=1; i <= n; i++){
    // Explicit expression for f, could code as separate function
    f(i) = h*h*(i*h*3.0+(i*h)*(i*h))*exp(i*h);
  }
  // solve the tridiagonal system, first forward substitution
  btemp = b(1);
  for(i = 2; i <= n; i++) {
    temp(i) = c(i-1) / btemp;
    btemp = b(i) - a(i) * temp(i);
    y(i) = (f(i) - a(i) * y(i-1)) / btemp;
  }
  // then backward substitution, the solution is in y()
  for(i = n-1; i >= 1; i--) {
    y(i) -= temp(i+1) * y(i+1);
  }
  // write results to the output file
  for(i = 1; i <= n; i++){
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << setw(15) << setprecision(8) << i*h;
    ofile << setw(15) << setprecision(8) << y(i);
    ofile << setw(15) << setprecision(8) << i*h*(1.0-i*h)*exp(i*h) <<endl;
  }
  ofile.close();
}
```

The program writes also the exact solution to file. In Fig. 4.4 we show the results obtained with $n = 10$. Even with so few points, the numerical solution is very close to the analytic answer. With $n = 100$ it is

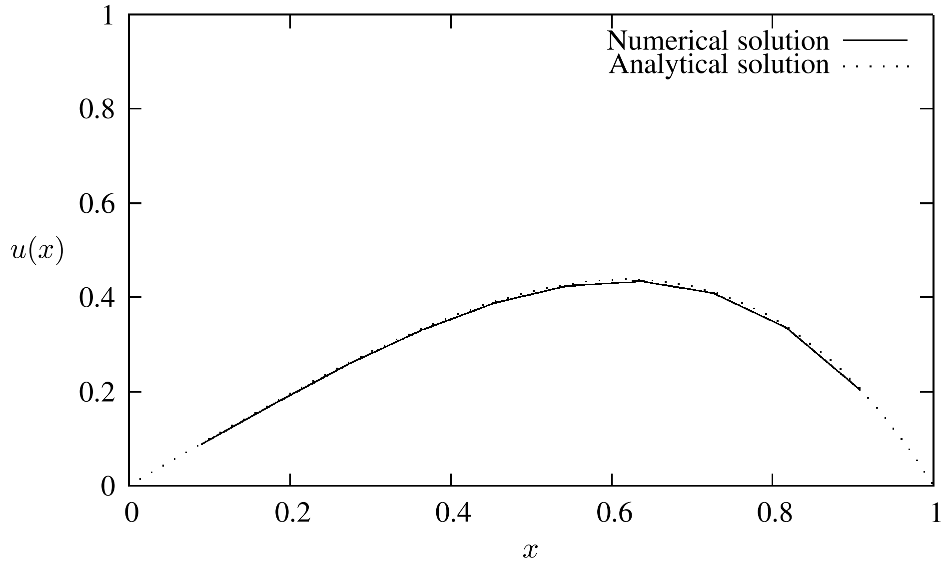


Figure 4.4: Numerical solution obtained with $n = 10$ compared with the analytical solution.

almost impossible to distinguish the numerical solution from the analytical one, as shown in Fig. 4.5. It is therefore instructive to study the relative error, which we display in Table 4.4 as function of the step length $h = 1/(n + 1)$.

Table 4.4: log_{10} values for the relative error and the step length h computed at $x = 0.5$.

n	$log_{10}(h)$	$\epsilon_i = log_{10}\left(\lvert(v_i - u_i)/u_i\rvert\right)$
10	-1.04	-2.29
100	-2.00	-4.19
1000	-3.00	-6.18
10^4	-4.00	-8.18
10^5	-5.00	-9.19
10^6	-6.00	-6.08

The mathematical truncation we made when computing the second derivative goes like $O(h^2)$. Our results for n from $n = 10$ to somewhere between $n = 10^4$ and $n = 10^5$ result in a slope which is almost exactly equal 2,in good agreement with the mathematical truncation made. Beyond $n = 10^5$ the relative error becomes bigger, telling us that there is no point in increasing n. For most practical application a relative error between 10^{-6} and 10^{-8} is more than sufficient, meaning that $n = 10^4$ may be an acceptable number of mesh points. Beyond $n = 10^5$, numerical round off errors take over, as discussed in the previous chapter as well.

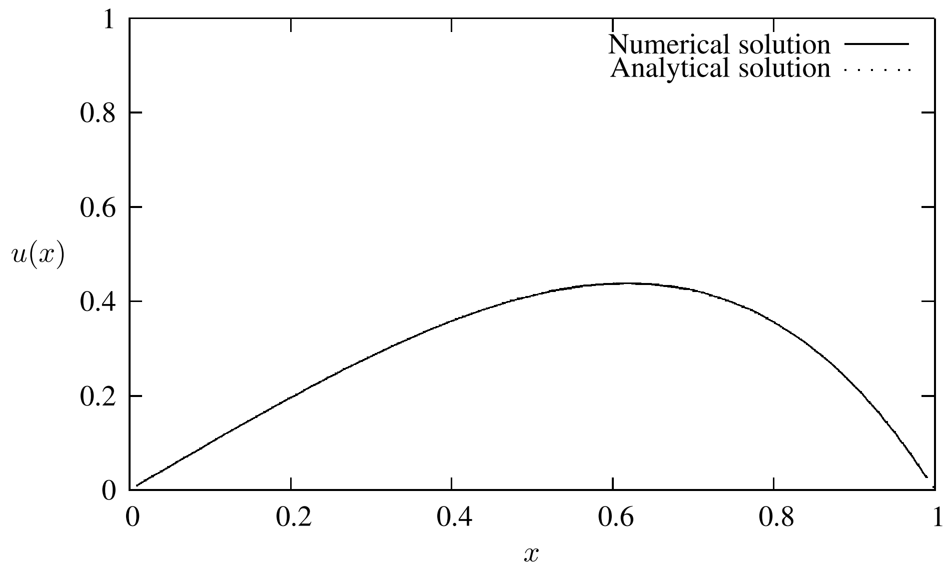


Figure 4.5: Numerical solution obtained with $n = 10$ compared with the analytical solution.

Chapter 5

Non-linear equations and roots of polynomials

5.1 Introduction

In physics we often encounter the problem of determining the root of a function $f(x)$. Especially, we may need to solve non-linear equations of one variable. Such equations are usually divided into two classes, algebraic equations involving roots of polynomials and transcendental equations. When there is only one independent variable, the problem is one-dimensional, namely to find the root or roots of a function. Except in linear problems, root finding invariably proceeds by iteration, and this is equally true in one or in many dimensions. This means that we cannot solve exactly the equations at hand. Rather, we start with some approximate trial solution. The chosen algorithm will in turn improve the solution until some predetermined convergence criterion is satisfied. The algoritms we discuss below attempt to implement this strategy. We will deal mainly with one-dimensional problems.

You may have encountered examples of so-called transcendental equations when solving the Schrödinger equation (SE) for a particle in a box potential. The one-dimensional SE for a particle with mass m is

$$-\frac{\hbar^2}{2m}\frac{d^2u}{drx^2}+V(x)u(x)=Eu(x), \tag{5.1}$$

and our potential is defined as

$$V(r)=\begin{cases} -V_0 & 0\le x<a \\ 0 & x>a \end{cases} \tag{5.2}$$

Bound states correspond to negative energy E and scattering states are given by positive energies. The SE takes the form (without specifying the sign of E)

$$\frac{d^2u(x)}{dx^2}+\frac{2m}{\hbar^2}\left(V_0+E\right)u(x)=0 \qquad x<a, \tag{5.3}$$

and

$$\frac{d^2u(x)}{dx^2}+\frac{2m}{\hbar^2}Eu(x)=0 \qquad x>a. \tag{5.4}$$

If we specialize to bound states $E<0$ and implement the boundary conditions on the wave function we obtain

$$u(r)=Asin(\sqrt{2m(V_0-|E|)}r/\hbar) \qquad r<a, \tag{5.5}$$

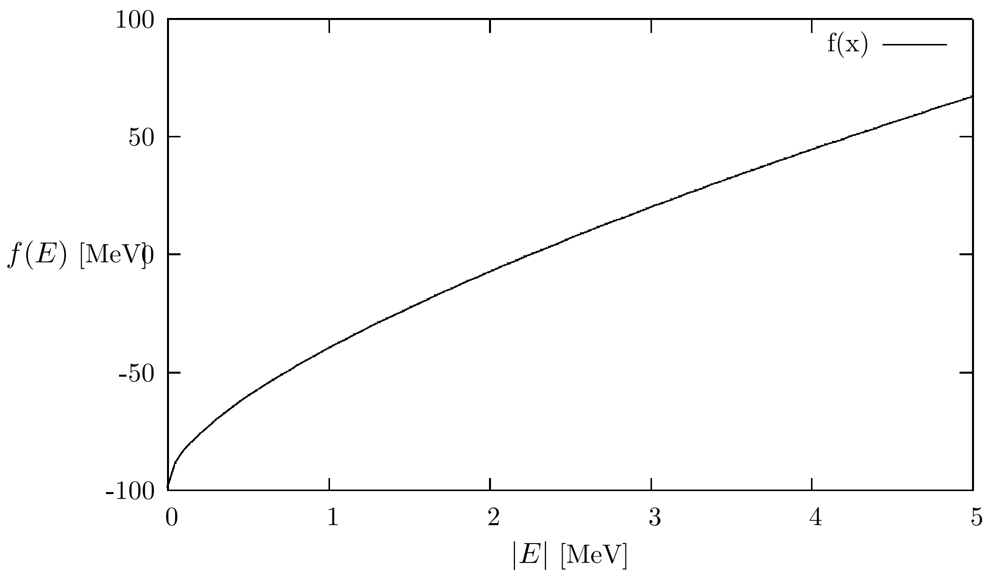


Figure 5.1: Plot of $f(E)$ Eq. (5.8) as function of energy |E| in MeV. $f(E)$ has dimension MeV. Note well that the energy E is for bound states.

and

$$u(r) = B\exp\left(-\sqrt{2m|E|}r/\hbar\right) \qquad r > a, \tag{5.6}$$

where A and B are constants. Using the continuity requirement on the wave function at $r = a$ one obtains the transcendental equation

$$\sqrt{2m(V_0 - |E|)}cot(\sqrt{2ma^2(V_0 - |E|)}/\hbar) = -\sqrt{2m|E|}. \tag{5.7}$$

This equation is an example of the kind of equations which could be solved by some of the methods discussed below. The algorithms we discuss are the bisection method, the secant, false position and Brent's methods and Newton-Raphson's method.

In order to find the solution for Eq. (5.7), a simple procedure is to define a function

$$f(E) = \sqrt{2m(V_0 - |E|)}cot(\sqrt{2ma^2(V_0 - |E|)}/\hbar) + \sqrt{2m|E|}. \tag{5.8}$$

and with chosen or given values for a and V_0 make a plot of this function and find the approximate region along the $E - axis$ where $f(E) = 0$. We show this in Fig. 5.1 for $V_0 = 20$ MeV, $a = 2$ fm and $m = 938$ MeV. Fig. 5.1 tells us that the solution is close to $|E| \approx 2.2$ (the binding energy of the deuteron). The methods we discuss below are then meant to give us a numerical solution for E where $f(E) = 0$ is satisfied and with E determined by a given numerical precision.

5.2 Iteration methods

To solve an equation of the type $f(x) = 0$ means mathematically to find all numbers s[1] so that $f(s) = 0$. In all actual calculations we are always limited by a given precision when doing numerics. Through an

[1]In the following discussion, the variable s is reserved for the value of x where we have a solution.

iterative search of the solution, the hope is that we can approach, within a given tolerance ϵ, a value x_0 which is a solution to $f(s) = 0$ if

$$|x_0 - s| < \epsilon, \tag{5.9}$$

and $f(s) = 0$. We could use other criteria as well like

$$\left|\frac{x_0 - s}{s}\right| < \epsilon, \tag{5.10}$$

and $|f(x_0)| < \epsilon$ or a combination of these. However, it is not given that the iterative process will converge and we would like to have some conditions on f which ensures a solution. This condition is provided by the so-called Lipschitz criterion. If the function f, defined on the interval $[a, b]$ satisfies for all x_1 and x_2 in the chosen interval the following condition

$$|f(x_1) - f(x_2)| \leq k\, |x_1 - x_2|\,, \tag{5.11}$$

with k a constant, then f is continuous in the interval $[a, b]$. If f is continuous in the interval $[a, b]$, then the secant condition gives

$$f(x_1) - f(x_2) = f'(\xi)(x_1 - x_2), \tag{5.12}$$

with x_1, x_2 within $[a, b]$ and ξ within $[x_1, x_2]$. We have then

$$|f(x_1) - f(x_2)| \leq |f'(\xi)|\, |x_1 - x_2|\,. \tag{5.13}$$

The derivative can be used as the constant k. We can now formulate the sufficient conditions for the convergence of the iterative search for solutions to $f(s) = 0$.

1. We assume that f is defined in the interval $[a, b]$.

2. f satisfies the Lipschitz condition with $k < 1$.

With these conditions, the equation $f(x) = 0$ has only one solution in the interval $[a, b]$ and it coverges after n iterations towards the solution s irrespective of choice for x_0 in the interval $[a, b]$. If we let x_n be the value of x after n iterations, we have the condition

$$|s - x_n| \leq \frac{k}{1-k}\, |x_1 - x_2|\,. \tag{5.14}$$

The proof can be found in the text of Bulirsch and Stoer. Since it is difficult numerically to find exactly the point where $f(s) = 0$, in the actual numerical solution one implements three tests of the type

1. $$|x_n - s| < \epsilon, \tag{5.15}$$

   and

2. $$|f(s)| < \delta, \tag{5.16}$$

3. and a maximum number of iterations N_{maxiter} in actual calculations.

5.3 Bisection method

This is an extremely simple method to code. The philosophy can best be explained by choosing a region in e.g., Fig. 5.1 which is close to where $f(E) = 0$. In our case $|E| \approx 2.2$. Choose a region $[a, b]$ so that $a = 1.5$ and $b = 3$. This should encompass the point where $f = 0$. Define then the point

$$c = \frac{a+b}{2}, \tag{5.17}$$

and calculate $f(c)$. If $f(a)f(c) < 0$, the solution lies in the region $[a, c] = [a, (a+b)/2]$. Change then $b \leftarrow c$ and calculate a new value for c. If $f(a)f(c) > 0$, the new interval is in $[c, b] = [(a+b)/2, b]$. Now you need to change $a \leftarrow c$ and evaluate then a new value for c. We can continue to halve the interval till we have reached a value for c which fulfils $f(c) = 0$ to a given numerical precision. The algorithm can be simply expressed in the following program

```
      ......
      fa = f(a);
      fb = f(b);
//    check if your interval is correct, if not return to main
      if  ( fa*fb > 0) {
         cout << ``\n Error, root not in interval'' << endl;
         return;
      }
      for (j=1; j <= iter_max; j++) {
         c=(a+b)/2;
         fc=f(c)
//   if this test is satisfied, we have the root c
         if  ( (abs(a-b) < epsilon ) || fc < delta ); return to main
         if ( fa*fc < 0){
            b=c  ;  fb=fc;
         }
         else{
            a=c ; fa=fc;
         }
      }
      ......
```

Note that one needs to define the values of δ, ϵ and `iter_max` when calling this function.

The bisection method is an almost foolproof method, although it may converge slowly towards the solution due to the fact that it halves the intervals. After n divisions by 2 we have a possible solution in the interval with length

$$\frac{1}{2^n}|b-a|, \tag{5.18}$$

and if we set $x_0 = (a+b)/2$ and let x_n be the midpoints in the intervals we obtain after n iterations that Eq. (5.14) results in

$$|s - x_n| \leq= \frac{1}{2^{n+1}}|b-a|, \tag{5.19}$$

since the nth interval has length $|b-a|/2^n$. Note that this convergence criterion is independent of the actual function $f(x)$ as long as this function fulfils the conditions discussed in the conditions discussed in the previous subsection.

As an example, suppose we wish to find how many iteration steps are needed in order to obtain a relative precision of 10^{-12} for x_n in the interval $[50, 63]$, that is

$$\frac{|s - x_n|}{|s|} \leq 10^{-12}. \tag{5.20}$$

It suffices in our case to study $s \geq 50$, which results in

$$\frac{|s - x_n|}{50} \leq 10^{-12}, \tag{5.21}$$

and with Eq. (5.19) we obtain

$$\frac{13}{2^{n+1}50} \leq 10^{-12}, \tag{5.22}$$

meaning $n \geq 37$. The code for the bisection method can look like this

```
    /*
    ** This function
    ** calculates a root between x1 and x2 of a function
    ** pointed to by (*func) using the method of bisection
    ** The root is returned with an accuracy of +- xacc.
    */

double bisection(double (*func)(double), double x1, double x2, double xacc)
{
  int        j;
  double     dx, f, fmid, xmid, rtb;

  f    = (*func)(x1);
  fmid = (*func)(x2);
  if(f*fmid >= 0.0) {
    cout << "\n\nError in function rtbis():" << endl;
    cout << "\nroot in function must be within" << endl;
    cout << "x1 ='' << x1 << ``and x2 `` << x2 << endl;
    exit(1);
  }
  rtb = f < 0.0 ? (dx = x2 - x1, x1) : (dx = x1 - x2, x2);
  for(j = 0; j < max_iterations; j++) {
    fmid = (*func)(xmid = rtb + (dx *= 0.5));
    if (fmid <= 0.0) rtb=xmid;
    if(fabs(dx) < xacc || fmid == 0.0) return rtb;
  }
  cout << "Error in the bisection:" << endl;      // should never reach
     this point
  cout "Too many iterations!"  << endl;
  exit(1);
}
// End: function bisection
```

In this function we transfer the lower and upper limit of the interval where we seek the solution, $[x_1, x_2]$. The variable `xacc` is the precision we opt for. Note that in this function the test $f(s) < \delta$ is not implemented. Rather, the test is done through $f(s) = 0$, which is not necessarily a good option.

Note also that this function transfer a pointer to the name of the given function through **double** (*func)(**double**).

5.4 Newton-Raphson's method

Perhaps the most celebrated of all one-dimensional root-finding routines is Newton's method, also called the Newton-Raphson method. This method is distinguished from the previously discussed methods by the fact that it requires the evaluation of both the function f and its derivative f' at arbitrary points. In this sense, it is taylored to cases with e.g., transcendental equations of the type shown in Eq. (5.8) where it is rather easy to evaluate the derivative. If you can only calculate the derivative numerically and/or your function is not of the smooth type, we discourage the use of this method.

The Newton-Raphson formula consists geometrically of extending the tangent line at a current point until it crosses zero, then setting the next guess to the abscissa of that zero-crossing. The mathematics behind this method is rather simple. Employing a Taylor expansion for x sufficiently close to the solution s, we have

$$f(s) = 0 = f(x) + (s-x)f'(x) + \frac{(s-x)^2}{2}f''(x) + \dots. \tag{5.23}$$

For small enough values of the function and for well-behaved functions, the terms beyond linear are unimportant, hence we obtain

$$f(x) + (s-x)f'(x) \approx 0, \tag{5.24}$$

yielding

$$s \approx x - \frac{f(x)}{f'(x)}. \tag{5.25}$$

Having in mind an iterative procedure, it is natural to start iterating with

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{5.26}$$

This is Newton-Raphson's method. It has a simple geometric interpretation, namely x_{n+1} is the point where the tangent from $(x_n, f(x_n))$ crosses the $x-$axis. Close to the solution, Newton-Raphson converges fast to the desired result. However, if we are far from a root, where the higher-order terms in the series are important, the Newton-Raphson formula can give grossly inaccurate results. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function. If an iteration places a trial guess near such a local extremum, so that the first derivative nearly vanishes, then Newton-Raphson may fail totally. An example is shown in Fig. 5.2

It is also possible to extract the convergence behavior of this method. Assume that the function f has a continuous second derivative around the solution s. If we define

$$e_{n+1} = x_{n+1} - s = x_n - \frac{f(x_n)}{f'(x_n)} - s, \tag{5.27}$$

and using Eq. (5.23) we have

$$e_{n+1} = e_n + \frac{-e_n f'(x_n) + e_n^2/2f''(\xi)}{f'(x_n)} = \frac{e_n^2/2f''(\xi)}{f'(x_n)}. \tag{5.28}$$

This gives

$$\frac{|e_{n+1}|}{|e_n|^2} = \frac{1}{2}\frac{|f''(\xi)|}{|f'(x_n)|^2} = \frac{1}{2}\frac{|f''(s)|}{|f'(s)|^2} \tag{5.29}$$

when $x_n \to s$. Our error constant k is then proportional to $|f''(s)|/|f'(s)|^2$ if the second derivative is different from zero. Clearly, if the first derivative is small, the convergence is slower. In general, if we are

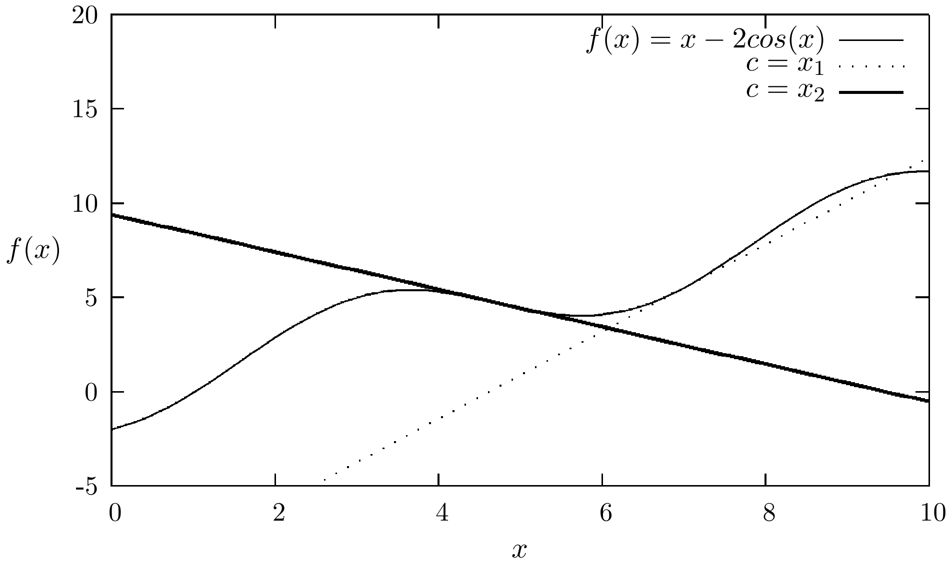


Figure 5.2: Example of a case where Newton-Raphson's method does not converge. For the function $f(x) = x - 2cos(x)$, we see that if we start at $x = 7$, the first iteration gives us that the first point where we cross the $x-$axis is given by x_1. However, using x_1 as a starting point for the next iteration results in a point x_2 which is close to a local minimum. The tangent here is close to zero and we will never approach the point where $f(x) = 0$.

able to start the iterative procedure near a root and we can easily evaluate the derivative, this is the method of choice. In cases where we may need to evaluate the derivative numerically, the previously described methods are easier and most likely safer to implement with respect to loss of numerical precision. Recall that the numerical evaluation of derivatives involves differences between function values at different x_n.

We can rewrite the last equation as

$$|e_{n+1}| = C|e_n|^2, \tag{5.30}$$

with C a constant. If we assume that $C \sim 1$ and let $e_n \sim 10^{-8}$, this results in $e_{n+1} \sim 10^{-16}$, and demonstrates clearly why Newton-Raphson's method may converge faster than the bisection method.

Summarizing, this method has a solution when f'' is continuous and s is a simple zero of f. Then there is a neighborhood of s and a constant C such that if Newton-Raphson's method is started in that neighborhood, the successive points become steadily closer to s and satisfy

$$|s - x_{n+1}| \le C|s - x_n|^2,$$

with $n \ge 0$. In some situations, the method guarantees to converge to a desired solution from an arbitrary starting point. In order for this to take place, the function f has to belong to $C^2(R)$, be increasing, convex and having a zero. Then this zero is unique and Newton's method converges to it from any starting point.

As a mere curiosity, suppose we wish to compute the square root of a number R, i.e., $\sqrt{R}$. Let $R > 0$ and define a function

$$f(x) = x^2 - R.$$

The variable x is a root if $f(x) = 0$. Newton-Raphson's method yields then the following iterative approach to the root

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{R}{x_n}\right), \tag{5.31}$$

a formula credited to Heron, a Greek engineer and architect who lived sometime between 100 B.C. and A.D. 100.

Suppose we wish to compute $\sqrt{13} = 3.6055513$ and start with $x_0 = 5$. The first iteration gives $x_1 = 3.8$, $x_2 = 3.6105263$, $x_3 = 3.6055547$ and $x_4 = 3.6055513$. With just four iterations and a not too optimal choice of x_0 we obtain the exact root to a precision of 8 digits. The above equation, together with range reduction , is used in the intrisic computational function which computes square roots.

Newton's method can be generalized to systems of several non-linear equations and variables. Consider the case with two equations

$$\begin{array}{ll} f_1(x_1, x_2) & = 0 \\ f_2(x_1, x_2) & = 0 \end{array}, \tag{5.32}$$

which we Taylor expand to obtain

$$\begin{array}{rl} 0 = f_1(x_1 + h_1, x_2 + h_2) = & f_1(x_1, x_2) + h_1 \partial f_1/\partial x_1 + h_2 \partial f_1/\partial x_2 + \dots \\ 0 = f_2(x_1 + h_1, x_2 + h_2) = & f_2(x_1, x_2) + h_1 \partial f_2/\partial x_1 + h_2 \partial f_2/\partial x_2 + \dots \end{array}. \tag{5.33}$$

Defining the Jacobian matrix $\hat{\mathbf{J}}$ we have

$$\hat{\mathbf{J}} = \left(\begin{array}{cc} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{array} \right), \tag{5.34}$$

we can rephrase Newton's method as

$$\left(\begin{array}{c} x_1^{n+1} \\ x_2^{n+1} \end{array} \right) = \left(\begin{array}{c} x_1^n \\ x_2^n \end{array} \right) + \left(\begin{array}{c} h_1^n \\ h_2^n \end{array} \right), \tag{5.35}$$

where we have defined

$$\left(\begin{array}{c} h_1^n \\ h_2^n \end{array} \right) = -\hat{\mathbf{J}}^{-1} \left(\begin{array}{c} f_1(x_1^n, x_2^n) \\ f_2(x_1^n, x_2^n) \end{array} \right). \tag{5.36}$$

We need thus to compute the inverse of the Jacobian matrix and it is to understand that difficulties may arise in case $\hat{\mathbf{J}}$ is nearly singular.

It is rather straightforward to extend the above scheme to systems of more than two non-linear equations.

The code for Newton-Raphson's method can look like this

```
      /*
      ** This function
      ** calculates a root between x1 and x2 of a function pointed to
      ** by (*funcd) using the Newton-Raphson method. The user-defined
      ** function funcd() returns both the function value and its first
      ** derivative at the point x,
      ** The root is returned with an accuracy of +- xacc.
      */

double newtonraphson(void (*funcd)(double, double *, double *), double x1,
   double x2,
         double xacc)
{
   int      j;
   double   df, dx, f, rtn;
```

```
  rtn = 0.5 * (x1 + x2);                    // initial guess
  for(j = 0; j < max_iterations; j++) {
     (*funcd)(rtn, &f, &df);
     dx    = f/df;
     rtn -= dx;
     if((x1 - rtn) * (rtn - x2) < 0.0)  {
        cout << "\n\nError in function newtonraphson:" << endl ;
        cout << "Jump out of interval bracket" << endl;
        exit(1);
     }
     if (fabs(dx) < xacc) return rtn;
  }
  cout << "Error in function newtonraphson:" << endl;
  cout << "Too many iterations!" << endl;
  exit(1);
}
// End: function newtonraphson
```

We transfer again the lower and upper limit of the interval where we seek the solution, $[x_1, x_2]$ and the variable `xacc`. Firthermore, it transfers a pointer to the name of the given function through **double** (*func)(**double**).

5.5 The secant method and other methods

For functions that are smooth near a root, the methods known respectively as false position (or regula falsi) and secant method generally converge faster than bisection but slower than Newton-Raphson. In both of these methods the function is assumed to be approximately linear in the local region of interest, and the next improvement in the root is taken as the point where the approximating line crosses the axis.

The algorithm for obtaining the solution for the secant method is rather simple. We start with the definition of the derivative

$$f'(x_n) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

and combine it with the iterative expression of Newton-Raphson's

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

to obtain

$$x_{n+1} = x_n - f(x_n)\left(\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}\right), \tag{5.37}$$

which we rewrite to

$$x_{n+1} = \frac{f(x_n)x_{n-1} - f(x_{n-1})x_n}{f(x_n) - f(x_{n-1})}. \tag{5.38}$$

This is the secant formula, implying that we are drawing a straight line from the point $(x_{n-1}, f(x_{n-1}))$ to $(x_n, f(x_n))$. Where it crosses the $x - axis$ we have the new point x_{n+1}. This is illustrated in Fig. 5.3.

In the numerical implementation found in the program library, the quantities x_{n-1}, x_n, x_{n+1} are changed to a, b and c respectively, i.e., we determine c by the point where a straight line from the point

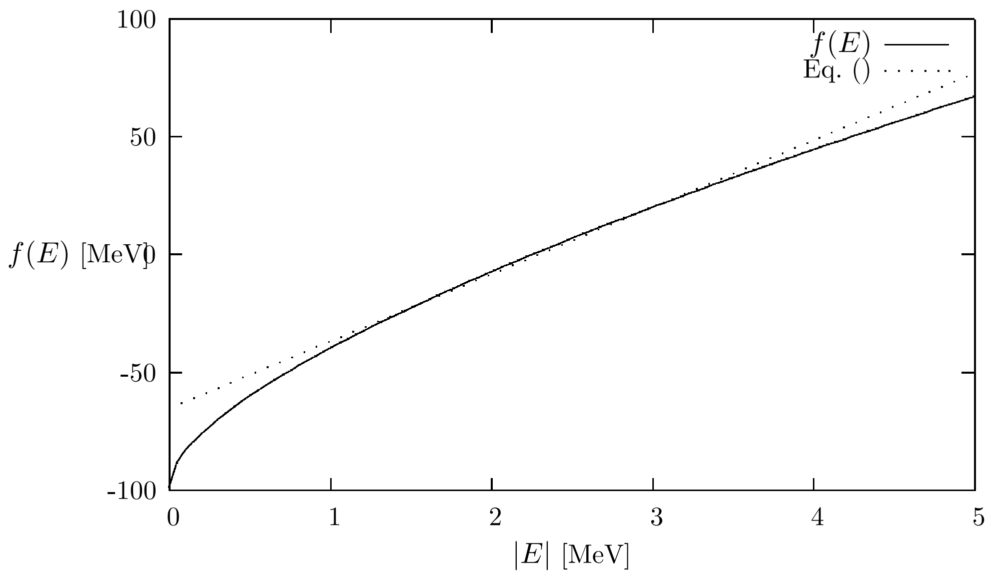


Figure 5.3: Plot of $f(E)$ Eq. (5.8) as function of energy |E|. The point c is determined by where the straight line from $(a, f(a))$ to $(b, f(b))$ crosses the $x - axis$.

$(a, f(a))$ to $(b, f(b))$ crosses the $x - axis$, that is

$$c = \frac{f(b)a - f(a)b}{f(b) - f(a)}. \tag{5.39}$$

We then see clearly the difference between the bisection method and the secant method. The convergence criterion for the secant method is

$$|e_{n+1}| \approx A|e_n|^{\alpha}, \tag{5.40}$$

with $\alpha \approx 1.62$. The convergence is better than linear, but not as good as Newton-Raphson's method which converges quadratically.

While the secant method formally converges faster than bisection, one finds in practice pathological functions for which bisection converges more rapidly. These can be choppy, discontinuous functions, or even smooth functions if the second derivative changes sharply near the root. Bisection always halves the interval, while the secant method can sometimes spend many cycles slowly pulling distant bounds closer to a root. We illustrate the weakness of this method in Fig. 5.4 where we show the results of the first three iterations, i.e., the first point is $c = x_1$, the next iteration gives $c = x_2$ while the third iterations ends with $c = x_3$. We may risk that one of the endpoints is kept fixed while the other one only slowly converges to the desired solution.

The search for the solution s proceeds in much of the same fashion as for the bisection method, namely after each iteration one of the previous boundary points is discarded in favor of the latest estimate of the root. A variation of the secant method is the so-called false position method (regula falsi from Latin) where the interval [a,b] is chosen so that $f(a)f(b) < 0$, else there is no solution. This is rather similar to the bisection method. Another possibility is to determine the starting point for the iterative search using three points $(a, f(a))$, $(b, f(b))$ and $(c, f(c))$. One can use Lagrange's interpolation formula for a polynomial, see the discussion in the next chapter. This procedure leads to Brent's method.

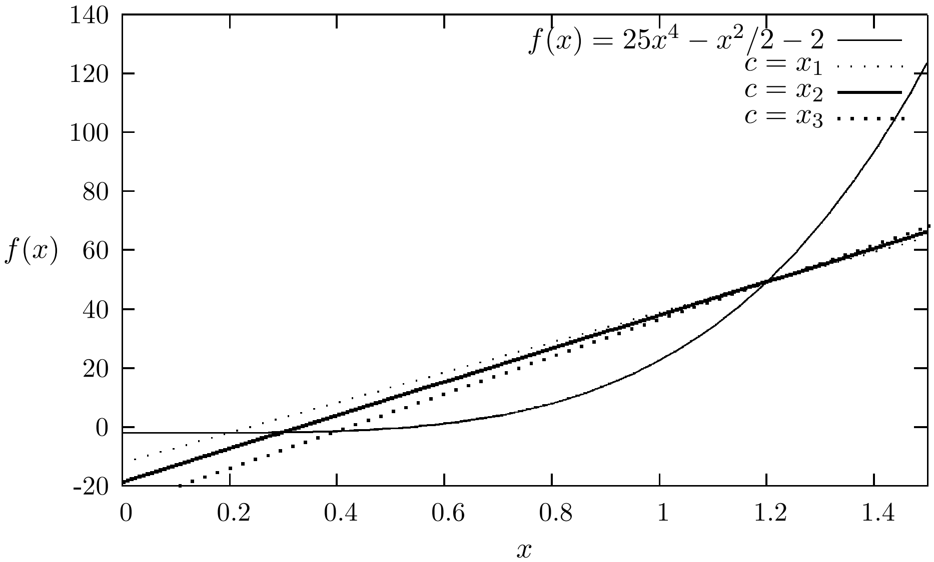


Figure 5.4: Plot of $f(x) = 25x^4 - x^2/2 - 2$. The various straight lines correspond to the determination of the point c after each iteration. c is determined by where the straight line from $(a, f(a))$ to $(b, f(b))$ crosses the $x - axis$. Here we have chosen three values for c, x_1, x_2 and x_3 which refer to the first, second and third iterations respectively.

5.6 Exercises and projects

Exercise 5.1: Comparison of methods

Write a code which implements the bisection method, Newton-Raphson's method and the secant method.

Find the positive roots of

$$x^2 - 4x\sin x + (2\sin x)^2 = 0,$$

using these three methods and compare the achieved accuracy number of iterations needed to find the solution. Give a critical discussion of the methods.

Project 5.1: Schrödinger's equation

We are going to study the solution of the Schrödinger equation (SE) for a system with a neutron and proton (the deuteron) moving in a simple box potential.

We begin our discussion of the SE with the neutron-proton (deuteron) system with a box potential $V(r)$. We define the radial part of the wave function $R(r)$ and introduce the definition $u(r) = rR(R)$ The radial part of the SE for two particles in their center-of-mass system and with orbital momentum $l = 0$ is then

$$-\frac{\hbar^2}{m}\frac{d^2u(r)}{dr^2} + V(r)u(r) = Eu(r),$$

with

$$m = 2\frac{m_p m_n}{m_p + m_n},$$

where m_p and m_n are the masses of the proton and neutron, respectively. We use here $m = 938$ MeV.

Our potential is defined as

$$V(r) = \begin{cases} -V_0 & 0 \leq r < a \\ 0 & r > a \end{cases}$$

Bound states correspond to negative energy E and scattering states are given by positive energies. The SE takes the form (without specifying the sign of E)

$$\frac{d^2u(r)}{dr^2} + \frac{m}{\hbar^2}\left(V_0 + E\right)u(r) = 0 \qquad r < a,$$

and

$$\frac{d^2u(r)}{dr^2} + \frac{m}{\hbar^2}Eu(r) = 0 \qquad r > a.$$

We are now going to search for eventual bound states, i.e., $E < 0$. The deuteron has only one bound state at energy $E = -2.223$ MeV. Discuss the boundary conditions on the wave function and use these to show that the solution to the SE is

$$u(r) = Asin(kr) \qquad r < a,$$

and

$$u(r) = B\exp\left(-\beta r\right) \qquad r > a,$$

where A and B are constants. We have also defined

$$k = \sqrt{m(V_0 - |E|)}/\hbar,$$

and

$$\beta = \sqrt{m|E|}/\hbar.$$

Show then, using the continuity requirement on the wave function that at $r = a$ you obtain the transcendental equation

$$kcot(ka) = -\beta. \tag{5.41}$$

Insert values of $V_0 = 60$ MeV and $a = 1.45$ fm (1 fm = 10^{-15} m) and make a plot plotting programs) of Eq. (5.41) as function of energy E in order to find eventual eigenvalues. See if these values result in a bound state for E.

When you have localized on your plot the point(s) where Eq. (5.41) is satisfied, obtain a numerical value for E using Newton-Raphson's method, the bisection method and the secant method. Make an analysis of these three methods and discuss how many iterations are needed to find a stable solution.

What is smallest possible value of V_0 which gives a bound state?

Chapter 6

Numerical interpolation, extrapolation and fitting of data

6.1 Introduction

Numerical interpolation and extrapolation are frequently used tools in numerical applications to physics. The often encountered situation is that of a function f at a set of points $x_1 \dots x_n$ where an analytic form is missing. The function f may represent some data points from experiment or the result of a lengthy large-scale computation of some physical quantity that cannot be cast into a simple analytical form.

We may then need to evaluate the function f at some point x within the data set $x_1 \dots x_n$, but where x differs from the tabulated values. In this case we are dealing with interpolation. If x is outside we are left with the more troublesome problem of numerical extrapolation. Below we will concentrate on two methods for interpolation and extrapolation, namely polynomial interpolation and extrapolation and the qubic spline interpolation approach.

6.2 Interpolation and extrapolation

6.2.1 Polynomial interpolation and extrapolation

Let us assume that we have a set of $N+1$ points $y_0 = f(x_0), y_1 = f(x_1), \dots, y_N = f(x_N)$ where none of the x_i values are equal. We wish to determine a polynomial of degree n so that

$$P_N(x_i) = f(x_i) = y_i, \qquad i = 0, 1, \dots, N \tag{6.1}$$

for our data points. If we then write P_n on the form

$$P_N(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \dots + a_N(x - x_0)\dots(x - x_{N-1}), \tag{6.2}$$

then Eq. (6.1) results in a triangular system of equations

$$\begin{array}{llll} a_0 & = f(x_0) & & \\ a_0 + & a_1(x_1 - x_0) & = f(x_1) & \\ a_0 + & a_1(x_2 - x_0) + & a_2(x_2 - x_0)(x_2 - x_1) & = f(x_2) \\ \dots & \dots & \dots & \dots \end{array} . \tag{6.3}$$

The coefficients $a_0, \dots, a_N$ are then determined in a recursive way, starting with $a_0, a_1, \dots$.

The classic of interpolation formulae was created by Lagrange and is given by

$$P_N(x) = \sum_{i=0}^{N} \prod_{k \neq i} \frac{x - x_k}{x_i - x_k} y_i. \tag{6.4}$$

If we have just two points (a straight line) we get

$$P_1(x) = \frac{x - x_0}{x_1 - x_0} y_1 + \frac{x - x_1}{x_0 - x_1} y_0, \tag{6.5}$$

and with three points (a parabolic approximation) we have

$$P_3(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} y_2 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} y_1 + \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} y_0 \tag{6.6}$$

and so forth. It is easy to see from the above equations that when $x = x_i$ we have that $f(x) = f(x_i)$ It is also possible to show that the approximation error (or rest term) is given by the second term on the right hand side of

$$f(x) = P_N(x) + \frac{\omega_{N+1}(x) f^{(N+1)}(\xi)}{(N+1)!}. \tag{6.7}$$

The function $\omega_{N+1}(x)$ is given by

$$\omega_{N+1}(x) = a_N(x - x_0) \dots (x - x_N), \tag{6.8}$$

and $\xi = \xi(x)$ is a point in the smallest interval containing all interpolation points x_j and x. The algorithm we provide however (the code POLINT in the program library) is based on divided differences. The recipe is quite simple. If we take $x = x_0$ in Eq. (6.2), we then have obviously that $a_0 = f(x_0) = y_0$. Moving a_0 over to the left-hand side and dividing by $x - x_0$ we have

$$\frac{f(x) - f(x_0)}{x - x_0} = a_1 + a_2(x - x_1) + \dots + a_N(x - x_1)(x - x_2) \dots (x - x_{N-1}), \tag{6.9}$$

where we hereafter omit the rest term

$$\frac{f^{(N+1)}(\xi)}{(N+1)!}(x - x_1)(x - x_2) \dots (x - x_N). \tag{6.10}$$

The quantity

$$f_{0x} = \frac{f(x) - f(x_0)}{x - x_0}, \tag{6.11}$$

is a divided difference of first order. If we then take $x = x_1$, we have that $a_1 = f_{01}$. Moving a_1 to the left again and dividing by $x - x_1$ we obtain

$$\frac{f_{0x} - f_{01}}{x - x_1} = a_2 + \dots + a_N(x - x_2) \dots (x - x_{N-1}). \tag{6.12}$$

and the quantity

$$f_{01x} = \frac{f_{0x} - f_{01}}{x - x_1}, \tag{6.13}$$

is a divided difference of second order. We note that the coefficient

$$a_1 = f_{01}, \tag{6.14}$$

is determined from f_{0x} by setting $x = x_1$. We can continue along this line and define the divided difference of order $k+1$ as

$$f_{01\dots kx} = \frac{f_{01\dots(k-1)x} - f_{01\dots(k-1)k}}{x - x_k}, \tag{6.15}$$

meaning that the corresponding coefficient a_k is given by

$$a_k = f_{01\dots(k-1)k}. \tag{6.16}$$

With these definitions we see that Eq. (6.7) can be rewritten as

$$f(x) = a_0 + \sum_{k=1} N f_{01\dots k}(x - x_0)\dots(x - x_{k-1}) + \frac{\omega_{N+1}(x) f^{(N+1)}(\xi)}{(N+1)!}. \tag{6.17}$$

If we replace $x_0, x_1, \dots, x_k$ in Eq. (6.15) with $x_{i+1}, x_{i+2}, \dots, x_k$, that is we count from $i+1$ to k instead of counting from 0 to k and replace x with x_i, we can then construct the following recursive algorithm for the calculation of divided differences

$$f_{x_i x_{i+1}\dots x_k} = \frac{f_{x_{i+1}\dots x_k} - f_{x_i x_{i+1}\dots x_{k-1}}}{x_k - x_i}. \tag{6.18}$$

Assuming that we have a table with function values $(x_j, f(x_j) = y_j)$ and need to construct the coefficients for the polynomial $P_N(x)$. We can then view the last equation by constructing the following table for the case where $N = 3$.

$$\begin{array}{cccccc}
x_0 & y_0 & & & & \\
 & & f_{x_0x_1} & & & \\
x_1 & y_1 & & f_{x_0x_1x_2} & & \\
 & & f_{x_1x_2} & & f_{x_0x_1x_2x_3} & . \\
x_2 & y_2 & & f_{x_1x_2x_3} & & \\
 & & f_{x2x_3} & & & \\
x_3 & y_3 & & & &
\end{array} \tag{6.19}$$

The coefficients we are searching for will then be the elements along the main diagonal. We can understand this algorithm by considering the following. First we construct the unique polynomial of order zero which passes through the point x_0, y_0. This is just a_0 discussed above. Therafter we construct the unique polynomial of order one which passes through both $x_0 y_0$ and $x_1 y_1$. This corresponds to the coefficient a_1 and the tabulated value $f_{x_0x_1}$ and together with a_0 results in the polynomial for a straight line. Likewise we define polynomial coefficients for all other couples of points such as $f_{x_1x_2}$ and $f_{x_2x_3}$. Furthermore, a coefficient like $a_2 = f_{x_0x_1x_2}$ spans now three points, and adding together $f_{x_0x_1}$ we obtain a polynomial which represents three points, a parabola. In this fashion we can continue till we have all coefficients. The function POLINT included in the library is based on an extension of this algorithm, knowns as Neville's algorithm. It is based on equidistant interpolation points. The error provided by the call to the function POLINT is based on the truncation error in Eq. (6.7).

Exercise 6.1
Use the function $f(x) = x^3$ to generate function values at four points $x_0 = 0$, $x_1 = 1$, $x_2 = 5$ and $x_3 = 6$. Use the above described method to show that the interpolating polynomial becomes $P_3(x) = x + 6x(x-1) + x(x-1)(x-5)$. Compare the exact answer with the polynomial P_3 and estimate the rest term.

6.3 Richardson's deferred extrapolation method

Here we present an elegant method to improve the precision of our mathematical truncation, without too many additional function evaluations. We will again study the evaluation of the first and second derivatives of $\exp(x)$ at a given point $x = \xi$. In Eqs. (3.3) and (3.4) for the first and second derivatives, we noted that the truncation error goes like $O(h^{2j})$.

Employing the mid-point approximation to the derivative, the various derivatives D of a given function $f(x)$ can then be written as

$$D(h) = D(0) + a_1h^2 + a_2h^4 + a_3h^6 + \dots, \tag{6.20}$$

where $D(h)$ is the calculated derivative, $D(0)$ the exact value in the limit $h \to 0$ and a_i are independent of h. By choosing smaller and smaller values for h, we should in principle be able to approach the exact value. However, since the derivatives involve differences, we may easily loose numerical precision as shown in the previous sections. A possible cure is to apply Richardson's deferred approach, i.e., we perform calculations with several values of the step h and extrapolate to $h = 0$. The philososphy is to combine different values of h so that the terms in the above equation involve only large exponents for h. To see this, assume that we mount a calculation for two values of the step h, one with h and the other with $h/2$. Then we have

$$D(h) = D(0) + a_1h^2 + a_2h^4 + a_3h^6 + \dots, \tag{6.21}$$

and

$$D(h/2) = D(0) + \frac{a_1h^2}{4} + \frac{a_2h^4}{16} + \frac{a_3h^6}{64} + \dots, \tag{6.22}$$

and we can eliminate the term with a_1 by combining

$$D(h/2) + \frac{D(h/2) - D(h)}{3} = D(0) - \frac{a_2h^4}{4} - \frac{5a_3h^6}{16}. \tag{6.23}$$

We see that this approximation to $D(0)$ is better than the two previous ones since the error now goes like $O(h^4)$. As an example, let us evaluate the first derivative of a function f using a step with lengths h and $h/2$. We have then

$$\frac{f_h - f_{-h}}{2h} = f_0' + O(h^2), \tag{6.24}$$

$$\frac{f_{h/2} - f_{-h/2}}{h} = f_0' + O(h^2/4), \tag{6.25}$$

which can be combined, using Eq. (6.23) to yield

$$\frac{-f_h + 8f_{h/2} - 8f_{-h/2} + f_{-h}}{6h} = f_0' - \frac{h^4}{480}f^{(5)}. \tag{6.26}$$

In practice, what happens is that our approximations to $D(0)$ goes through a series of steps

$$\begin{array}{llll} D_0^{(0)} & & & \\ D_0^{(1)} & D_1^{(0)} & & \\ D_0^{(2)} & D_1^{(1)} & D_2^{(0)} & \\ D_0^{(3)} & D_1^{(2)} & D_2^{(1)} & D_3^{(0)} \\ \dots & \dots & \dots & \dots \end{array}, \tag{6.27}$$

where the elements in the first column represent the given approximations

$$D_0^{(k)} = D(h/2^k). \tag{6.28}$$

This means that $D_1^{(0)}$ in the second column and row is the result of the extrapolating based on $D_0^{(0)}$ and $D_0^{(1)}$. An element $D_m^{(k)}$ in the table is then given by

$$D_m^{(k)} = D_{m-1}^{(k)} + \frac{D_{m-1}^{(k+1)} - D_{m-1}^{(k)}}{4^m - 1} \tag{6.29}$$

with $m > 0$. I.e., it is a linear combination of the element to the left of it and the element right over the latter.

In Table 3.1 we presented the results for various step sizes for the second derivative of $\exp{(x)}$ using $f_0'' = \frac{f_h - 2f_0 + f_{-h}}{h^2}$. The results were compared with the exact ones for various x values. Note well that as the step is decreased we get closer to the exact value. However, if it is further increased, we run into problems of loss of precision. This is clearly seen for $h = 0000001$. This means that even though we could let the computer run with smaller and smaller values of the step, there is a limit for how small the step can be made before we loose precision. Consider now the results in Table 6.1 where we choose to employ Richardson's extrapolation scheme. In this calculation we have computed our function with only three possible values for the step size, namely h, $h/2$ and $h/4$ with $h = 0.1$. The agreement with the exact value is amazing! The extrapolated result is based upon the use of Eq. (6.29). We will use this

x	$h = 0.1$	$h = 0.05$	$h = 0.025$	Extrapolat	Error
0.0	1.00083361	1.00020835	1.00005208	1.00000000	0.00000000
1.0	2.72054782	2.71884818	2.71842341	2.71828183	0.00000001
2.0	7.39521570	7.39059561	7.38944095	7.38905610	0.00000003
3.0	20.10228045	20.08972176	20.08658307	20.08553692	0.00000009
4.0	54.64366366	54.60952560	54.60099375	54.59815003	0.00000024
5.0	148.53687797	148.44408109	148.42088912	148.41315910	0.00000064

Table 6.1: Result for numerically calculated second derivatives of $\exp{(x)}$ using extrapolation. The first three values are those calculated with three different step sizes, h, $h/2$ and $h/4$ with $h = 0.1$. The extrapolated result to $h = 0$ should then be compared with the exact ones from Table 3.1.

method to obtain improved eigenvalues in chapter 12.

6.4 Qubic spline interpolation

Qubic spline interpolation is among one of the mostly used methods for interpolating between data points where the arguments are organized as ascending series. In the library program we supply such a function, based on the so-called qubic spline method to be described below.

A spline function consists of polynomial pieces defined on subintervals. The different subintervals are connected via various continuity relations.

Assume we have at our disposal $n + 1$ points $x_0, x_1, \dots x_n$ arranged so that $x_0 < x_1 < x_2 < \dots x_{n-1} < x_n$ (such points are called knots). A spline function s of degree k with $n + 1$ knots is defined as follows

- On every subinterval $[x_{i-1}, x_i)$ s is a polynomial of degree $\leq k$.
- s has $k-1$ continuous derivatives in the whole interval $[x_0, x_n]$.

As an example, consider a spline function of degree $k = 1$ defined as follows

$$s(x) = \begin{cases} s_0(x) = a_0x + b_0 & x \in [x_0, x_1) \\ s_1(x) = a_1x + b_1 & x \in [x_1, x_2) \\ \dots & \dots \\ s_{n-1}(x) = a_{n-1}x + b_{n-1} & x \in [x_{n-1}, x_n] \end{cases} \tag{6.30}$$

In this case the polynomial consists of series of straight lines connected to each other at every endpoint. The number of continuous derivatives is then $k - 1 = 0$, as expected when we deal with straight lines. Such a polynomial is quite easy to construct given $n+1$ points $x_0, x_1, \dots x_n$ and their corresponding function values.

The most commonly used spline function is the one with $k = 3$, the so-called qubic spline function. Assume that we have in adddition to the $n + 1$ knots a series of functions values $y_0 = f(x_0), y_1 = f(x_1), \dots y_n = f(x_n)$. By definition, the polynomials s_{i-1} and s_i are thence supposed to interpolate the same point i, i.e.,

$$s_{i-1}(x_i) = y_i = s_i(x_i), \tag{6.31}$$

with $1 \leq i \leq n-1$. In total we have n polynomials of the type

$$s_i(x) = a_{i0} + a_{i1}x + a_{i2}x^2 + a_{i2}x^3, \tag{6.32}$$

yielding $4n$ coefficients to determine. Every subinterval provides in addition the $2n$ conditions

$$y_i = s(x_i), \tag{6.33}$$

and

$$s(x_{i+1}) = y_{i+1}, \tag{6.34}$$

to be fulfilled. If we also assume that s' and s'' are continuous, then

$$s'_{i-1}(x_i) = s'_i(x_i), \tag{6.35}$$

yields $n - 1$ conditions. Similarly,

$$s''_{i-1}(x_i) = s''_i(x_i), \tag{6.36}$$

results in additional $n-1$ conditions. In total we have $4n$ coefficients and $4n-2$ equations to determine them, leaving us with 2 degrees of freedom to be determined.

Using the last equation we define two values for the second derivative, namely

$$s''_i(x_i) = f_i, \tag{6.37}$$

and

$$s''_i(x_{i+1}) = f_{i+1}, \tag{6.38}$$

and setting up a straight line between f_i and f_{i+1} we have

$$s''_i(x) = \frac{f_i}{x_{i+1} - x_i}(x_{i+1} - x) + \frac{f_{i+1}}{x_{i+1} - x_i}(x - x_i), \tag{6.39}$$

and integrating twice one obtains

$$s_i(x) = \frac{f_i}{6(x_{i+1} - x_i)}(x_{i+1} - x)^3 + \frac{f_{i+1}}{6(x_{i+1} - x_i)}(x - x_i)^3 + c(x - x_i) + d(x_{i+1} - x). \qquad (6.40)$$

Using the conditions $s_i(x_i) = y_i$ and $s_i(x_{i+1}) = y_{i+1}$ we can in turn determine the constants c and d resulting in

$$\begin{aligned} s_i(x) = & \quad \tfrac{f_i}{6(x_{i+1}-x_i)}(x_{i+1} - x)^3 + \tfrac{f_{i+1}}{6(x_{i+1}-x_i)}(x - x_i)^3 \\ + & \quad (\tfrac{y_{i+1}}{x_{i+1}-x_i} - \tfrac{f_{i+1}(x_{i+1}-x_i)}{6})(x - x_i) + (\tfrac{y_i}{x_{i+1}-x_i} - \tfrac{f_i(x_{i+1}-x_i)}{6})(x_{i+1} - x). \end{aligned} \qquad (6.41)$$

How to determine the values of the second derivatives f_i and f_{i+1}? We use the continuity assumption of the first derivatives

$$s'_{i-1}(x_i) = s'_i(x_i), \qquad (6.42)$$

and set $x = x_i$. Defining $h_i = x_{i+1} - x_i$ we obtain finally the following expression

$$h_{i-1}f_{i-1} + 2(h_i + h_{i-1})f_i + h_i f_{i+1} = \frac{6}{h_i}(y_{i+1} - y_i) - \frac{6}{h_{i-1}}(y_i - y_{i-1}), \qquad (6.43)$$

and introducing the shorthands $u_i = 2(h_i + h_{i-1})$, $v_i = \frac{6}{h_i}(y_{i+1} - y_i) - \frac{6}{h_{i-1}}(y_i - y_{i-1})$, we can reformulate the problem as a set of linear equations to be solved through e.g., Gaussian elemination, namely

$$\left[\begin{array}{cccccccc} u_1 & h_1 & 0 & \dots & & & & \\ h_1 & u_2 & h_2 & 0 & \dots & & & \\ 0 & h_2 & u_3 & h_3 & 0 & \dots & & \\ \dots & & \dots & \dots & \dots & \dots & \dots & \\ & \dots & & & 0 & h_{n-3} & u_{n-2} & h_{n-2} \\ & & & & & 0 & h_{n-2} & u_{n-1} \end{array}\right] \left[\begin{array}{c} f_1 \\ f_2 \\ f_3 \\ \dots \\ f_{n-2} \\ f_{n-1} \end{array}\right] = \left[\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ \dots \\ v_{n-2} \\ v_{n-1} \end{array}\right]. \qquad (6.44)$$

Note that this is a set of tridiagonal equations and can be solved through only $O(n)$ operations. The functions supplied in the program library are *spline* and *splint*. In order to use qubic spline interpolation you need first to call

```
spline(double x[], double y[], int n, double yp1, double yp2, double y2[])
```

This function takes as input $x[0,..,n-1]$ and $y[0,..,n-1]$ containing a tabulation $y_i = f(x_i)$ with $x_0 < x_1 < .. < x_{n-1}$ together with the first derivatives of $f(x)$ at x_0 and x_{n-1}, respectively. Then the function returns $y2[0,..,n-1]$ which contanin the second derivatives of $f(x_i)$ at each point x_i. n is the number of points. This function provides the qubic spline interpolation for all subintervals and is called only once. Thereafter, if you wish to make various interpolations, you need to call the function

```
splint(double x[], double y[], double y2a[], int n, double x, double *y)
```

which takes as input the tabulated values $x[0,..,n-1]$ and $y[0,..,n-1]$ and the output y2a[0,..,n - 1] from *spline*. It returns the value y corresponding to the point x.

Chapter 7

Numerical integration

7.1 Introduction

In this chapter we discuss some of the classic formulae such as the trapezoidal rule and Simpson's rule for equally spaced abscissas and formulae based on Gaussian quadrature. The latter are more suitable for the case where the abscissas are not equally spaced. The emphasis is on methods for evaluating one-dimensional integrals. In chapter 8 we show how Monte Carlo methods can be used to compute multi-dimensional integrals. We discuss also how to compute singular integrals and outline a physics project which combines numerical integration techniques and inverse of a matrix to solve quantum mechanical scattering problems.

We end this chapter with an extensive discussion on MPI and parallel computing. The examples focus on parallelization of algorithms for computing integrals.

The integral

$$I = \int_a^b f(x)dx \tag{7.1}$$

has a very simple meaning. If we consider Fig. 7.1 the integral I simply represents the area enscribed by the function $f(x)$ starting from $x = a$ and ending at $x = b$. Two main methods will be discussed below, the first one being based on equal (or allowing for slight modifications) steps and the other on more adaptive steps, namely so-called Gaussian quadrature methods. Both main methods encompass a plethora of approximations and only some of them will be discussed here.

7.2 Newton-Cotes quadrature: equal step methods

In considering equal step methods, our basic tool is the Taylor expansion of the function $f(x)$ around a point x and a set of surrounding neighbouring points. The algorithm is rather simple, and the number of approximations unlimited!

– Choose a step size

$$h = \frac{b-a}{N}$$

where N is the number of steps and a and b the lower and upper limits of integration.

– Choose then to stop the Taylor expansion of the function $f(x)$ at a certain derivative. You should also choose how many points around x are to be included in the evaluation of the derivatives.

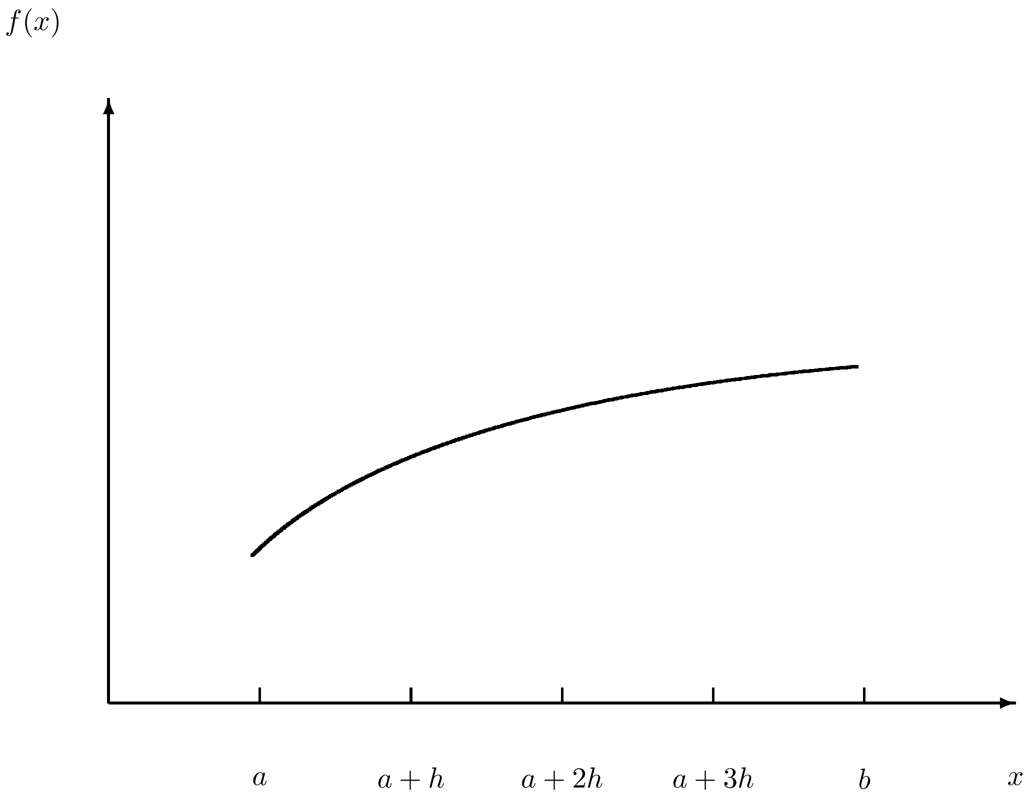


Figure 7.1: Area enscribed by the function $f(x)$ starting from $x = a$ to $x = b$. It is subdivided in several smaller areas whose evaluation is to be approximated by the techniques discussed in the text. The areas under the curve can for example be approximated by rectangular boxes or trapezoids.

– With these approximations to $f(x)$ perform the integration.

Such a small measure may seemingly allow for the derivation of various integrals. To see this, let us briefly recall the discussion in the previous section and especially Fig. 3.1. First, we can rewrite the desired integral as

$$\int_a^b f(x)dx = \int_a^{a+2h} f(x)dx + \int_{a+2h}^{a+4h} f(x)dx + \dots \int_{b-2h}^{b} f(x)dx.$$

The strategy then is to find a reliable Taylor expansion for $f(x)$ in the smaller sub intervals. Consider e.g., evaluating

$$\int_{-h}^{+h} f(x)dx \tag{7.2}$$

where we will Taylor expand $f(x)$ around a point x_0, see Fig. 3.1. The general form for the Taylor expansion around x_0 goes like

$$f(x = x_0 \pm h) = f(x_0) \pm hf' + \frac{h^2 f''}{2} \pm \frac{h^3 f'''}{6} + O(h^4).$$

Let us now suppose that we split the integral in Eq. (7.2) in two parts, one from $-h$ to x_0 and the other from x_0 to h. Next we assume that we can use the two-point formula for the derivative, that is we can approximate $f(x)$ in these two regions by a straight line, as indicated in the figure. This means that every small element under the function $f(x)$ looks like a trapezoid, and as you may expect, the pertinent numerical approach to the integral bears the predictable name 'trapezoidal rule'. It means also that we are trying to approximate our function $f(x)$ with a first order polynomial, that is $f(x) = a + bx$. The constant b is the slope given by the first derivative at $x = x_0$

$$f' = \frac{f(x_0 + h) - f(x_0)}{h} + O(h),$$

or

$$f' = \frac{f(x_0) - f(x_0 - h)}{h} + O(h),$$

and if we stop the Taylor expansion at that point our function becomes,

$$f(x) = f_0 + \frac{f_h - f_0}{h}x + O(x^2),$$

for $x = x_0$ to $x = x_0 + h$ and

$$f(x) = f_0 + \frac{f_0 - f_{-h}}{h}x + O(x^2),$$

for $x = x_0 - h$ to $x = x_0$. The error goes like $O(x^2)$. If we then evaluate the integral we obtain

$$\int_{-h}^{+h} f(x)dx = \frac{h}{2}(f_h + 2f_0 + f_{-h}) + O(h^3), \tag{7.3}$$

which is the well-known trapezoidal rule. Concerning the error in the approximation made, $O(h^3) = O((b-a)^3/N^3)$, you should note the following. *This is the local error!* Since we are splitting the integral from a to b in N pieces, we will have to perform approximately N such operations. This means that the *global error* goes like $\approx O(h^2)$. To see that, we use the trapezoidal rule to compute the integral of Eq. (7.1),

$$I = \int_a^b f(x)dx = h\left(f(a)/2 + f(a+h) + f(a+2h) + \cdots + f(b-h) + f_b/2\right), \tag{7.4}$$

with a global error which goes like $O(h^2)$. The correct mathematical expression for the local error for the trapezoidal rule is

$$\int_a^b f(x)dx - \frac{b-a}{2}\left[f(a)+f(b)\right] = -\frac{h^3}{12}f^{(2)}(\xi),$$

and the global error reads

$$\int_a^b f(x)dx - T_h(f) = -\frac{b-a}{12}h^2 f^{(2)}(\xi),$$

where T_h is the trapezoidal result and $\xi \in [a,b]$. It can easily be implemented numerically through the following simple algorithm

- Choose the number of mesh points and fix the step.
- calculate $f(a)$ and $f(b)$ and multiply with $h/2$
- Perform a loop over $n = 1$ to $n-1$ ($f(a)$ and $f(b)$ are known) and sum up the terms $f(a+h)+f(a+2h)+f(a+3h)+\cdots+f(b-h)$. Each step in the loop corresponds to a given value $a+nh$.
- Multiply the final result by h and add $hf(a)/2$ and $hf(b)/2$.

A simple function which implements this algorithm is as follows

```
double trapezoidal_rule(double a, double b, int n, double (*func)(double))
{
      double trapez_sum;
      double fa, fb, x, step;
      int    j;
      step=(b-a)/((double) n);
      fa=(*func)(a)/2. ;
      fb=(*func)(b)/2. ;
      trapez_sum=0.;
      for (j=1; j <= n-1; j++){
         x=j*step+a;
         trapez_sum+=(*func)(x);
      }
      trapez_sum=(trapez_sum+fb+fa)*step;
      return trapez_sum;
}  // end trapezoidal_rule
```

The function returns a new value for the specific integral through the variable **trapez_sum**. There is one new feature to note here, namely the transfer of a user defined function called **func** in the definition

```
void trapezoidal_rule(double a, double b, int n, double *trapez_sum,
                      double (*func)(double) )
```

What happens here is that we are transferring a pointer to the name of a user defined function, which has as input a double precision variable and returns a double precision number. The function **trapezoidal_rule** is called as

```
trapezoidal_rule(a, b, n, &myfunction )
```

in the calling function. We note that **a**, **b** and **n** are called by value, while **trapez_sum** and the user defined function **my_function** are called by reference.

Another very simple approach is the so-called midpoint or rectangle method. In this case the integration area is split in a given number of rectangles with length h and height given by the mid-point value of the function. This gives the following simple rule for approximating an integral

$$I = \int_a^b f(x)dx \approx h\sum_{i=1}^{N} f(x_{i-1/2}), \tag{7.5}$$

where $f(x_{i-1/2})$ is the midpoint value of f for a given rectangle. We will discuss its truncation error below. It is easy to implement this algorithm, as shown here

```
double rectangle_rule(double a, double b, int n, double (*func)(double))
{
        double rectangle_sum;
        double fa, fb, x, step;
        int    j;
        step=(b-a)/((double) n);
        rectangle_sum=0.;
        for (j = 0; j <= n; j++){
           x = (j+0.5)*step+;    // midpoint of a given rectangle
           rectangle_sum+=(*func)(x);   //  add value of function.
        }
        rectangle_sum *= step;  //  multiply with step length.
        return rectangle_sum;
}  // end rectangle_rule
```

The correct mathematical expression for the local error for the rectangular rule $R_i(h)$ for element i is

$$\int_{-h}^{h} f(x)dx - R_i(h) = -\frac{h^3}{24}f^{(2)}(\xi),$$

and the global error reads

$$\int_a^b f(x)dx - R_h(f) = -\frac{b-a}{24}h^2 f^{(2)}(\xi),$$

where R_h is the result obtained with rectangular rule and $\xi \in [a,b]$.

Instead of using the above linear two-point approximations for f, we could use the three-point formula for the derivatives. This means that we will choose formulae based on function values which lie symmetrically around the point where we preform the Taylor expansion. It means also that we are approximating our function with a second-order polynomial $f(x) = a + bx + cx^2$. The first and second derivatives are given by

$$\frac{f_h - f_{-h}}{2h} = f'_0 + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j},$$

and

$$\frac{f_h - 2f_0 + f_{-h}}{h^2} = f''_0 + 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j},$$

and we note that in both cases the error goes like $O(h^{2j})$. With the latter two expressions we can now approximate the function f as

$$f(x) = f_0 + \frac{f_h - f_{-h}}{2h} x + \frac{f_h - 2f_0 + f_{-h}}{2h^2} x^2 + O(x^3).$$

Inserting this formula in the integral of Eq. (7.2) we obtain

$$\int_{-h}^{+h} f(x)dx = \frac{h}{3}\left(f_h + 4f_0 + f_{-h}\right) + O(h^5),$$

which is Simpson's rule. Note that the improved accuracy in the evaluation of the derivatives gives a better error approximation, $O(h^5)$ vs. $O(h^3)$. But this is just the *local error approximation*. Using Simpson's rule we can easily compute the integral of Eq. (7.1) to be

$$I = \int_a^b f(x)dx = \frac{h}{3}\left(f(a) + 4f(a+h) + 2f(a+2h) + \cdots + 4f(b-h) + f_b\right), \tag{7.6}$$

with a global error which goes like $O(h^4)$. More formal expressions for the local and global errors are for the local error

$$\int_a^b f(x)dx - \frac{b-a}{6}\left[f(a) + 4f((a+b)/2) + f(b)\right] = -\frac{h^5}{90} f^{(4)}(\xi),$$

and for the global error

$$\int_a^b f(x)dx - S_h(f) = -\frac{b-a}{180} h^4 f^{(4)}(\xi).$$

with $\xi \in [a,b]$ and S_h the results obtained with Simpson's method. The method can easily be implemented numerically through the following simple algorithm

- Choose the number of mesh points and fix the step.
- calculate $f(a)$ and $f(b)$
- Perform a loop over $n = 1$ to $n-1$ ($f(a)$ and $f(b)$ are known) and sum up the terms $4f(a+h) + 2f(a+2h) + 4f(a+3h) + \cdots + 4f(b-h)$. Each step in the loop corresponds to a given value $a + nh$. Odd values of n give 4 as factor while even values yield 2 as factor.
- Multiply the final result by $\frac{h}{3}$.

In more general terms, what we have done here is to approximate a given function $f(x)$ with a polynomial of a certain degree. One can show that given $n+1$ distinct points $x_0,\dots,x_n \in [a,b]$ and $n+1$ values $y_0,\dots,y_n$ there exists a unique polynomial $P_n(x)$ with the property

$$p_n(x_j) = y_j \quad j = 0,\dots,n$$

In the Lagrange representation discussed in chapter 6, this interpolating polynomial is given by

$$P_n = \sum_{k=0}^{n} l_k y_k,$$

with the Lagrange factors

$$l_k(x) = \prod_{\substack{i=0\\ i\neq k}}^{n} \frac{x-x_i}{x_k-x_i} \quad k=0,\dots,n,$$

see for example the text of Kress [30] or Burlich and Stoer [40] for details. If we for example set $n=1$, we obtain

$$P_1(x) = y_0\frac{x-x_1}{x_0-x_1} + y_1\frac{x-x_0}{x_1-x_0} = \frac{y_1-y_0}{x_1-x_0}x - \frac{y_1x_0+y_0x_1}{x_1-x_0},$$

which we recognize as the equation for a straight line.

The polynomial interpolatory quadrature of order n with equidistant quadrature points $x_k = a+kh$ and step $h=(b-a)/n$ is called the Newton-Cotes quadrature formula of order n. The integral is

$$\int_a^b f(x)dx \approx \int_a^b p_n(x)dx = \sum_{k=0}^{n} w_k f(x_k)$$

with

$$w_k = h\frac{(-1)^{n-k}}{k!(n-k)!}\int_0^n \prod_{\substack{j=0\\ j\neq k}}^{n} (z-j)dz,$$

for $k=0,\dots,n$.

7.2.1 Romberg integration

7.3 Gaussian quadrature

The methods we have presented hitherto are taylored to problems where the mesh points x_i are equidistantly spaced, x_i differing from x_{i+1} by the step h. These methods are well suited to cases where the integrand may vary strongly over a certain region or if we integrate over the solution of a differential equation.

If however our integrand varies only slowly over a large interval, then the methods we have discussed may only slowly converge towards a chosen precision[1]. As an example,

$$I = \int_1^b x^{-2} f(x)dx,$$

[1]You could e.g., impose that the integral should not change as function of increasing mesh points beyond the sixth digit.

may converge very slowly to a given precision if b is large and/or $f(x)$ varies slowly as function of x at large values. One can obviously rewrite such an integral by changing variables to $t = 1/x$ resulting in

$$I = \int_{b^{-1}}^{1} f(t^{-1})dt,$$

which has a small integration range and hopefully the number of mesh points needed is not that large.

However there are cases where no trick may help, and where the time expenditure in evaluating an integral is of importance. For such cases, we would like to recommend methods based on Gaussian quadrature. Here one can catch at least two birds with a stone, namely, increased precision and fewer (less time) mesh points. But it is important that the integrand varies smoothly over the interval, else we have to revert to splitting the interval into many small subintervals and the gain achieved may be lost. The mathematical details behind the theory for Gaussian quadrature formulae is quite terse. If you however are interested in the derivation, we advice you to consult the text of Stoer and Bulirsch [3], see especially section 3.6. Here we limit ourselves to merely delineate the philosophy and show examples of practical applications.

The basic idea behind all integration methods is to approximate the integral

$$I = \int_{a}^{b} f(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i),$$

where ω and x are the weights and the chosen mesh points, respectively. In our previous discussion, these mesh points were fixed at the beginning, by choosing a given number of points N. The weigths ω resulted then from the integration method we applied. Simpson's rule, see Eq. (7.6) would give

$$\omega : \{h/3, 4h/3, 2h/3, 4h/3, \dots, 4h/3, h/3\},$$

for the weights, while the trapezoidal rule resulted in

$$\omega : \{h/2, h, h, \dots, h, h/2\}.$$

In general, an integration formula which is based on a Taylor series using N points, will integrate exactly a polynomial P of degree $N-1$. That is, the N weights ω_n can be chosen to satisfy N linear equations, see chapter 3 of Ref. [3]. A greater precision for a given amount of numerical work can be achieved if we are willing to give up the requirement of equally spaced integration points. In Gaussian quadrature (hereafter GQ), both the mesh points and the weights are to be determined. The points will not be equally spaced[2]. The theory behind GQ is to obtain an arbitrary weight ω through the use of so-called orthogonal polynomials. These polynomials are orthogonal in some interval say e.g., [-1,1]. Our points x_i are chosen in some optimal sense subject only to the constraint that they should lie in this interval. Together with the weights we have then $2N$ (N the number of points) parameters at our disposal.

Even though the integrand is not smooth, we could render it smooth by extracting from it the weight function of an orthogonal polynomial, i.e., we are rewriting

$$I = \int_{a}^{b} f(x)dx = \int_{a}^{b} W(x)g(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i), \tag{7.7}$$

[2]Typically, most points will be located near the origin, while few points are needed for large x values since the integrand is supposed to vary smoothly there. See below for an example.

where g is smooth and W is the weight function, which is to be associated with a given orthogonal polynomial.

The weight function W is non-negative in the integration interval $x \in [a, b]$ such that for any $n \geq 0$ $\int_a^b |x|^n W(x)dx$ is integrable. The naming weight function arises from the fact that it may be used to give more emphasis to one part of the interval than another. A quadrature formula

$$\int_a^b W(x)f(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i), \tag{7.8}$$

with N distinct quadrature points (mesh points) is a called a Gaussian quadrature formula if it integrates all polynomials $p \in P_{2N-1}$ exactly, that is

$$\int_a^b W(x)p(x)dx = \sum_{i=1}^{N} \omega_i p(x_i), \tag{7.9}$$

It is assumed that $W(x)$ is continuous and positive and that the integral

$$\int_a^b W(x)dx$$

exists. Note that the replacement of $f \to Wg$ is normally a better approximation due to the fact that we may isolate possible singularities of W and its derivatives at the endpoints of the interval.

The quadrature weights or just weights (not to be confused with the weight function) are positive and the sequence of Gaussian quadrature formulae is convergent if the sequence Q_N of quadrature formulae

$$Q_N(f) \to Q(f) = \int_a^b f(x)dx,$$

in the limit $n \to \infty$. Then we say that the sequence

$$Q_N(f) = \sum_{i=1}^{N} \omega_i^{(N)} f(x_i^{(N)}),$$

is convergent for all polynomials p, that is

$$Q_N(p) = Q(p)$$

if there exits a constant C such that

$$\sum_{i=1}^{N} |\omega_i^{(N)}| \leq C,$$

for all N which are natural numbers.

The error for the Gaussian quadrature formulae of order N is given by

$$\int_a^b W(x)f(x)dx - \sum_{k=1}^{N} w_k f(x_k) = \frac{f^{2N}(\xi)}{(2N)!} \int_a^b W(x)[q_N(x)]^2 dx$$

where q_N is the chosen orthogonal polynomial and ξ is a number in the interval $[a, b]$. We have assumed that $f \in C^{2N}[a, b]$, viz. the space of all real or complex $2N$ times continuously differentiable functions.

In physics there are several important orthogonal polynomials which arise from the solution of differential equations. These are Legendre, Hermite, Laguerre and Chebyshev polynomials. They have the following weight functions

Weight function	Interval	Polynomial
$W(x) = 1$	$x \in [-1, 1]$	Legendre
$W(x) = e^{-x^2}$	$-\infty \leq x \leq \infty$	Hermite
$W(x) = e^{-x}$	$0 \leq x \leq \infty$	Laguerre
$W(x) = 1/(\sqrt{1-x^2})$	$-1 \leq x \leq 1$	Chebyshev

The importance of the use of orthogonal polynomials in the evaluation of integrals can be summarized as follows.

- As stated above, methods based on Taylor series using N points will integrate exactly a polynomial P of degree $N-1$. If a function $f(x)$ can be approximated with a polynomial of degree $N-1$

$$f(x) \approx P_{N-1}(x),$$

with N mesh points we should be able to integrate exactly the polynomial P_{N-1}.

- Gaussian quadrature methods promise more than this. We can get a better polynomial approximation with order greater than N to $f(x)$ and still get away with only N mesh points. More precisely, we approximate

$$f(x) \approx P_{2N-1}(x),$$

and with only N mesh points these methods promise that

$$\int f(x)dx \approx \int P_{2N-1}(x)dx = \sum_{i=0}^{N-1} P_{2N-1}(x_i)\omega_i,$$

The reason why we can represent a function $f(x)$ with a polynomial of degree $2N-1$ is due to the fact that we have $2N$ equations, N for the mesh points and N for the weights.

The mesh points are the zeros of the chosen orthogonal polynomial of order N, and the weights are determined from the inverse of a matrix. An orthogonal polynomials of degree N defined in an interval $[a, b]$ has precisely N distinct zeros on the open interval (a, b).

Before we detail how to obtain mesh points and weights with orthogonal polynomials, let us revisit some features of orthogonal polynomials by specializing to Legendre polynomials. In the text below, we reserve hereafter the labelling L_N for a Legendre polynomial of order N, while P_N is an arbitrary polynomial of order N. These polynomials form then the basis for the Gauss-Legendre method.

7.3.1 Orthogonal polynomials, Legendre

The Legendre polynomials are the solutions of an important differential equation in physics, namely

$$C(1-x^2)P - m_l^2 P + (1-x^2)\frac{d}{dx}\left((1-x^2)\frac{dP}{dx}\right) = 0.$$

C is a constant. For $m_l = 0$ we obtain the Legendre polynomials as solutions, whereas $m_l \neq 0$ yields the so-called associated Legendre polynomials. This differential equation arises in for example the solution of the angular dependence of Schrödinger's equation with spherically symmetric potentials such as the Coulomb potential.

The corresponding polynomials P are

$$L_k(x) = \frac{1}{2^k k!}\frac{d^k}{dx^k}(x^2-1)^k \qquad k = 0, 1, 2, \ldots,$$

which, up to a factor, are the Legendre polynomials L_k. The latter fulfil the orthorgonality relation

$$\int_{-1}^{1} L_i(x)L_j(x)dx = \frac{2}{2i+1}\delta_{ij}, \tag{7.10}$$

and the recursion relation

$$(j+1)L_{j+1}(x) + jL_{j-1}(x) - (2j+1)xL_j(x) = 0. \tag{7.11}$$

It is common to choose the normalization condition

$$L_N(1) = 1.$$

With these equations we can determine a Legendre polynomial of arbitrary order with input polynomials of order $N-1$ and $N-2$.

As an example, consider the determination of L_0, L_1 and L_2. We have that

$$L_0(x) = c,$$

with c a constant. Using the normalization equation $L_0(1) = 1$ we get that

$$L_0(x) = 1.$$

For $L_1(x)$ we have the general expression

$$L_1(x) = a + bx,$$

and using the orthorgonality relation

$$\int_{-1}^{1} L_0(x)L_1(x)dx = 0,$$

we obtain $a = 0$ and with the condition $L_1(1) = 1$, we obtain $b = 1$, yielding

$$L_1(x) = x.$$

We can proceed in a similar fashion in order to determine the coefficients of L_2

$$L_2(x) = a + bx + cx^2,$$

using the orthorgonality relations

$$\int_{-1}^{1} L_0(x)L_2(x)dx = 0,$$

and

$$\int_{-1}^{1} L_1(x)L_2(x)dx = 0,$$

and the condition $L_2(1) = 1$ we would get

$$L_2(x) = \frac{1}{2}\left(3x^2 - 1\right). \tag{7.12}$$

We note that we have three equations to determine the three coefficients a, b and c.

Alternatively, we could have employed the recursion relation of Eq. (7.11), resulting in

$$2L_2(x) = 3xL_1(x) - L_0,$$

which leads to Eq. (7.12).

The orthogonality relation above is important in our discussion on how to obtain the weights and mesh points. Suppose we have an arbitrary polynomial Q_{N-1} of order $N-1$ and a Legendre polynomial $L_N(x)$ of order N. We could represent Q_{N-1} by the Legendre polynomials through

$$Q_{N-1}(x) = \sum_{k=0}^{N-1} \alpha_k L_k(x), \tag{7.13}$$

where α_k's are constants.

Using the orthogonality relation of Eq. (7.10) we see that

$$\int_{-1}^{1} L_N(x) Q_{N-1}(x) dx = \sum_{k=0}^{N-1} \int_{-1}^{1} L_N(x) \alpha_k L_k(x) dx = 0. \tag{7.14}$$

We will use this result in our construction of mesh points and weights in the next subsection.

In summary, the first few Legendre polynomials are

$$L_0(x) = 1,$$

$$L_1(x) = x,$$

$$L_2(x) = (3x^2 - 1)/2,$$

$$L_3(x) = (5x^3 - 3x)/2,$$

and

$$L_4(x) = (35x^4 - 30x^2 + 3)/8.$$

The following simple function implements the above recursion relation of Eq. (7.11). for computing Legendre polynomials of order N.

```
//  This function computes the Legendre polynomial of degree N

double legendre( int n, double x)
{
        double r, s, t;
        int m;
        r = 0; s = 1.;
        // Use recursion relation to generate p1 and p2
        for (m=0; m < n; m++ )
        {
           t = r; r = s;
           s = (2*m+1)*x*r - m*t;
           s /= (m+1);
         } // end of do loop
         return s;
}    // end of function legendre
```

The variable s represents $L_{j+1}(x)$, while r holds $L_j(x)$ and t the value $L_{j-1}(x)$.

7.3.2 Mesh points and weights with orthogonal polynomials

To understand how the weights and the mesh points are generated, we define first a polynomial of degree $2N-1$ (since we have $2N$ variables at hand, the mesh points and weights for N points). This polynomial can be represented through polynomial division by

$$P_{2N-1}(x) = L_N(x)P_{N-1}(x) + Q_{N-1}(x),$$

where $P_{N-1}(x)$ and $Q_{N-1}(x)$ are some polynomials of degree $N-1$ or less. The function $L_N(x)$ is a Legendre polynomial of order N.

Recall that we wanted to approximate an arbitrary function $f(x)$ with a polynomial P_{2N-1} in order to evaluate

$$\int_{-1}^{1} f(x)dx \approx \int_{-1}^{1} P_{2N-1}(x)dx,$$

we can use Eq. (7.14) to rewrite the above integral as

$$\int_{-1}^{1} P_{2N-1}(x)dx = \int_{-1}^{1} (L_N(x)P_{N-1}(x) + Q_{N-1}(x))dx = \int_{-1}^{1} Q_{N-1}(x)dx,$$

due to the orthogonality properties of the Legendre polynomials. We see that it suffices to evaluate the integral over $\int_{-1}^{1} Q_{N-1}(x)dx$ in order to evaluate $\int_{-1}^{1} P_{2N-1}(x)dx$. In addition, at the points x_k where L_N is zero, we have

$$P_{2N-1}(x_k) = Q_{N-1}(x_k) \qquad k = 0, 1, \ldots, N-1,$$

and we see that through these N points we can fully define $Q_{N-1}(x)$ and thereby the integral. Note that we have chosen to let the numbering of the points run from 0 to $N-1$. The reason for this choice is that we wish to have the same numbering as the order of a polynomial of degree $N-1$. This numbering will be useful below when we introduce the matrix elements which define the integration weights w_i.

We develope then $Q_{N-1}(x)$ in terms of Legendre polynomials, as done in Eq. (7.13),

$$Q_{N-1}(x) = \sum_{i=0}^{N-1} \alpha_i L_i(x). \tag{7.15}$$

Using the orthogonality property of the Legendre polynomials we have

$$\int_{-1}^{1} Q_{N-1}(x)dx = \sum_{i=0}^{N-1} \alpha_i \int_{-1}^{1} L_0(x)L_i(x)dx = 2\alpha_0,$$

where we have just inserted $L_0(x) = 1$! Instead of an integration problem we need now to define the coefficient α_0. Since we know the values of Q_{N-1} at the zeros of L_N, we may rewrite Eq. (7.15) as

$$Q_{N-1}(x_k) = \sum_{i=0}^{N-1} \alpha_i L_i(x_k) = \sum_{i=0}^{N-1} \alpha_i L_{ik} \qquad k = 0, 1, \ldots, N-1. \tag{7.16}$$

Since the Legendre polynomials are linearly independent of each other, none of the columns in the matrix L_{ik} are linear combinations of the others. This means that the matrix L_{ik} has an inverse with the properties

$$\mathbf{L}^{-1}\mathbf{L} = \mathbf{I}.$$

Multiplying both sides of Eq. (7.16) with $\sum_{j=0}^{N-1} L_{ji}^{-1}$ results in

$$\sum_{i=0}^{N-1} (L^{-1})_{ki} Q_{N-1}(x_i) = \alpha_k, \tag{7.17}$$

and since

$$\int_{-1}^{1} P_{2N-1}(x)dx = \int_{-1}^{1} Q_{N-1}(x)dx = 2\alpha_0 = 2\sum_{i=0}^{N-1}(L^{-1})_{0i} P_{2N-1}(x_i),$$

we see that if we identify the weights with $2(L^{-1})_{0i}$, where the points x_i are the zeros of L, we have an integration formula of the type

$$\int_{-1}^{1} P_{2N-1}(x)dx = \sum_{i=0}^{N-1} \omega_i P_{2N-1}(x_i)$$

and if our function $f(x)$ can be approximated by a polynomial P of degree $2N-1$, we have finally that

$$\int_{-1}^{1} f(x)dx \approx \int_{-1}^{1} P_{2N-1}(x)dx = \sum_{i=0}^{N-1} \omega_i P_{2N-1}(x_i).$$

In summary, the mesh points x_i are defined by the zeros of L while the weights are given by $2(L^{-1})_{0i}$.

7.3.3 Application to the case $N = 2$

Let us visualize the above formal results for the case $N = 2$. This means that we can approximate a function $f(x)$ with a polynomial $P_3(x)$ of order $2N - 1 = 3$.

The mesh points are the zeros of $L_2(x) = 1/2(3x^2 - 1)$. These points are $x_0 = -1/\sqrt{3}$ and $x_1 = 1/\sqrt{3}$.

Specializing Eq. (7.16)

$$Q_{N-1}(x_k) = \sum_{i=0}^{N-1} \alpha_i L_i(x_k) \qquad k = 0, 1, \ldots, N-1.$$

to $N = 2$ yields

$$Q_1(x_0) = \alpha_0 - \alpha_1 \frac{1}{\sqrt{3}},$$

and

$$Q_1(x_1) = \alpha_0 + \alpha_1 \frac{1}{\sqrt{3}},$$

since $L_0(x = \pm 1/\sqrt{3}) = 1$ and $L_1(x = \pm 1/\sqrt{3}) = \pm 1/\sqrt{3}$.

The matrix L_{ik} defined in Eq. (7.16) is then

$$L_{ik} = \begin{pmatrix} 1 & -\frac{1}{\sqrt{3}} \\ 1 & \frac{1}{\sqrt{3}} \end{pmatrix},$$

with an inverse given by

$$(L)_{ik}^{-1} = \frac{\sqrt{3}}{2} \begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ -1 & 1 \end{pmatrix}.$$

The weights are given by the matrix elements $2(L_{0k})^{-1}$. We have thence $\omega_0 = 1$ and $\omega_1 = 1$.

Obviously, there is no problem in changing the numbering of the matrix elements $i, k = 0, 1, 2, \ldots, N-1$ to $i, k = 1, 2, \ldots, N$. We have chosen to start from zero, since we deal with polynomials of degree $N-1$.

Summarizing, for Legendre polynomials with $N = 2$ we have weights

$$\omega : \{1, 1\},$$

and mesh points

$$x : \left\{-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right\}.$$

If we wish to integrate

$$\int_{-1}^{1} f(x)dx,$$

with $f(x) = x^2$, we approximate

$$I = \int_{-1}^{1} x^2 dx \approx \sum_{i=0}^{N-1} \omega_i x_i^2.$$

The exact answer is $2/3$. Using $N = 2$ with the above two weights and mesh points we get

$$I = \int_{-1}^{1} x^2 dx = \sum_{i=0}^{1} \omega_i x_i^2 = \frac{1}{3} + \frac{1}{3} = \frac{2}{3},$$

the exact answer!

If we were to emply the trapezoidal rule we would get

$$I = \int_{-1}^{1} x^2 dx = \frac{b-a}{2}\left((a)^2 + (b)^2\right)/2 = \frac{1-(-1)}{2}\left((-1)^2 + (1)^2\right)/2 = 1!$$

With just two points we can calculate exactly the integral for a second-order polynomial since our methods approximates the exact function with higher order polynomial. How many points do you need with the trapezoidal rule in order to achieve a similar accuracy?

7.3.4 General integration intervals for Gauss-Legendre

Note that the Gauss-Legendre method is not limited to an interval [-1,1], since we can always through a change of variable

$$t = \frac{b-a}{2}x + \frac{b+a}{2},$$

rewrite the integral for an interval [a,b]

$$\int_{a}^{b} f(t)dt = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{(b-a)x}{2} + \frac{b+a}{2}\right) dx.$$

If we have an integral on the form

$$\int_{0}^{\infty} f(t)dt,$$

we can choose new mesh points and weights by using the mapping

$$\tilde{x}_i = tan\left\{\frac{\pi}{4}(1+x_i)\right\},$$

and

$$\tilde{\omega}_i = \frac{\pi}{4}\frac{\omega_i}{cos^2\left(\frac{\pi}{4}(1+x_i)\right)},$$

where x_i and ω_i are the original mesh points and weights in the interval $[-1,1]$, while $\tilde{x}_i$ and $\tilde{\omega}_i$ are the new mesh points and weights for the interval $[0,\infty]$.

To see that this is correct by inserting the the value of $x_i = -1$ (the lower end of the interval $[-1,1]$) into the expression for $\tilde{x}_i$. That gives $\tilde{x}_i = 0$, the lower end of the interval $[0,\infty]$. For $x_i = 1$, we obtain $\tilde{x}_i = \infty$. To check that the new weights are correct, recall that the weights should correspond to the derivative of the mesh points. Try to convince yourself that the above expression fulfils this condition.

7.3.5 Other orthogonal polynomials

Laguerre polynomials

If we are able to rewrite our integral of Eq. (7.7) with a weight function $W(x) = x^\alpha e^{-x}$ with integration limits $[0,\infty]$, we could then use the Laguerre polynomials. The polynomials form then the basis for the Gauss-Laguerre method which can be applied to integrals of the form

$$I = \int_0^\infty f(x)dx = \int_0^\infty x^\alpha e^{-x} g(x)dx.$$

These polynomials arise from the solution of the differential equation

$$\left(\frac{d^2}{dx^2} - \frac{d}{dx} + \frac{\lambda}{x} - \frac{l(l+1)}{x^2}\right)\mathcal{L}(x) = 0,$$

where l is an integer $l \geq 0$ and λ a constant. This equation arises e.g., from the solution of the radial Schrödinger equation with a centrally symmetric potential such as the Coulomb potential. The first few polynomials are

$$\mathcal{L}_0(x) = 1,$$

$$\mathcal{L}_1(x) = 1 - x,$$

$$\mathcal{L}_2(x) = 2 - 4x + x^2,$$

$$\mathcal{L}_3(x) = 6 - 18x + 9x^2 - x^3,$$

and

$$\mathcal{L}_4(x) = x^4 - 16x^3 + 72x^2 - 96x + 24.$$

They fulfil the orthorgonality relation

$$\int_{-\infty}^\infty e^{-x}\mathcal{L}_n(x)^2 dx = 1,$$

and the recursion relation

$$(n+1)\mathcal{L}_{n+1}(x) = (2n+1-x)\mathcal{L}_n(x) - n\mathcal{L}_{n-1}(x).$$

Hermite polynomials

In a similar way, for an integral which goes like

$$I = \int_{-\infty}^{\infty} f(x)dx = \int_{-\infty}^{\infty} e^{-x^2} g(x)dx.$$

we could use the Hermite polynomials in order to extract weights and mesh points. The Hermite polynomials are the solutions of the following differential equation

$$\frac{d^2H(x)}{dx^2} - 2x\frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0. \tag{7.18}$$

A typical example is again the solution of Schrödinger's equation, but this time with a harmonic oscillator potential. The first few polynomials are

$$H_0(x) = 1,$$
$$H_1(x) = 2x,$$
$$H_2(x) = 4x^2 - 2,$$
$$H_3(x) = 8x^3 - 12,$$

and

$$H_4(x) = 16x^4 - 48x^2 + 12.$$

They fulfil the orthorgonality relation

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x)^2 dx = 2^n n! \sqrt{\pi},$$

and the recursion relation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x).$$

7.3.6 Applications to selected integrals

Before we proceed with some selected applications, it is important to keep in mind that since the mesh points are not evenly distributed, a careful analysis of the behavior of the integrand as function of x and the location of mesh points is mandatory. To give you an example, in the Table below we show the mesh points and weights for the integration interval [0,100] for $N = 10$ points obtained by the Gauss-Legendre method. Clearly, if your function oscillates strongly in any subinterval, this approach needs to be refined, either by choosing more points or by choosing other integration methods. Note also that for integration intervals like for example $x \in [0, \infty]$, the Gauss-Legendre method places more points at the beginning of the integration interval. If your integrand varies slowly for large values of x, then this method may be appropriate.

Let us here compare three methods for integrating, namely the trapezoidal rule, Simpson's method and the Gauss-Legendre approach. We choose two functions to integrate:

$$\int_1^{100} \frac{\exp(-x)}{x} dx,$$

and

$$\int_0^3 \frac{1}{2 + x^2} dx.$$

A program example which uses the trapezoidal rule, Simpson's rule and the Gauss-Legendre method is included here. The corresponding Fortran program is located as programs/chapter7/program1.f90.

Table 7.1: Mesh points and weights for the integration interval [0,100] with $N = 10$ using the Gauss-Legendre method.

i	x_i	ω_i
1	1.305	3.334
2	6.747	7.473
3	16.030	10.954
4	28.330	13.463
5	42.556	14.776
6	57.444	14.776
7	71.670	13.463
8	83.970	10.954
9	93.253	7.473
10	98.695	3.334

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter07/cpp/program1.cpp

```
#include <iostream>
#include "lib.h"
using namespace std;
//      Here we define various functions called by the main program
//      this function defines the function to integrate
double int_function(double x);
//    Main function begins here
int main()
{
     int n;
     double a, b;
     cout << "Read in the number of integration points" << endl;
     cin >> n;
     cout << "Read in integration limits" << endl;
     cin >> a >> b;
//   reserve space in memory for vectors containing the mesh points
//   weights and function values for the use of the gauss-legendre
//   method
     double *x = new double [n];
     double *w = new double [n];
//   set up the mesh points and weights
     gauleg(a, b,x,w, n);
//   evaluate the integral with the Gauss-Legendre method
//   Note that we initialize the sum
     double int_gauss = 0.;
     for ( int i = 0;  i < n; i++){
        int_gauss+=w[i]*int_function(x[i]);
     }
//    final output
      cout << "Trapez-rule = " << trapezoidal_rule(a, b,n, int_function)
           << endl;
      cout << "Simpson's rule = " << simpson(a, b,n, int_function)
           << endl;
```

```
      cout << "Gaussian quad = " << int_gauss << endl;
      delete [] x;
      delete [] w;
      return 0;
}  // end of main program
//  this function defines the function to integrate
double int_function(double x)
{
  double value = 4./(1.+x*x);
  return value;
} // end of function to evaluate
```

To be noted in this program is that we can transfer the name of a given function to integrate. In Table 7.2 we show the results for the first integral using various mesh points, while Table 7.3 displays the corresponding results obtained with the second integral. We note here that, since the area over where we integrate is rather large and the integrand goes slowly to zero for large values of x, both the trapezoidal rule and Simpson's method need quite many points in order to approach the Gauss-Legendre method. This integrand demonstrates clearly the strength of the Gauss-Legendre method (and other GQ methods as well), viz., few points are needed in order to achieve a very high precision.

Table 7.2: Results for $\int_1^{100} \exp{(-x)}/x dx$ using three different methods as functions of the number of mesh points N.

N	Trapez	Simpson	Gauss-Legendre
10	1.821020	1.214025	0.1460448
20	0.912678	0.609897	0.2178091
40	0.478456	0.333714	0.2193834
100	0.273724	0.231290	0.2193839
1000	0.219984	0.219387	0.2193839

The second Table however shows that for smaller integration intervals, both the trapezoidal rule and Simpson's method compare well with the results obtained with the Gauss-Legendre approach.

Table 7.3: Results for $\int_0^3 1/(2+x^2)dx$ using three different methods as functions of the number of mesh points N.

N	Trapez	Simpson	Gauss-Legendre
10	0.798861	0.799231	0.799233
20	0.799140	0.799233	0.799233
40	0.799209	0.799233	0.799233
100	0.799229	0.799233	0.799233
1000	0.799233	0.799233	0.799233

7.4 Treatment of singular Integrals

So-called principal value (PV) integrals are often employed in physics, from Green's functions for scattering to dispersion relations. Dispersion relations are often related to measurable quantities and provide important consistency checks in atomic, nuclear and particle physics. A PV integral is defined as

$$I(x) = P\int_a^b dt\frac{f(t)}{t-x} = \lim_{\epsilon\to 0^+}\left[\int_a^{x-\epsilon} dt\frac{f(t)}{t-x} + \int_{x+\epsilon}^b dt\frac{f(t)}{t-x}\right],$$

and arises in applications of Cauchy's residue theorem when the pole x lies on the real axis within the interval of integration $[a,b]$.

An important assumption is that the function $f(t)$ is continuous on the interval of integration.

In case $f(t)$ is an analytic expression or it has an analytic continuation in the complex plane, it may be possible to obtain an expression on closed form for the above integral.

However, the situation which we are often confronted with is that $f(t)$ is only known at some points t_i with corresponding values $f(t_i)$. In order to obtain $I(x)$ we need to resort to a numerical evaluation.

To evaluate such an integral, let us first rewrite it as

$$P\int_a^b dt\frac{f(t)}{t-x} = \int_a^{x-\Delta} dt\frac{f(t)}{t-x} + \int_{x+\Delta}^b dt\frac{f(t)}{t-x} + P\int_{x-\Delta}^{x+\Delta} dt\frac{f(t)}{t-x},$$

where we have isolated the principal value part in the last integral.

Defining a new variable $u = t-x$, we can rewrite the principal value integral as

$$I_\Delta(x) = P\int_{-\Delta}^{+\Delta} du\frac{f(u+x)}{u}. \tag{7.19}$$

One possibility is to Taylor expand $f(u+x)$ around $u=0$, and compute derivatives to a certain order as we did for the Trapezoidal rule or Simpson's rule. Since all terms with even powers of u in the Taylor expansion dissapear, we have that

$$I_\Delta(x) \approx \sum_{n=0}^{N_{max}} f^{(2n+1)}(x)\frac{\Delta^{2n+1}}{(2n+1)(2n+1)!}.$$

To evaluate higher-order derivatives may be both time consuming and delicate from a numerical point of view, since there is always the risk of loosing precision when calculating derivatives numerically. Unless we have an analytic expression for $f(u+x)$ and can evaluate the derivatives in a closed form, the above approach is not the preferred one.

Rather, we show here how to use the Gauss-Legendre method to compute Eq. (7.19). Let us first introduce a new variable $s = u/\Delta$ and rewrite Eq. (7.19) as

$$I_\Delta(x) = P\int_{-1}^{+1} ds\frac{f(\Delta s+x)}{s}. \tag{7.20}$$

The integration limits are now from -1 to 1, as for the Legendre polynomials. The principal value in Eq. (7.20) is however rather tricky to evaluate numerically, mainly since computers have limited precision. We will here use a subtraction trick often used when dealing with singular integrals in numerical calculations. We introduce first the calculus relation

$$\int_{-1}^{+1}\frac{ds}{s} = 0.$$

It means that the curve $1/(s)$ has equal and opposite areas on both sides of the singular point $s = 0$.

If we then note that $f(x)$ is just a constant, we have also

$$f(x)\int_{-1}^{+1}\frac{ds}{s} = \int_{-1}^{+1} f(x)\frac{ds}{s} = 0.$$

Subtracting this equation from Eq. (7.20) yields

$$I_\Delta(x) = P\int_{-1}^{+1} ds\frac{f(\Delta s + x)}{s} = \int_{-1}^{+1} ds\frac{f(\Delta s + x) - f(x)}{s}, \tag{7.21}$$

and the integrand is now longer singular since we have that $\lim_{s\to x}(f(s+x) - f(x)) = 0$ and for the particular case $s = 0$ the integrand is now finite.

Eq. (7.21) is now rewritten using the Gauss-Legendre method resulting in

$$\int_{-1}^{+1} ds\frac{f(\Delta s + x) - f(x)}{s} = \sum_{i=1}^{N}\omega_i\frac{f(\Delta s_i + x) - f(x)}{s_i}, \tag{7.22}$$

where s_i are the mesh points (N in total) and ω_i are the weights.

In the selection of mesh points for a PV integral, it is important to use an even number of points, since an odd number of mesh points always picks $s_i = 0$ as one of the mesh points. The sum in Eq. (7.22) will then diverge.

Let us apply this method to the integral

$$I(x) = P\int_{-1}^{+1} dt\frac{e^t}{t}. \tag{7.23}$$

The integrand diverges at $x = t = 0$. We rewrite it using Eq. (7.21) as

$$P\int_{-1}^{+1} dt\frac{e^t}{t} = \int_{-1}^{+1}\frac{e^t - 1}{t}, \tag{7.24}$$

since $e^x = e^0 = 1$. With Eq. (7.22) we have then

$$\int_{-1}^{+1}\frac{e^t - 1}{t} \approx \sum_{i=1}^{N}\omega_i\frac{e^{t_i} - 1}{t_i}. \tag{7.25}$$

The exact results is 2.11450175075..... With just two mesh points we recall from the previous subsection that $\omega_1 = \omega_2 = 1$ and that the mesh points are the zeros of $L_2(x)$, namely $x_1 = -1/\sqrt{3}$ and $x_2 = 1/\sqrt{3}$. Setting $N = 2$ and inserting these values in the last equation gives

$$I_2(x = 0) = \sqrt{3}\left(e^{1/\sqrt{3}} - e^{-1/\sqrt{3}}\right) = 2.1129772845.$$

With six mesh points we get even the exact result to the tenth digit

$$I_6(x = 0) = 2.11450175075!$$

We can repeat the above subtraction trick for more complicated integrands. First we modify the integration limits to $\pm\infty$ and use the fact that

$$\int_{-\infty}^{\infty}\frac{dk}{k - k_0} = 0.$$

It means that the curve $1/(k-k_0)$ has equal and opposite areas on both sides of the singular point k_0. If we break the integral into one over positive k and one over negative k, a change of variable $k \to -k$ allows us to rewrite the last equation as

$$\int_0^\infty \frac{dk}{k^2-k_0^2} = 0.$$

We can use this to express a principal values integral as

$$\mathcal{P}\int_0^\infty \frac{f(k)dk}{k^2-k_0^2} = \int_0^\infty \frac{(f(k)-f(k_0))dk}{k^2-k_0^2}, \tag{7.26}$$

where the right-hand side is no longer singular at $k=k_0$, it is proportional to the derivative df/dk, and can be evaluated numerically as any other integral.

Such a trick is often used when evaluating integral equations, as discussed in the next section.

7.5 Adaptive quadrature methods

To be finalized end fall 2008

7.6 Multi-dimensional integrals

To be finalized end fall 2008

7.7 Parallel computing

We end this chapter by discussing modern supercomputing concepts like parallel computing. In particular, we will introduce you to the usage of the Message Passing Interface (MPI) library. MPI is a library, not a programming language. It specifies the names, calling sequences and results of functions or subroutines to be called from C++ or Fortran programs, and the classes and methods that make up the MPI C++ library. The programs that users write in Fortran or C++ are compiled with ordinary compilers and linked with the MPI library. MPI programs should be able to run on all possible machines and run all MPI implementetations without change. An excellent reference is the text by Karniadakis and Kirby II [17].

7.7.1 Brief survey of supercomputing concepts and terminologies

Since many discoveries in science are nowadays obtained via large-scale simulations, there is an everlasting wish and need to do larger simulations using shorter computer time. The development of the capacity for single-processor computers (even with increased processor speed and memory) can hardly keep up with the pace of scientific computing. The solution to the needs of the scientific computing and high-performance computing (HPC) communities has therefore been parallel computing.

The basic ideas of parallel computing is that multiple processors are involved to solve a global problem. The essence is to divide the entire computation evenly among collaborative processors.

Today's supercomputers are parallel machines and can achieve peak performances almost up to 10^{15} floating point operations per second, so-called peta-scale computers, see for example the list over the world's top 500 supercomputers at `www.top500.org`. This list gets updated twice per year and sets up the ranking according to a given supercomputer's performance on a benchmark code from the LINPACK library. The benchmark solves a set of linear equations using the best software for a given platform.

To understand the basic philosophy, it is useful to have a rough picture of how to classify different hardware models. We distinguish betwen three major groups, (i) conventional single-processor computers, normally called SISD (single-instruction-single-data) machines, (ii) so-called SIMD machines (single-instruction-multiple-data), which incorporate the idea of parallel processing using a large number of processing units to execute the same instruction on different data and finally (iii) modern parallel computers, so-called MIMD (multiple-instruction- multiple-data) machines that can execute different instruction streams in parallel on different data. On a MIMD machine the different parallel processing units perform operations independently of each others, only subject to synchronization via a given message passing interface at specified time intervals. MIMD machines are the dominating ones among present supercomputers, and we distinguish between two types of MIMD computers, namely shared memory machines and distributed memory machines. In shared memory systems the central processing units (CPU) share the same address space. Any CPU can access any data in the global memory. In distributed memory systems each CPU has its own memory. The CPUs are connected by some network and may exchange messages. A recent trend are so-called ccNUMA (cache-coherent-non-uniform-memory-access) systems which are clusters of SMP (symmetric multi-processing) machines and have a virtual shared memory.

Distributed memory machines, in particular those based on PC clusters, are nowadays the most widely used and cost-effective, although farms of PC clusters require large infrastuctures and yield additional expenses for cooling. PC clusters with Linux as operating systems are easy to setup and offer several advantages, since they are built from standard commodity hardware with the open source software (Linux) infrastructure. The designer can improve performance proportionally with added machines. The commodity hardware can be any of a number of mass-market, stand-alone compute nodes as simple as two networked computers each running Linux and sharing a file system or as complex as thousands of nodes with a high-speed, low-latency network. In addition to the increased speed of present individual processors (and most machines come today with dual cores or four cores, so-called quad-cores) the position of such commodity supercomputers has been strenghtened by the fact that a library like MPI has made parallel computing portable and easy. Although there are several implementations, they share the same core commands. Message-passing is a mature programming paradigm and widely accepted. It often provides an efficient match to the hardware.

7.7.2 Parallelism

When we discuss parallelism, it is common to subdivide different algorithms in three major groups.

– **Task parallelism**:the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulations and numerical integration are examples of possible applications. Since there is more or less no communication between different processors, task parallelism results in almost a perfect mathematical parallelism and is commonly dubbed embarassingly parallel (EP). The examples in this chapter fall under that category. The use of the MPI library is then limited to some few function calls and the programming is normally very simple.

– **Data parallelism**: use of multiple threads (e.g., one thread per processor) to dissect loops over arrays etc. This paradigm requires a single memory address space. Communication and synchronization between the processors are often hidden, and it is thus easy to program. However, the user surrenders much control to a specialized compiler. An example of data parallelism is compiler-based parallelization.

– **Message-passing**: all involved processors have an independent memory address space. The user is responsible for partitioning the data/work of a global problem and distributing the subproblems to the processors. Collaboration between processors is achieved by explicit message passing, which is used for data transfer plus synchronization.

  This paradigm is the most general one where the user has full control. Better parallel efficiency is usually achieved by explicit message passing. However, message-passing programming is more difficult. We will meet examples of this in connection with the solution eigenvalue problems in chapter 12 and of partial differential equations in chapter 15.

Before we proceed, let us look at two simple examples. We will also use these simple examples to define the speedup factor of a parallel computation. The first case is that of the additions of two vectors of dimension n,

$$\mathbf{z} = \alpha\mathbf{x} + \beta\mathbf{y},$$

where α and β are two real or complex numbers and $\mathbf{z}, \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ or $\in \mathbb{C}^n$. For every element we have thus

$$z_i = \alpha x_i + \beta y_i.$$

For every element z_i we have three floating point operations, two multiplications and one addition. If we assume that these operations take the same time Δt, then the total time spent by one processor is

$$T_1 = 3n\Delta t.$$

Suppose now that we have access to a parallel supercomputer with P processors. Assume also that $P \leq n$. We split then these addition and multiplication operations on every processor so that every processor performs $3n/P$ operations in total, resulting in a time $T_P = 3n\Delta t/P$ for every single processor. We also assume that the time needed to gather together these subsums is neglible

If we have a perfect parallelism, our speedup should be P, the number of processors available. We see that this is case by computing the relation between the time used in case of only one processor and the time used if we can access P processors. The speedup S_P is defined as

$$S_P = \frac{T_1}{T_P} = \frac{3n\Delta t}{3n\Delta t/P} = P,$$

a perfect speedup. As mentioned above, we call calculations that yield a perfect speedup for embarassingly parallel. The efficiency is defined as

$$\eta(P) = \frac{S(P)}{P}.$$

Our next example is that of the inner product of two vectors defined in Eq. (4.5),

$$c = \sum_{j=1}^{n} x_j y_j.$$

We assume again that $P \leq n$ and define $I = n/P$. Each processor is assigned with its own subset of local multiplications $c_P = \sum_p x_p y_p$, where p runs over all possible terms for processor P. As an example, assume that we have four processors. Then we have

$$c_1 = \sum_{j=1}^{n/4} x_j y_j, \qquad c_2 = \sum_{j=n/4+1}^{n/2} x_j y_j,$$

$$c_3 = \sum_{j=n/2+1}^{3n/4} x_j y_j, \qquad c_4 = \sum_{j=3n/4+1}^{n} x_j y_j.$$

We assume again that the time for every operation is Δt. If we have only one processor, the total time is $T_1 = (2n-1)\Delta t$. For four processors, we must now add the time needed to add $c_1 + c_2 + c_3 + c_4$, which is $3\Delta t$ (three additions) and the time needed to communicate the local result c_P to all other processors. This takes roughly $(P-1)\Delta t_c$, where Δt_c need not equal Δt.

The speedup for four processors becomes now

$$S_4 = \frac{T_1}{T_4} = \frac{(2n-1)\Delta t}{(n/2-1)\Delta t + 3\Delta t + 3\Delta t_c} = \frac{4n-2}{10+n},$$

if $\Delta t = \Delta t_c$. For $n = 100$, the speedup is $S_4 = 3.62 < 4$. For P processors the inner products yields a speedup

$$S_P = \frac{(2n-1)}{(2I+P-2)) + (P-1)\gamma},$$

with $\gamma = \Delta t_c/\Delta t$. Even with $\gamma = 0$, we see that the speedup is less than P.

The communication time Δt_c can reduce significantly the speedup. However, even if it is small, there are other factors as well which may reduce the efficiency η_p. For example, we may have an uneven load balance, meaning that not all the processors can perform useful work at all time, or that the number of processors doesn't match properly the size of the problem, or memory problems, or that a so-called startup time penalty known as latency may slow down the transfer of data. Crucial here is the rate at which messages are transferred

7.7.3 MPI with simple examples

When we want to parallelize a sequential algorithm, there are at least two aspects we need to consider, namely

- Identify the part(s) of a sequential algorithm that can be executed in parallel. This can be difficult.
- Distribute the global work and data among P processors. Stated differently, here you need to understand how you can get computers to run in parallel. From a practical point of view it means to implement parallel programming tools.

In this chapter we focus mainly on the last point. MPI is then a tool for writing programs to run in parallel, without needing to know much (in most cases nothing) about a given machine's architecture. MPI programs work on both shared memory and distributed memory machines. Furthermore, MPI is a very rich and complicated library. But it is not necessary to use all the features. The basic and most used functions have been optimized for most machine architectures

Before we proceed, we need to clarify some concepts, in particular the usage of the words process and processor. We refer to process as a logical unit which executes its own code, in an MIMD style. The processor is a physical device on which one or several processes are executed. The MPI standard uses the concept process consistently throughout its documentation. However, since we only consider situations where one processor is responsible for one process, we therefore use the two terms interchangeably in the discussion below, hopefully without creating ambiguities.

The six most important MPI functions are

- MPI_ Init - initiate an MPI computation
- MPI_Finalize - terminate the MPI computation and clean up
- MPI_Comm_size - how many processes participate in a given MPI computation.
- MPI_Comm_rank - which rank does a given process have. The rank is a number between 0 and size-1, the latter representing the total number of processes.
- MPI_Send - send a message to a particular process within an MPI computation
- MPI_Recv - receive a message from a particular process within an MPI computation.

The first MPI C++ program is a rewriting of our 'hello world' program (without the computation of the sine function) from chapter 2. We let every process write "Hello world" on the standard output.

`http://www.fys.uio.no/compphys/cp/programs/FYS3150/MPI/chapter07/program2.cpp`

```cpp
//     First C++ example of MPI Hello world
using namespace std;
#include <mpi.h>
#include <iostream>

int main (int nargs, char* args[])
{
     int numprocs, my_rank;
//    MPI initializations
     MPI_Init (&nargs, &args);
     MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
     MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
     cout << "Hello world, I have  rank " << my_rank << " out of " <<
         numprocs << endl;
//   End MPI
      MPI_Finalize ();
    return 0;
}
```

The corresponding Fortran program reads

```fortran
PROGRAM hello
   INCLUDE "mpif.h"
   INTEGER:: numprocs, my_rank, ierr

   CALL  MPI_INIT(ierr)
   CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
   CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
   WRITE(*,*)"Hello world, I've rank ",my_rank," out of ",numprocs
   CALL MPI_FINALIZE(ierr)

END PROGRAM hello
```

MPI is a message-passing library where all the routines have a corresponding C++-bindings[3] MPI_Command_name or Fortran-bindings (function names are by convention in uppercase, but can also be in lower case) MPI_COMMAND_NAME

To use the MPI library you must include header files which contain definitions and declarations that are needed by the MPI library routines. The following line must appear at the top of any source code file that will make an MPI call. For Fortran you must put in the beginning **INCLUDE** 'mpif.h' while for C++ you need to include the statement #**include** `"mpi.h"`. These header files contain the declarations of functions, variabels etc. needed by the MPI library.

The first MPI call must be MPI_INIT, which initializes the message passing routines, as defined in for example **INTEGER**:: ierr and **CALL** MPI_INIT(ierr) for the Fortran example. The variable ierr is an integer which holds an error code when the call returns. The value of ierr is however of little use since, by default, MPI aborts the program when it encounters an error. However, ierr must be included when MPI starts. For the C++ code we have the call to the function int MPI_Init(int *argc, char *argv) where argc and argv are arguments passed to main. MPI does not use these arguments in any way, however, and in MPI-2 implementations, NULL may be passed instead. When you have finished you must call the function MPI_Finalize. In Fortran you use the statement **CALL** MPI_FINALIZE(ierr) while for C++ we use the function int MPI_Finalize(void).

In addition to these calls, we have also included calls to so-called inquiry functions. There are two MPI calls that are usually made soon after initialization. They are for C++, MPI_COMM_SIZE ((MPI_COMM_WORLD, &numprocs) and **CALL** MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr) for Fortran. The function MPI_COMM_SIZE returns the number of tasks in a specified MPI communicator (comm when we refer to it in generic function calls below).

In MPI you can divide your total number of tasks into groups, called communicators. What does that mean? All MPI communication is associated with what one calls a communicator that describes a group of MPI processes with a name (context). The communicator designates a collection of processes which can communicate with each other. Every process is then identified by its rank. The rank is only meaningful within a particular communicator. A communicator is thus used as a mechanism to identify subsets of processes. MPI has the flexibility to allow you to define different types of communicators, see for example [16]. However, here we have used the communicator MPI_COMM_WORLD that contains all the MPI processes that are initiated when we run the program.

The variable numprocs refers to the number of processes we have at our disposal. The function MPI_COMM_RANK returns the rank (the name or identifier) of the tasks running the code. Each task (or processor) in a communicator is assigned a number my_rank from 0 to $\mathrm{numprocs} - 1$.

We are now ready to perform our first MPI calculations.

Running codes with MPI

To compile and load the above C++ code (after having understood how to use a local cluster), we can use the command

```
mpicxx -O2 -o program2.x  program2.cpp
```

and try to run with ten nodes using the command

```
mpiexec -np 10 ./program2.x
```

[3]The C++ bindings used in practice are the same as the C bindings, although reading older texts like [15–17] one finds extensive discussions on the difference between C and C++ bindings. Throughout this text we will use the C bindings.

If we wish to use the Fortran version we need to replace the C++ compiler statement mpicc with mpif90 or equivalent compilers. The name of the compiler is obviously system dependent. The command mpirun may be instead of mpiexec. Here you need to check your own system.

When we run MPI all processes use the same binary executable version of the code and all processes are running exactly the same code. The question is then how can we tell the difference between our parallel code running on a given number of processes and a serial code? There are two major distinctions you should keep in mind: (i) MPI lets each process have a particular rank to determine which instructions are run on a particular process and (ii) the processes communicate with each other in order to finalize a task. Even if all processes receive the same set of instructions, they will normally not execute the same instructions.We will exemplify this in connection with our integration example below.

The above example spits out the following output

```
Hello world, I've rank 0 out of 10 procs.
Hello world, I've rank 1 out of 10 procs.
Hello world, I've rank 4 out of 10 procs.
Hello world, I've rank 3 out of 10 procs.
Hello world, I've rank 9 out of 10 procs.
Hello world, I've rank 8 out of 10 procs.
Hello world, I've rank 2 out of 10 procs.
Hello world, I've rank 5 out of 10 procs.
Hello world, I've rank 7 out of 10 procs.
Hello world, I've rank 6 out of 10 procs.
```

The output to screen is not ordered since all processes are trying to write to screen simultaneously. It is then the operating system which opts for an ordering. If we wish to have an organized output, starting from the first process, we may rewrite our program as follows

http://www.fys.uio.no/compphys/cp/programs/FYS3150/MPI/chapter07/program3.cpp

```cpp
//    Second C++ example of MPI Hello world
using namespace std;
#include <mpi.h>
#include <iostream>

int main (int nargs, char* args[])
{
     int numprocs, my_rank, i;
//   MPI initializations
     MPI_Init (&nargs, &args);
     MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
     MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
     for (i = 0; i < numprocs; i++) {
       MPI_Barrier (MPI_COMM_WORLD);
       if (i == my_rank) {
         cout << "Hello world, I have  rank " << my_rank << " out of " <<
            numprocs << endl;
         fflush (stdout);
       }
     }
//   End MPI
      MPI_Finalize ();
```

```
    return 0;
}
```

Here we have used the MPI_Barrier function to ensure that every process has completed its set of instructions in a particular order. A barrier is a special collective operation that does not allow the processes to continue until all processes in the communicator (here MPI_COMM_WORLD) have called MPI_Barrier. The output is now

```
Hello world, I've rank 0 out of 10 procs.
Hello world, I've rank 1 out of 10 procs.
Hello world, I've rank 2 out of 10 procs.
Hello world, I've rank 3 out of 10 procs.
Hello world, I've rank 4 out of 10 procs.
Hello world, I've rank 5 out of 10 procs.
Hello world, I've rank 6 out of 10 procs.
Hello world, I've rank 7 out of 10 procs.
Hello world, I've rank 8 out of 10 procs.
Hello world, I've rank 9 out of 10 procs.
```

The barriers make sure that all processes have reached the same point in the code. Many of the collective operations like MPI_ALLREDUCE to be discussed later, have the same property; viz. no process can exit the operation until all processes have started. However, this is slightly more time-consuming since the processes synchronize between themselves as many times as there are processes. In the next Hello world example we use the send and receive functions in order to a have a synchronized action.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/MPI/chapter07/program4.cpp

```
//    Third C++ example of MPI Hello world
using namespace std;
#include <mpi.h>
#include <iostream>

int main (int nargs, char* args[])
{
     int numprocs, my_rank, flag;
//   MPI initializations
     MPI_Status status;
     MPI_Init (&nargs, &args);
     MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
     MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
     //   Send and Receive example
     if (my_rank > 0)
       MPI_Recv (&flag, 1, MPI_INT, my_rank-1, 100, MPI_COMM_WORLD, &status)
           ;
       cout << "Hello world, I have  rank " << my_rank << " out of " <<
           numprocs << endl;
     if (my_rank < numprocs-1)
         MPI_Send (&my_rank, 1, MPI_INT, my_rank+1, 100, MPI_COMM_WORLD);
//  End MPI
      MPI_Finalize ();
    return 0;
}
```

The basic sending of messages is given by the function MPI_SEND, which in C++ is defined as

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
     MPI_Comm comm)
```

while in Fortran we would call this function with the following parameters

```
CALL MPI_SEND(buf, count, MPI_TYPE, dest, tag, comm, ierr).
```

This single command allows the passing of any kind of variable, even a large array, to any group of tasks. The variable buf is the variable we wish to send while count is the number of variables we are passing. If we are passing only a single value, this should be 1. If we transfer an array, it is the overall size of the array. For example, if we want to send a 10 by 10 array, count would be $10 \times 10 = 100$ since we are actually passing 100 values.

We define the type of variable using MPI_TYPE in order to let MPI function know what to expect. The destination of the send is declared via the variable dest, which gives the ID number of the task we are sending the message to. The variable tag is a way for the receiver to verify that it is getting the message it expects. The message tag is an integer number that we can assign any value, normally a large number (larger than the expected number of processes). The communicator comm is the group ID of tasks that the message is going to. For complex programs, tasks may be divided into groups to speed up connections and transfers. In small programs, this will more than likely be in MPI_COMM_WORLD.

Furthermore, when an MPI routine is called, the Fortran or C++ data type which is passed must match the corresponding MPI integer constant. An integer is defined as MPI_INT in C++ and MPI_INTEGER in Fortran. A double precision real is MPI_DOUBLE in C++ and MPI_DOUBLE_PRECISION in Fortran and single precision real is MPI_FLOAT in C++ and MPI_REAL in Fortran. For further definitions of data types see chapter five of Ref. [16].

Once you have sent a message, you must receive it on another task. The function MPI_RECV is similar to the send call. In C++ we would define this as

```
int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source, int
    tag, MPI_Comm comm, MPI_Status *status )
```

while in Fortran we would use the call

```
CALL MPI_RECV(buf, count, MPI_TYPE, source, tag, comm, status, ierr)}.
```

The arguments that are different from those in MPI_SEND are buf which is the name of the variable where you will be storing the received data, source which replaces the destination in the send command. This is the return ID of the sender.

Finally, we have used MPI_Status status; where one can check if the receive was completed. The source or tag of a received message may not be known if wildcard values are used in the receive function. In C++, MPI Status is a structure that contains further information. One can obtain this information using

```
MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count)}
```

The output of this code is the same as the previous example, but now process 0 sends a message to process 1, which forwards it further to process 2, and so forth.

Armed with this wisdom, performed all hello world greetings, we are now ready for serious work.

7.7.4 Numerical integration with MPI

To integrate numerically with MPI we need to define how to send and receive data types. This means also that we need to specify which data types to send to MPI functions.

The program listed here integrates

$$\pi = \int_0^1 dx \frac{4}{1+x^2}$$

by simply adding up areas of rectangles according to the algorithm discussed in Eq. (7.5), rewritten here

$$I = \int_a^b f(x)dx \approx h\sum_{i=1}^{N} f(x_{i-1/2}),$$

where $f(x) = 4/(1+x^2)$. This is a brute force way of obtaining an integral but suffices to demonstrate our first application of MPI to mathematical problems. What we do is to subdivide the integration range $x \in [0,1]$ into n rectangles. Increasing n should obviously increase the precision of the result, as discussed in the beginning of this chapter. The parallel part proceeds by letting every process collect a part of the sum of the rectangles. At the end of the computation all the sums from the processes are summed up to give the final global sum. The program below serves thus as a simple example on how to integrate in parallel. We will refine it in the next examples and we will also add a simple example on how to implement the trapezoidal rule.

`http://www.fys.uio.no/compphys/cp/programs/FYS3150/MPI/chapter07/program5.cpp`

```
//    Reactangle rule and numerical integration using MPI send and Receive
using namespace std;
#include <mpi.h>
#include <iostream>

int main (int nargs, char* args[])
{
   int numprocs, my_rank, i, n = 1000;
   double local_sum, rectangle_sum, x, h;
   //   MPI initializations
   MPI_Init (&nargs, &args);
   MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   //   Read from screen a possible new vaue of n
   if (my_rank == 0 && nargs > 1) {
      n = atoi(args[1]);
   }
   h = 1.0/n;
   //  Broadcast n and h to all processes
   MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Bcast (&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   //  Every process sets up its contribution to the integral
   local_sum = 0.;
   for (i = my_rank; i < n; i += numprocs) {
     x = (i+0.5)*h;
     local_sum += 4.0/(1.0+x*x);
  }
   local_sum *= h;
   if (my_rank == 0) {
     MPI_Status status;
```

```
30         rectangle_sum = local_sum;
31         for (i=1; i < numprocs; i++) {
32           MPI_Recv(&local_sum,1,MPI_DOUBLE,MPI_ANY_SOURCE,500,
    MPI_COMM_WORLD,&status);
33           rectangle_sum += local_sum;
34         }
35         cout << "Result: " << rectangle_sum  << endl;
36       }  else
37         MPI_Send(&local_sum,1,MPI_DOUBLE,0,500,MPI_COMM_WORLD);
38       // End MPI
39       MPI_Finalize ();
40       return 0;
41   }
```

After the standard initializations with MPI such as

```
MPI_Init, MPI_Comm_size, MPI_Comm_rank,
```

MPI_COMM_WORLD contains now the number of processes defined by using for example

```
mpirun -np 10 ./prog.x
```

In line 4 we check if we have read in from screen the number of mesh points n. Note that in line 7 we fix $n = 1000$, however we have the possibility to run the code with a different number of mesh points as well. If my_rank equals zero, which correponds to the master node, then we read a new value of n if the number of arguments is larger than two. This can be done as follows when we run the code

```
mpiexec -np 10 ./prog.x  10000
```

In line 17 we define also the step length h. In lines 19 and 20 we use the broadcast function MPI_Bcast. We use this particular function because we want data on one processor (our master node) to be shared with all other processors. The broadcast function sends data to a group of processes. The MPI routine MPI_Bcast transfers data from one task to a group of others. The format for the call is in C++ given by the parameters of MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);. MPI_Bcast (&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD); in case of a double. The general structure of this function is `int MPI_Bcast( void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`. All processes call this function, both the process sending the data (with rank zero) and all the other processes in MPI_COMM_WORLD. Every process has now copies of n and h, the number of mesh points and the step length, respectively.

We transfer the addresses of n and h. The second argument represents the number of data sent. In case of a one-dimensional array, one needs to transfer the number of array elements. If you have an $n \times m$ matrix, you must transfer $n \times m$. We need also to specify whether the variable type we transfer is a non-numerical such as a logical or character variable or numerical of the integer, real or complex type.

We transfer also an integer variable `int root`. This variable specifies the process which has the original copy of the data. Since we fix this value to zero in the call in lines 19 and 20, it means that it is the master process which keeps this information. For Fortran, this function is called via the statement CALL MPI_BCAST(buff, count, MPI_TYPE, root, comm, ierr).

In lines 23-27, every process sums its own part of the final sum used by the rectangle rule. The receive statement collects the sums from all other processes in case my_rank == 0, else an MPI send is performed.

The above function is not very elegant. Furthermore, the MPI instructions can be simplified by using the functions MPI_Reduce or MPI_Allreduce. The first function takes information from all processes and sends the result of the MPI operation to one process only, typically the master node. If we use

MPI_Allreduce, the result is sent back to all processes, a feature which is useful when all nodes need the value of a joint operation. We limit ourselves to MPI_Reduce since it is only one process which will print out the final number of our calculation, The arguments to MPI_Allreduce are the same.

The MPI_Reduce function is defined as follows **int** MPI_Bcast(**void** *senddata, **void*** resultdata , **int** count, MPI_Datatype datatype, MPI_Op, **int** root, MPI_Comm comm). The two variables senddata and resultdata are obvious, besides the fact that one sends the address of the variable or the first element of an array. If they are arrays they need to have the same size. The variable count represents the total dimensionality, 1 in case of just one variable, while MPI_Datatype defines the type of variable which is sent and received. The new feature is MPI_Op. MPI_Op defines the type of operation we want to do. There are many options, see again Refs. [15–17] for full list. In our case, since we are summing the rectangle contributions from every process we define MPI_Op = MPI_SUM. If we have an array or matrix we can search for the largest og smallest element by sending either MPI_MAX or MPI_MIN. If we want the location as well (which array element) we simply transfer MPI_MAXLOC or MPI_MINOC. If we want the product we write MPI_PROD. MPI_Allreduce is defined as **int** MPI_Bcast(**void** *senddata, **void*** resultdata , **int** count, MPI_Datatype datatype, MPI_Op, MPI_Comm comm).

The function we list in the next example is the MPI extension of program1.cpp. The difference is that we employ only the trapezoidal rule. It is easy to extend this code to include gaussian quadrature or other methods.

It is also worth noting that every process has now its own starting and ending point. We read in the number of integration points n and the integration limits a and b. These are called a and b. They serve to define the local integration limits used by every process. The local integration limits are defined as local_a = a + my_rank *(b−a)/numprocs and local_b = a + (my_rank−1)*(b−a)/numprocs. These two variables are transfered to the method for the trapezoidal rule. These two methods return the local sum variable local_sum. MPI_Reduce collects all the local sums and returns the total sum, which is written out by the master node. The program below implements this. We have also added the possibility to measure the total time used by the code via the calls to MPI_Wtime.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/MPI/chapter07/program6.cpp

```
//     Trapezoidal rule and numerical integration using MPI with MPI_Reduce
using namespace std;
#include <mpi.h>
#include <iostream>

//      Here we define various functions called by the main program

double int_function(double );
double trapezoidal_rule(double , double , int , double (*)(double));

//    Main function begins here
int main (int nargs, char* args[])
{
  int n, local_n, numprocs, my_rank;
  double a, b, h, local_a, local_b, total_sum, local_sum;
  double  time_start, time_end, total_time;
  //  MPI initializations
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  time_start = MPI_Wtime();
```

```
  //  Fixed values for a, b and n
  a = 0.0 ; b = 1.0;  n = 1000;
  h = (b-a)/n;    // h is the same for all processes
  local_n = n/numprocs;  // make sure n > numprocs, else integer division
      gives zero
  // Length of each process' interval of
  // integration = local_n*h.
  local_a = a + my_rank*local_n*h;
  local_b = local_a + local_n*h;
  total_sum = 0.0;
  local_sum = trapezoidal_rule(local_a, local_b, local_n, &int_function);
  MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
     MPI_COMM_WORLD);
  time_end = MPI_Wtime();
  total_time = time_end-time_start;
  if ( my_rank == 0) {
    cout << "Trapezoidal rule = " <<  total_sum << endl;
    cout << "Time = " <<  total_time  << " on number of processors: "  <<
        numprocs  << endl;
  }
  // End MPI
  MPI_Finalize ();
  return 0;
}  // end of main program

//  this function defines the function to integrate
double int_function(double x)
{
  double value = 4./(1.+x*x);
  return value;
} // end of function to evaluate

//  this function defines the trapezoidal rule
double trapezoidal_rule(double a, double b, int n, double (*func)(double))
{
  double trapez_sum;
  double fa, fb, x, step;
  int    j;
  step=(b-a)/((double) n);
  fa=(*func)(a)/2. ;
  fb=(*func)(b)/2. ;
  trapez_sum=0.;
  for (j=1; j <= n-1; j++){
    x=j*step+a;
    trapez_sum+=(*func)(x);
  }
  trapez_sum=(trapez_sum+fb+fa)*step;
  return trapez_sum;
}  // end trapezoidal_rule
```

An obvious extension of this code is to read from file or screen the integration variables. One could also use the program library to call a particular integration method.

Chapter 8

Outline of the Monte-Carlo strategy

'Iacta Alea est', the die is cast, is what Julius Caesar is reported by Suetonius to have said on January 10, 49 BC as he led his army across the River Rubicon in Northern Italy. (Twelve Ceasars)*Gaius Suetonius*

8.1 Introduction

Monte Carlo methods are nowadays widely used, from the integration of multi-dimensional integrals to solving ab initio problems in chemistry, physics, medicine, biology, or even Dow-Jones forecasting. Computational finance is one of the novel fields where Monte Carlo methods have found a new field of applications, with financial engineering as an emerging field, see for example Refs. [41, 42]. Emerging fields like econophysics [43–45] are new examples of wide applications of Monte Carlo methods.

Numerical methods that are known as Monte Carlo methods can be loosely described as statistical simulation methods, where statistical simulation is defined in quite general terms to be any method that utilizes sequences of random numbers to perform the simulation. As mentioned in the introduction to this text, a central algorithm in Monte Carlo methods is the Metropolis algorithm, ranked as one of the top ten algorithms in the last century. We discuss this algorithm in the next chapter.

Statistical simulation methods may be contrasted to conventional numerical discretization methods, which typically are applied to ordinary or partial differential equations that describe some underlying physical or mathematical system. In many applications of Monte Carlo, the physical process is simulated directly, and there is no need to even write down the differential equations that describe the behavior of the system. The only requirement is that the physical (or mathematical) system be described by probability distribution functions (PDF's). Once the PDF's are known, the Monte Carlo simulation can proceed by random sampling from the PDF's. Many simulations are then performed (multiple "trials" or "histories") and the desired result is taken as an average over the number of observations (which may be a single observation or perhaps millions of observations). In many practical applications, one can predict the statistical error (the "variance") in this average result, and hence an estimate of the number of Monte Carlo trials that are needed to achieve a given error. If we assume that the physical system can be described by a given probability density function, then the Monte Carlo simulation can proceed by sampling from these PDF's, which necessitates a fast and effective way to generate random numbers uniformly distributed on the interval [0,1]. The outcomes of these random samplings, or trials, must be accumulated or tallied in an appropriate manner to produce the desired result, but the essential characteristic of Monte Carlo is the use of random sampling techniques (and perhaps other algebra to manipulate the outcomes) to arrive at a solution of the physical problem. In contrast, a conventional numerical solution approach would start with the mathematical model of the physical system, discretizing the differential equations and then solving a set of algebraic equations for the unknown state of the system. It should be kept in mind though, that this general description of Monte Carlo methods may not directly apply to some applications. It is natural to think that Monte Carlo methods are used to simulate random, or stochastic, processes, since these can be described by PDF's. However, this coupling is actually too restrictive because many Monte Carlo applications have no apparent stochastic content, such as the evaluation of a definite integral or the inversion of a system of linear equations. However, in these cases and others, one can pose the desired solution in terms of PDF's, and while this transformation may seem artificial, this step allows the system to be treated as a stochastic process for the purpose of simulation and hence Monte Carlo methods can be applied to simulate the system.

There are, at least four ingredients which are crucial in order to understand the basic Monte-Carlo strategy. These are

1. Random variables,
2. probability distribution functions (PDF),
3. moments of a PDF
4. and its pertinent variance σ.

All these topics will be discussed at length below. We feel however that a brief explanation may be appropriate in order to convey the strategy behind a Monte-Carlo calculation. Let us first demistify the somewhat obscure concept of a random variable. The example we choose is the classic one, the tossing of two dice, its outcome and the corresponding probability. In principle, we could imagine being able to determine exactly the motion of the two dice, and with given initial conditions determine the outcome of the tossing. Alas, we are not capable of pursuing this ideal scheme. However, it does not mean that we do not have a certain knowledge of the outcome. This partial knowledge is given by the probablity of obtaining a certain number when tossing the dice. To be more precise, the tossing of the dice yields the following possible values

$$[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. \tag{8.1}$$

These values are called the *domain*. To this domain we have the corresponding *probabilities*

$$[1/36, 2/36/3/36, 4/36, 5/36, 6/36, 5/36, 4/36, 3/36, 2/36, 1/36]. \tag{8.2}$$

The numbers in the domain are the outcomes of the physical process tossing the dice. *We cannot tell beforehand whether the outcome is 3 or 5 or any other number in this domain. This defines the randomness of the outcome, or unexpectedness or any other synonimous word which encompasses the uncertitude of the final outcome.* The only thing we can tell beforehand is that say the outcome 2 has a certain probability. If our favorite hobby is to spend an hour every evening throwing dice and registering the sequence of outcomes, we will note that the numbers in the above domain

$$[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], \tag{8.3}$$

appear in a random order. After 11 throws the results may look like

$$[10, 8, 6, 3, 6, 9, 11, 8, 12, 4, 5]. \tag{8.4}$$

Eleven new attempts may results in a totally different sequence of numbers and so forth. Repeating this exercise the next evening, will most likely never give you the same sequences. Thus, we say that the outcome of this hobby of ours is truely random.

Random variables are hence characterized by a domain which contains all possible values that the random value may take. This domain has a corresponding PDF.

To give you another example of possible random number spare time activities, consider the radioactive decay of an α-particle from a certain nucleus. Assume that you have at your disposal a Geiger-counter which registers every 10 ms whether an α-particle reaches the counter or not. If we record a hit as 1 and no observation as zero, and repeat this experiment for a long time, the outcome of the experiment is also truely random. We cannot form a specific pattern from the above observations. The only possibility to say something about the outcome is given by the PDF, which in this case is the well-known exponential function

$$\lambda \exp -(\lambda x), \tag{8.5}$$

with λ being proportional to the half-life of the given nucleus which decays.

Good texts on Monte Carlo methods are the monographs of Robert and Casella, Johnson and Fishman, see Refs. [46–48].

8.1.1 First illustration of the use of Monte-Carlo methods, crude integration

With this definition of a random variable and its associated PDF, we attempt now a clarification of the Monte-Carlo strategy by using the evaluation of an integral as our example.

In chapter 7 we discussed standard methods for evaluating an integral like

$$I = \int_0^1 f(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i), \tag{8.6}$$

where ω_i are the weights determined by the specific integration method (like Simpson's or Taylor's methods) with x_i the given mesh points. To give you a feeling of how we are to evaluate the above integral using Monte-Carlo, we employ here the crudest possible approach. Later on we will present slightly more refined approaches. This crude approach consists in setting all weights equal 1, $\omega_i = 1$. That corresponds to the rectangle method presented in Eq. (7.5), displayed again here

$$I = \int_a^b f(x)dx \approx h \sum_{i=1}^{N} f(x_{i-1/2}),$$

where $f(x_{i-1/2})$ is the midpoint value of f for a given value $x_{i-1/2}$. Setting $h = (b-a)/N$ where $b = 1$, $a = 0$, we can then rewrite the above integral as

$$I = \int_0^1 f(x)dx \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i), \tag{8.7}$$

where x_i are the midpoint values of x. Introducing the concept of the average of the function f for a given PDF $p(x)$ as

$$\langle f \rangle = \frac{1}{N} \sum_{i=1}^{N} f(x_i)p(x_i), \tag{8.8}$$

and identify $p(x)$ with the uniform distribution, viz $p(x) = 1$ when $x \in [0,1]$ and zero for all other values of x. The integral is is then the average of f over the interval $x \in [0,1]$

$$I = \int_0^1 f(x)dx \approx \langle f \rangle. \tag{8.9}$$

In addition to the average value $\langle f \rangle$ the other important quantity in a Monte-Carlo calculation is the variance σ^2 and the standard deviation σ. We define first the variance of the integral with f for a uniform distribution in the interval $x \in [0,1]$ to be

$$\sigma_f^2 = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - \langle f \rangle)^2 p(x_i), \tag{8.10}$$

amd inserting the uniform distribution this yields

$$\sigma_f^2 = \frac{1}{N}\sum_{i=1}^{N} f(x_i)^2 - \left(\frac{1}{N}\sum_{i=1}^{N} f(x_i)\right)^2, \tag{8.11}$$

or

$$\sigma_f^2 = \left(\langle f^2 \rangle - \langle f \rangle^2\right). \tag{8.12}$$

which is nothing but a measure of the extent to which f deviates from its average over the region of integration. The standard deviation is defined as the square root of the variance. If we consider the above results for a fixed value of N as a measurement, we could however recalculate the above average and variance for a series of different measurements. If each such measumerent produces a set of averages for the integral I denoted $\langle f \rangle_l$, we have for M measurements that the integral is given by

$$\langle I \rangle_M = \frac{1}{M}\sum_{l=1}^{M}\langle f \rangle_l. \tag{8.13}$$

We show in section 8.3 that if we can consider the probability of correlated events to be zero, we can rewrite the variance of these series of measurements as (equating $M = N$)

$$\sigma_N^2 \approx \frac{1}{N}\left(\langle f^2 \rangle - \langle f \rangle^2\right) = \frac{\sigma_f^2}{N}. \tag{8.14}$$

We note that the standard deviation is proportional with the inverse square root of the number of measurements

$$\sigma_N \sim \frac{1}{\sqrt{N}}. \tag{8.15}$$

The aim of Monte Carlo calculations is to have σ_N as small as possible after N samples. The results from one sample represents, since we are using concepts from statistics, a 'measurement'.

The scaling in the previous equation is clearly unfavorable compared even with the trapezoidal rule. In the previous chapter we saw that the trapezoidal rule carries a truncation error $O(h^2)$, with h the step length. In general, methods based on a Taylor expansion such as the trapezoidal rule or Simpson's rule have a truncation error which goes like $\sim O(h^k)$, with $k \geq 1$. Recalling that the step size is defined as $h = (b-a)/N$, we have an error which goes like $\sim N^{-k}$.

However, Monte Carlo integration is more efficient in higher dimensions. To see this, let us assume that our integration volume is a hypercube with side L and dimension d. This cube contains hence

$N = (L/h)^d$ points and therefore the error in the result scales as $N^{-k/d}$ for the traditional methods. The error in the Monte carlo integration is however independent of d and scales as $\sigma \sim 1/\sqrt{N}$, always! Comparing this error with that of the traditional methods, shows that Monte Carlo integration is more efficient than an order-k algorithm when $d > 2k$. In order to expose this, consider the definition of the quantum mechanical energy of a system consisting of 10 particles in three dimensions. The energy is the expectation value of the Hamiltonian H and reads

$$E = \frac{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) H(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) \Psi(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N)}{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) \Psi(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N)},$$

where Ψ is the wave function of the system and $\mathbf{R}_i$ are the coordinates of each particle. If we want to compute the above integral using for example Gaussian quadrature and use for example ten mesh points for the ten particles, we need to compute a ten-dimensional integral with a total of 10^{30} mesh points. As an amusing exercise, assume that you have access to today's fastest computer with a theoretical peak capacity of more than 100 Teraflops, that is 10^{14} floating point operations per second. Assume also that every mesh point corresponds to one floating operation per second. Estimate then the time needed to compute this integral with a traditional method like Gaussian quadrature and compare this number with the estimated lifetime of the universe, $T \approx 4.7 \times 10^{17}$s. Do you have the patience to wait?

We end this first part with a discussion of a brute force Monte Carlo program which integrates

$$\int_0^1 dx \frac{4}{1+x^2} = \pi, \tag{8.16}$$

where the input is the desired number of Monte Carlo samples. Note that we transfer the variable $idum$ in order to initialize the random number generator from the function $ran0$. The variable $idum$ gets changed for every sampling. This variable is called the *seed*.

What we are doing is to employ a random number generator to obtain numbers x_i in the interval $[0, 1]$ through a call to one of the library functions $ran0$, $ran1$, $ran2$ or $ran3$ which generate random numbers in the interval $x \in [0, 1]$. These functions will be discussed in the next section. Here we simply employ these functions in order to generate a random variable. All random number generators produce pseudo-random numbers in the interval $[0, 1]$ using the so-called uniform probability distribution $p(x)$ defined as

$$p(x) = \frac{1}{b-a} \Theta(x-a) \Theta(b-x), \tag{8.17}$$

with $a = 0$ og $b = 1$. If we have a general interval $[a, b]$, we can still use these random number generators through a change of variables

$$z = a + (b-a)x, \tag{8.18}$$

with x in the interval $[0, 1]$.

The present approach to the above integral is often called 'crude' or 'Brute-Force' Monte-Carlo. Later on in this chapter we will study refinements to this simple approach. The reason for doing so is that a random generator produces points that are distributed in a homogenous way in the interval $[0, 1]$. If our function is peaked around certain values of x, we may end up sampling function values where $f(x)$ is small or near zero. Better schemes which reflect the properties of the function to be integrated are thence needed.

The algorithm is as follows

- Choose the number of Monte Carlo samples N.

– Perform a loop over N and for each step generate a a random number x_i in the interval $[0,1]$ trough a call to a random number generator.

– Use this number to evaluate $f(x_i)$.

– Evaluate the contributions to the mean value and the standard deviation for each loop.

– After N samples calculate the final mean value and the standard deviation.

The following C/C++ program[1] implements the above algorithm using the library function $ran0$ to compute π. Note again the inclusion of the $lib.h$ file which has the random number generator function **ran0**.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter08/cpp/program1.cpp

```
#include <iostream>
#include "lib.h"
using namespace std;

//      Here we define various functions called by the main program
//      this function defines the function to integrate

double func(double x);

//      Main function begins here
int main()
{
     int i, n;
     long idum;
     double crude_mc, x, sum_sigma, fx, variance;
     cout << "Read in the number of Monte-Carlo samples" << endl;
     cin >> n;
     crude_mc = sum_sigma=0. ; idum=-1 ;
//     evaluate the integral with the a crude Monte-Carlo method
      for ( i = 1;  i <= n; i++){
           x=ran0(&idum);
           fx=func(x);
           crude_mc += fx;
           sum_sigma += fx*fx;
      }
      crude_mc = crude_mc/((double) n );
      sum_sigma = sum_sigma/((double) n );
      variance=sum_sigma-crude_mc*crude_mc;

//     final output
      cout << " variance= " << variance << " Integral = "
           << crude_mc << " Exact= " << M_PI << endl;
}  // end of main program
// this function defines the function to integrate
double func(double x)
{
  double value;
```

[1]The Fortran 90/95 programs are not listed in the main text, they are found under the corresponding chapter as programs/chapter8/program$n$.f90.

Table 8.1: Results for $I = \int_0^1 dx 1/(1+x^2)$ as function of number of Monte Carlo samples N. The exact answer is $3.14159E+00$ for the integral and $4.13581E-01$ for the variance with six leading digits.

N	I	σ_N
10	3.10263E+00	3.98802E-01
100	3.02933E+00	4.04822E-01
1000	3.13395E+00	4.22881E-01
10000	3.14195E+00	4.11195E-01
100000	3.14003E+00	4.14114E-01
1000000	3.14213E+00	4.13838E-01
10000000	3.14177E+00	4.13523E-01
10^9	3.14162E+00	4.13581E-01

```
  value = 4/(1.+x*x);
  return value;
} // end of function to evaluate
```

We note that as N increases, the integral itself never reaches more than an agreement to the fourth or fifth digit. The variance also oscillates around its exact value $4.13581E-01$. Note well that the variance need not be zero but one can, with appropriate redefinitions of the integral be made smaller. A smaller variance yields also a smaller standard deviation. Improvements to this crude Monte Carlo approach will be discussed in the coming sections.

As an alternative, we could have used the random number generator provided by the C/C++ compiler through the functions $srand$ and $rand$. In this case we initialise it via the function $srand$. The random number generator is called via the function $rand$, which returns an integer from 0 to its the maximum value, defined by the variable RAND_MAX as demonstrated in the next few lines of code.

```
  invers_period = 1./RAND_MAX;
  // initialise the random number generator
  srand(time(NULL));
  // obtain a floating number x in [0,1]
      x = double(rand())*invers_period;
```

8.1.2 Second illustration, particles in a box

We give here an example of how a system evolves towards a well defined equilibrium state.

Consider a box divided into two equal halves separated by a wall. At the beginning, time $t = 0$, there are N particles on the left side. A small hole in the wall is then opened and one particle can pass through the hole per unit time.

After some time the system reaches its equilibrium state with equally many particles in both halves, $N/2$. Instead of determining complicated initial conditions for a system of N particles, we model the system by a simple statistical model. In order to simulate this system, which may consist of $N \gg 1$ particles, we assume that all particles in the left half have equal probabilities of going to the right half. We introduce the label n_l to denote the number of particles at every time on the left side, and $n_r = N - n_l$ for those on the right side. The probability for a move to the right during a time step Δt is n_l/N. The algorithm for simulating this problem may then look like as follows

- Choose the number of particles N.

- Make a loop over time, where the maximum time should be larger than the number of particles N.
- For every time step Δt there is a probability n_l/N for a move to the right. Compare this probability with a random number x.
- If $x \leq n_l/N$, decrease the number of particles in the left half by one, i.e., $n_l = n_l - 1$. Else, move a particle from the right half to the left, i.e., $n_l = n_l + 1$.
- Increase the time by one unit (the external loop).

In this case, a Monte Carlo sample corresponds to one time unit Δt.

The following simple C/C++-program illustrates this model.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter08/cpp/program2.cpp

```
// Particles in a box
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace  std;

ofstream ofile;
int main(int argc, char* argv[])
{
  char *outfilename;
  int initial_n_particles, max_time, time, random_n, nleft;
  long idum;
  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  // Read in data
  cout << "Initial number of particles = " << endl ;
  cin >> initial_n_particles;
  // setup of initial conditions
  nleft = initial_n_particles;
  max_time = 10*initial_n_particles;
  idum = -1;
  // sampling over number of particles
  for( time=0; time <= max_time; time++){
    random_n = ((int) initial_n_particles*ran0(&idum));
    if ( random_n <= nleft){
      nleft -= 1;
    }
    else{
      nleft += 1;
    }
```

```
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << setw(15) << time;
    ofile << setw(15) << nleft << endl;
  }
  return 0;
} // end main function
```

The enclosed figure shows the development of this system as function of time steps. We note that for $N = 1000$ after roughly 2000 time steps, the system has reached the equilibrium state. There are however noteworthy fluctuations around equilibrium.

If we denote $\langle n_l \rangle$ as the number of particles in the left half as a time average after *equilibrium is reached*, we can define the standard deviation as

$$\sigma = \sqrt{\langle n_l^2 \rangle - \langle n_l \rangle^2}. \tag{8.19}$$

This problem has also an analytic solution to which we can compare our numerical simulation. If $n_l(t)$ are the number of particles in the left half after t moves, the change in $n_l(t)$ in the time interval Δt is

$$\Delta n = \left(\frac{N - n_l(t)}{N} - \frac{n_l(t)}{N} \right) \Delta t, \tag{8.20}$$

and assuming that n_l and t are continuous variables we arrive at

$$\frac{dn_l(t)}{dt} = 1 - \frac{2n_l(t)}{N}, \tag{8.21}$$

whose solution is

$$n_l(t) = \frac{N}{2} \left(1 + e^{-2t/N} \right), \tag{8.22}$$

with the initial condition $n_l(t = 0) = N$.

8.1.3 Radioactive decay

Radioactive decay is among one of the classical examples on use of Monte-Carlo simulations. Assume that a the time $t = 0$ we have $N(0)$ nuclei of type X which can decay radioactively. At a time $t > 0$ we are left with $N(t)$ nuclei. With a transition probability ω, which expresses the probability that the system will make a transition to another state during a time step of one second, we have the following first-order differential equation

$$dN(t) = -\omega N(t) dt, \tag{8.23}$$

whose solution is

$$N(t) = N(0) e^{-\omega t}, \tag{8.24}$$

where we have defined the mean lifetime τ of X as

$$\tau = \frac{1}{\omega}. \tag{8.25}$$

If a nucleus X decays to a daugther nucleus Y which also can decay, we get the following coupled equations

$$\frac{dN_X(t)}{dt} = -\omega_X N_X(t), \tag{8.26}$$

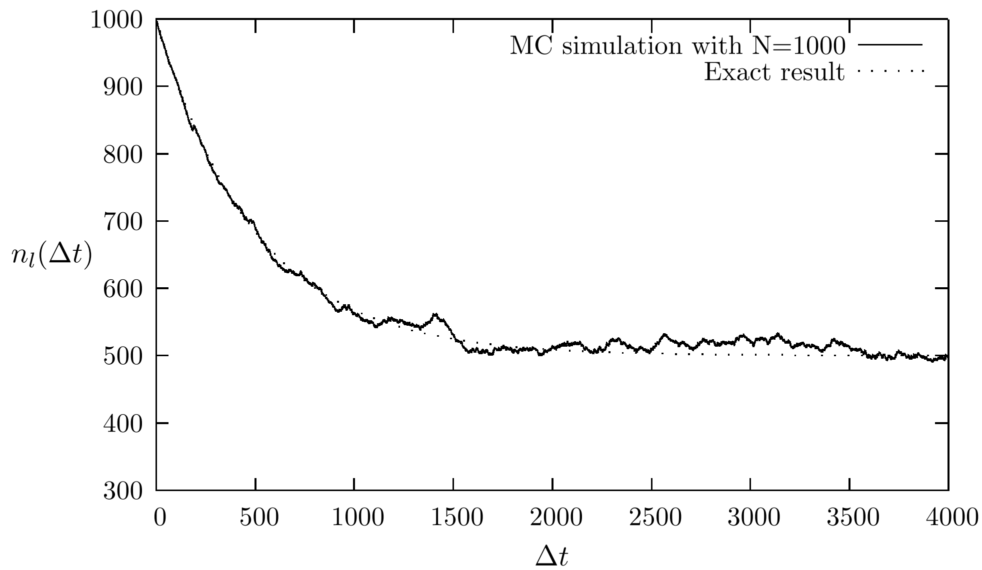


Figure 8.1: Number of particles in the left half of the container as function of the number of time steps. The solution is compared with the analytic expression. $N = 1000$.

and

$$\frac{dN_Y(t)}{dt} = -\omega_Y N_Y(t) + \omega_X N_X(t). \tag{8.27}$$

The program example in the next subsection illustrates how we can simulate such the decay process of one type of nuclei through a Monte Carlo sampling procedure.

8.1.4 Program example for radioactive decay of one type of nucleus

The program is split in four tasks, a main program with various declarations,

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter08/cpp/program3.cpp

```
// Radioactive decay of nuclei
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace std;
ofstream ofile;
// Function to read in data from screen
void initialise(int&, int&, int&, double& ) ;
// The Mc sampling for nuclear decay
void mc_sampling(int, int, int, double, int*);
// prints to screen the results of the calculations
void output(int, int, int *);
int main(int argc, char* argv[])
{
  char *outfilename;
  int initial_n_particles, max_time, number_cycles;
```

```
  double decay_probability;
  int *ncumulative;
  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  // Read in data
  initialise(initial_n_particles, max_time, number_cycles,
             decay_probability) ;
  ncumulative = new int [max_time+1];
  // Do the mc sampling
  mc_sampling(initial_n_particles, max_time, number_cycles,
              decay_probability, ncumulative);
  // Print out results
  output(max_time, number_cycles, ncumulative);
  delete [] ncumulative;
  return 0;
} // end of main function
```

followed by a part which performs the Monte Carlo sampling

```
void mc_sampling(int initial_n_particles, int max_time,
                 int number_cycles, double decay_probability,
                 int *ncumulative)
{
  int cycles, time, np, n_unstable, particle_limit;
  long idum;

  idum=-1;  // initialise random number generator
  // loop over monte carlo cycles
  // One monte carlo loop is one sample
  for (cycles = 1; cycles <= number_cycles; cycles++){
    n_unstable = initial_n_particles;
    //  accumulate the number of particles per time step per trial
    ncumulative[0] += initial_n_particles;
    // loop over each time step
    for (time=1; time <= max_time; time++){
      // for each time step, we check each particle
      particle_limit = n_unstable;
      for ( np = 1; np <=  particle_limit; np++) {
        if( ran0(&idum) <= decay_probability) {
          n_unstable=n_unstable-1;
        }
      }  // end of loop over particles
      ncumulative[time] += n_unstable;
    }  // end of loop over time steps
  }    // end of loop over MC trials
```

```
}    // end mc_sampling function
```

and finally functions for reading input and writing output data. The latter are not listed here, see under program/chapter8/program3.cpp for a full listing. The input variables are the number of Monte Carlo cycles, the maximum number of time steps, the initial number of particles and the decay probability. The output consists of the number of remaining nuclei at each time step.

8.1.5 Brief summary

In essence the Monte Carlo method contains the following ingredients

- A PDF which characterizes the system
- Random numbers which are generated so as to cover in a as uniform as possible way on the unity interval [0,1].
- A sampling rule
- An error estimation
- Techniques for improving the errors

In the next section we discuss various PDF's which may be of relevance here, thereafter we discuss how to compute random numbers. Section 8.4 discusses Monte Carlo integration in general, how to choose the correct weighting function and how to evaluate integrals with dimensions $d > 1$.

8.2 Probability distribution functions

Hitherto, we have tacitly used properties of probability distribution functions in our computation of expectation values. Here and there we have referred to the uniform PDF. It is now time to present some general features of PDFs which we may encounter when doing physics and how we define various expectation values. In addition, we derive the central limit theorem and discuss its meaning in the light of properties of various PDFs.

The following table collects properties of probability distribution functions. In our notation we reserve the label $p(x)$ for the probability of a certain event, while $P(x)$ is the cumulative probability.

Table 8.2: Important properties of PDFs.

	Discrete PDF	Continuous PDF
Domain	$\{x_1, x_2, x_3, \dots, x_N\}$	$[a, b]$
Probability	$p(x_i)$	$p(x)dx$
Cumulative	$P_i = \sum_{l=1}^{i} p(x_l)$	$P(x) = \int_a^x p(t)dt$
Positivity	$0 \leq p(x_i) \leq 1$	$p(x) \geq 0$
Positivity	$0 \leq P_i \leq 1$	$0 \leq P(x) \leq 1$
Monotonic	$P_i \geq P_j$ if $x_i \geq x_j$	$P(x_i) \geq P(x_j)$ if $x_i \geq x_j$
Normalization	$P_N = 1$	$P(b) = 1$

With a PDF we can compute expectation values of selected quantities such as

$$\langle x^k\rangle = \frac{1}{N}\sum_{i=1}^{N} x_i^k p(x_i), \tag{8.28}$$

if we have a discrete PDF or

$$\langle x^k\rangle = \int_a^b x^k p(x)dx, \tag{8.29}$$

in the case of a continuous PDF. We have already defined the mean value μ and the variance σ^2.

The expectation value of a quantity $f(x)$ is then given by for example

$$\langle f\rangle = \int_a^b f(x)p(x)dx. \tag{8.30}$$

We have already seen the use of the last equation when we applied the crude Monte Carlo approach to the evaluation of an integral.

There are at least three PDFs which one may encounter. These are the

1. uniform distribution

$$p(x) = \frac{1}{b-a}\Theta(x-a)\Theta(b-x), \tag{8.31}$$

   yielding probabilities different from zero in the interval $[a,b]$. The mean value and the variance for this distribution are discussed in section 8.3.

2. The exponential distribution

$$p(x) = \alpha e^{-\alpha x}, \tag{8.32}$$

   yielding probabilities different from zero in the interval $[0,\infty)$ and with mean value

$$\mu = \int_0^\infty xp(x)dx = \int_0^\infty x\alpha e^{-\alpha x}dx = \frac{1}{\alpha} \tag{8.33}$$

   and variance

$$\sigma^2 = \int_0^\infty x^2 p(x)dx - \mu^2 = \frac{1}{\alpha^2}. \tag{8.34}$$

3. Finally, we have the so-called univariate normal distribution, or just the normal distribution

$$p(x) = \frac{1}{b\sqrt{2\pi}}\exp\left(-\frac{(x-a)^2}{2b^2}\right) \tag{8.35}$$

   with probabilities different from zero in the interval $(-\infty,\infty)$. The integral $\int_{-\infty}^{\infty}\exp\left(-(x^2\right)dx$ appears in many calculations, its value is $\sqrt{\pi}$, a result we will need when we compute the mean value and the variance. The mean value is

$$\mu = \int_0^\infty xp(x)dx = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty} x\exp\left(-\frac{(x-a)^2}{2b^2}\right)dx, \tag{8.36}$$

   which becomes with a suitable change of variables

$$\mu = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty} b\sqrt{2}(a+b\sqrt{2}y)\exp -y^2 dy = a. \tag{8.37}$$

Similarly, the variance becomes

$$\sigma^2 = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty}(x-\mu)^2\exp\left(-\frac{(x-a)^2}{2b^2}\right)dx, \tag{8.38}$$

and inserting the mean value and performing a variable change we obtain

$$\sigma^2 = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty}b\sqrt{2}(b\sqrt{2}y)^2\exp\left(-y^2\right)dy = \frac{2b^2}{\sqrt{\pi}}\int_{-\infty}^{\infty}y^2\exp\left(-y^2\right)dy, \tag{8.39}$$

and performing a final integration by parts we obtain the well-known result $\sigma^2 = b^2$. It is useful to introduce the standard normal distribution as well, defined by $\mu = a = 0$, viz. a distribution centered around zero and with a variance $\sigma^2 = 1$, leading to

$$p(x) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{x^2}{2}\right). \tag{8.40}$$

The exponential and uniform distributions have simple cumulative functions, whereas the normal distribution does not, being proportional to the so-called error function $erf(x)$, given by

$$P(x) = .\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x}\exp\left(-\frac{t^2}{2}\right)dt, \tag{8.41}$$

which is difficult to evaluate in a quick way. Later in this chapter we will present an algorithm by Box and Mueller which allows us to compute the cumulative distribution using random variables sampled from the uniform distribution.

Some other PDFs which one encounters often in the natural sciences are the binomial distribution

$$p(x) = \left(\begin{array}{c} n \\ x \end{array}\right) y^x(1-y)^{n-x} \; x = 0, 1, \dots, n, \tag{8.42}$$

where y is the probability for a specific event, such as the tossing of a coin or moving left or right in case of a random walker. Note that x is a discrete stochastic variable.

The sequence of binomial trials is characterized by the following definitions

- Every experiment is thought to consist of N independent trials.
- In every independent trial one registers if a specific situation happens or not, such as the jump to the left or right of a random walker.
- The probability for every outcome in a single trial has the same value, for example the outcome of tossing a coin is always $1/2$.

In the next chapter we will show that the probability distribution for a random walker approaches the binomial distribution.

In order to compute the mean and variance we need to recall Newton's binomial formula

$$(a+b)^m = \sum_{n=0}^{m}\left(\begin{array}{c} m \\ n \end{array}\right) a^n b^{m-n},$$

which can be used to show that

$$\sum_{x=0}^{n}\left(\begin{array}{c} n \\ x \end{array}\right) y^x(1-y)^{n-x} = (y+1-y)^n = 1, \tag{8.43}$$

the PDF is normalized to one. The mean value is

$$\mu = \sum_{x=0}^{n} x\left(\begin{array}{c} n \\ x \end{array}\right) y^x(1-y)^{n-x} = \sum_{x=0}^{n} x\frac{n!}{x!(n-x)!}y^x(1-y)^{n-x}, \tag{8.44}$$

resulting in

$$\mu = \sum_{x=0}^{n} x\frac{(n-1)!}{(x-1)!(n-1-(x-1))!}y^{x-1}(1-y)^{n-1-(x-1)}, \tag{8.45}$$

which we rewrite as

$$\mu = ny\sum_{\nu=0}^{n}\left(\begin{array}{c} n-1 \\ \nu \end{array}\right) y^\nu(1-y)^{n-1-\nu} = ny(y+1-y)^{n-1} = ny. \tag{8.46}$$

The variance is slightly trickier to get, see the next exercises. It reads $\sigma^2 = ny(1-y)$.

Another important distribution with discrete stochastic variables x is the Poisson model, which resembles the exponential distribution and reads

$$p(x) = \frac{\lambda^x}{x!}e^{-\lambda} \; x = 0, 1, \dots, ; \lambda > 0. \tag{8.47}$$

In this case both the mean value and the variance are easier to calculate,

$$\mu = \sum_{x=0}^{\infty} x\frac{\lambda^x}{x!}e^{-\lambda} = \lambda e^{-\lambda}\sum_{x=1}^{\infty}\frac{\lambda^{x-1}}{(x-1)!} = \lambda, \tag{8.48}$$

and the variance is $\sigma^2 = \lambda$. Example of applications of the Poisson distribution is the counting of the number of α-particles emitted from a radioactive source in a given time interval. In the limit of $n \to \infty$ and for small probabilities y, the binomial distribution approaches the Poisson distribution. Setting $\lambda = ny$, with y the probability for an event in the binomial distribution we can show that

$$\lim_{n\to\infty}\left(\begin{array}{c} n \\ x \end{array}\right) y^x(1-y)^{n-x}e^{-\lambda}\sum_{x=1}^{\infty} = \frac{\lambda^x}{x!}e^{-\lambda}, \tag{8.49}$$

see for example Refs. [46, 47] for a proof.

8.2.1 Multivariable Expectation Values

Let us recapitulate some of the above concepts using a discrete PDF (which is what we end up doing anyway on a computer). The mean value of a random variable X with range $x_1, x_2, \dots, N$ is

$$\langle x \rangle = \mu = \frac{1}{N}\sum_{i=1}^{N} x_i p(x_i),$$

and the variance is

$$\langle\sigma^2\rangle = \frac{1}{N}\sum_{i=1}^{N}(x_i - \langle x\rangle)^2 p(x_i) = \frac{1}{N}\sum_{i=1}^{N}\langle(x_i-\mu_i)^2\rangle.$$

Assume now that we have two independent sets of measurements X_1 and X_2 with corresponding mean and variance μ_1 and μ_2 and $\langle\sigma^2\rangle_{X_1}$ and $\langle\sigma^2\rangle_{X_2}$. It follows that if we define the new stochastic variable

$$Y = X_1 + X_2, \tag{8.50}$$

we have

$$\mu_Y = \mu_1 + \mu_2, \tag{8.51}$$

and

$$\langle\sigma^2\rangle_Y = \langle(Y-\mu_Y)^2\rangle = \langle(X_1-\mu_1)^2\rangle + \langle(X_2-\mu_2)^2\rangle + 2\langle(X_1-\mu_1)(X_2-\mu_2)\rangle. \tag{8.52}$$

It is useful to define the so-called covariance, given by

$$\langle cov(X_1, X_2)\rangle = \langle(X_1-\mu_1)(X_2-\mu_2)\rangle \tag{8.53}$$

where we consider the averages μ_1 and μ_2 as the outcome of two separate measurements. The covariance measures thus the degree of correlation between variables. We can then rewrite the variance of Y as

$$\langle\sigma^2\rangle_Y = \sum_{j=1}^{2}\langle(X_j-\mu_j)^2\rangle + 2cov(X_1, X_2), \tag{8.54}$$

which in our notation becomes

$$\langle\sigma^2\rangle_Y = \langle\sigma^2\rangle_{X_1} + \langle\sigma^2\rangle_{X_2} + 2cov(X_1, X_2). \tag{8.55}$$

If X_1 and X_2 are two independent variables we can show that the covariance is zero, but one cannot deduce from a zero covariance whether the variables are independent or not. If our random variables which we generate are truely random numbers, then the covariance should be zero. We will see tests of standard random number generators in the next section. A way to measure the correlation between two sets of stochastic variables is the so-called correlation function $\rho(X_1, X_2)$ defined as

$$\rho(X_1, X_2) = \frac{\langle cov(X_1, X_2)\rangle}{\sqrt{\langle\sigma^2\rangle_{X_1}\langle\sigma^2\rangle_{X_2}}}. \tag{8.56}$$

Obviously, if the covariance is zero due to the fact that the variables are independent, then the correlation is zero. This quantity is often called the correlation coefficient between X_1 and X_2. We can extend this analysis to a set of stochastic variables $Y = (X_1 + X_2 + \cdots + X_N)$. We now assume that we have N different measurements of the mean and variance of a given variable. Each measurement consists again of N measurements, although we could have chosen the latter to be different from N. As an example, every evening for N days you measure N throws of two dice. The mean and variance are defined as above. The total mean value is defined as

$$\langle\mu_Y\rangle = \sum_{i=1}^{N}\langle\mu_i\rangle. \tag{8.57}$$

The total variance is however now defined as

$$\langle\sigma^2\rangle_Y = \langle(Y-\mu_Y)^2\rangle = \sum_{j=1}^{N}\langle(X_j-\mu_j)\rangle^2 = \sum_{j=1}^{N}\langle\sigma^2\rangle_{X_j} + 2\sum_{j<k}^{N}\langle(X_j-\mu_j)\rangle\langle(X_k-\mu_k)\rangle, \quad (8.58)$$

or

$$\langle\sigma^2\rangle_Y = \sum_{j=1}^{N}\langle\sigma^2\rangle_{X_j} + 2\sum_{j<k}^{N} cov(X_j, X_k). \quad (8.59)$$

If the variables are independent, the covariance is zero and the variance is reduced to

$$\langle\sigma^2\rangle_Y = \sum_{j=1}^{N}\langle\sigma^2\rangle_{X_j}, \quad (8.60)$$

and if we assume that all sets of measurements produce the same variance $\langle\sigma^2\rangle$, we end up with

$$\langle\sigma^2\rangle_Y = N\langle\sigma^2\rangle. \quad (8.61)$$

In the next subsection we combine these results with the central limit theorem in order to obtain the classical expression for the standard deviation.

8.2.2 *The central limit theorem*

Suppose we have a PDF $p(x)$ from which we generate a series N of averages $\langle x_i\rangle$. Each mean value $\langle x_i\rangle$ is viewed as the average of a specific measurement, e.g., throwing dice 100 times and then taking the average value, or producing a certain amount of random numbers. For notational ease, we set $\langle x_i\rangle = x_i$ in the discussion which follows.

If we compute the mean z of N such mean values x_i

$$z = \frac{x_1 + x_2 + \cdots + x_N}{N}, \quad (8.62)$$

the question we pose is which is the PDF of the new variable z.

The probability of obtaining an average value z is the product of the probabilities of obtaining arbitrary individual mean values x_i, but with the constraint that the average is z. We can express this through the following expression

$$\tilde{p}(z) = \int dx_1 p(x_1)\int dx_2 p(x_2)\ldots\int dx_N p(x_N)\delta(z - \frac{x_1 + x_2 + \cdots + x_N}{N}), \quad (8.63)$$

where the δ-function enbodies the constraint that the mean is z. All measurements that lead to each individual x_i are expected to be independent, which in turn means that we can express $\tilde{p}$ as the product of individual $p(x_i)$.

If we use the integral expression for the δ-function

$$\delta(z - \frac{x_1 + x_2 + \cdots + x_N}{N}) = \frac{1}{2\pi}\int_{-\infty}^{\infty} dq e^{\left(iq(z - \frac{x_1+x_2+\cdots+x_N}{N})\right)}, \quad (8.64)$$

and inserting $e^{i\mu q - i\mu q}$ where μ is the mean value we arrive at

$$\tilde{p}(z) = \frac{1}{2\pi}\int_{-\infty}^{\infty} dq e^{(iq(z-\mu))}\left[\int_{-\infty}^{\infty} dx p(x) e^{(iq(\mu - x)/N)}\right]^N, \quad (8.65)$$

with the integral over x resulting in

$$\int_{-\infty}^{\infty} dxp(x)\exp\left(iq(\mu-x)/N\right) = \int_{-\infty}^{\infty} dxp(x)\left[1+\frac{iq(\mu-x)}{N}-\frac{q^2(\mu-x)^2}{2N^2}+\dots\right]. \quad (8.66)$$

The second term on the rhs disappears since this is just the mean and employing the definition of σ^2 we have

$$\int_{-\infty}^{\infty} dxp(x)e^{(iq(\mu-x)/N)} = 1-\frac{q^2\sigma^2}{2N^2}+\dots, \quad (8.67)$$

resulting in

$$\left[\int_{-\infty}^{\infty} dxp(x)\exp\left(iq(\mu-x)/N\right)\right]^N \approx \left[1-\frac{q^2\sigma^2}{2N^2}+\dots\right]^N, \quad (8.68)$$

and in the limit $N\rightarrow\infty$ we obtain

$$\tilde{p}(z) = \frac{1}{\sqrt{2\pi}(\sigma/\sqrt{N})}\exp\left(-\frac{(z-\mu)^2}{2(\sigma/\sqrt{N})^2}\right), \quad (8.69)$$

which is the normal distribution with variance $\sigma_N^2 = \sigma^2/N$, where σ is the variance of the PDF $p(x)$ and μ is also the mean of the PDF $p(x)$.

Thus, the central limit theorem states that the PDF $\tilde{p}(z)$ of the average of N random values corresponding to a PDF $p(x)$ is a normal distribution whose mean is the mean value of the PDF $p(x)$ and whose variance is the variance of the PDF $p(x)$ divided by N, the number of values used to compute z.

The theorem is satisfied by a large class of PDFs. Note however that for a finite N, it is not always possible to find a closed expression for $\tilde{p}(x)$. The central limit theorem leads then to the well-known expression for the standard deviation, given by

$$\sigma_N = \frac{\sigma}{\sqrt{N}}. \quad (8.70)$$

The latter is true only if the average value is known exactly. This is obtained in the limit $N\rightarrow\infty$ only. Because the mean and the variance are measured quantities we obtain the familiar expression in statistics

$$\sigma_N \approx \frac{\sigma}{\sqrt{N-1}}. \quad (8.71)$$

8.3 Random numbers

Uniform deviates are just random numbers that lie within a specified range (typically 0 to 1), with any one number in the range just as likely as any other. They are, in other words, what you probably think random numbers are. However, we want to distinguish uniform deviates from other sorts of random numbers, for example numbers drawn from a normal (Gaussian) distribution of specified mean and standard deviation. These other sorts of deviates are almost always generated by performing appropriate operations on one or more uniform deviates, as we will see in subsequent sections. So, a reliable source of random uniform deviates, the subject of this section, is an essential building block for any sort of stochastic modeling or Monte Carlo computer work. A disclaimer is however appropriate. It should be fairly obvious that something as deterministic as a computer cannot generate purely random numbers.

Numbers generated by any of the standard algorithm are in reality pseudo random numbers, hopefully abiding to the following criteria:

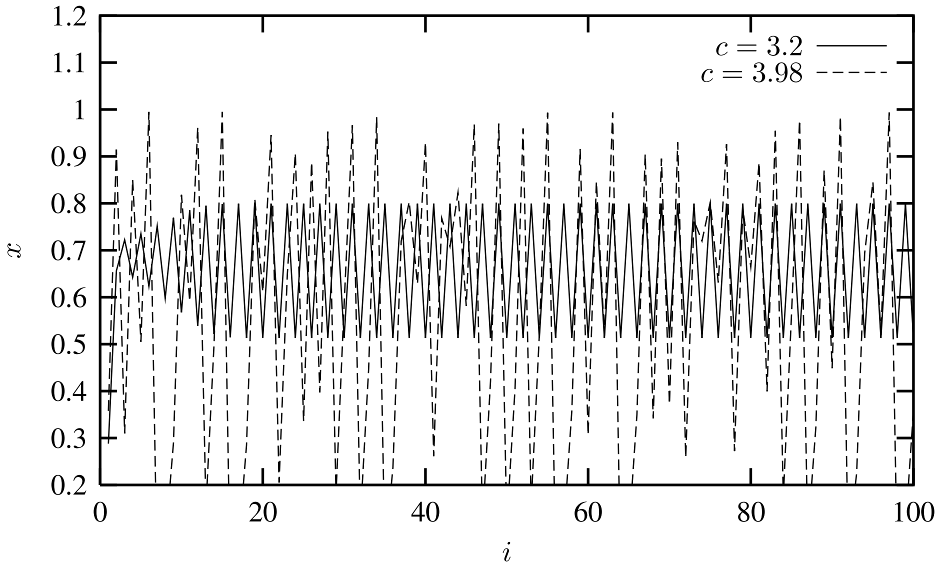


Figure 8.2: Plot of the logistic mapping $x_{i+1} = cx_i(1 - x_i)$ for $x_0 = 0.1$ and $c = 3.2$ and $c = 3.98$.

1. they produce a uniform distribution in the interval [0,1].
2. correlations between random numbers are negligible
3. the period before the same sequence of random numbers is repeated is as large as possible and finally
4. the algorithm should be fast.

That correlations, see below for more details, should be as small as possible resides in the fact that every event should be independent of the other ones. As an example, a particular simple system that exhibits a seemingly random behavior can be obtained from the iterative process

$$x_{i+1} = cx_i(1 - x_i), \tag{8.72}$$

which is often used as an example of a chaotic system. c is constant and for certain values of c and x_0 the system can settle down quickly into a regular periodic sequence of values $x_1, x_2, x_3, \dots$. For $x_0 = 0.1$ and $c = 3.2$ we obtain a periodic pattern as shown in Fig. 8.2. Changing c to $c = 3.98$ yields a sequence which does not converge to any specific pattern. The values of x_i seem purely random. Although the latter choice of c yields a seemingly random sequence of values, the various values of x harbor subtle correlations that a truly random number sequence would not possess.

The most common random number generators are based on so-called Linear congruential relations of the type

$$N_i = (aN_{i-1} + c)\mathrm{MOD}(M), \tag{8.73}$$

which yield a number in the interval [0,1] through

$$x_i = N_i/M \tag{8.74}$$

The number M is called the period and it should be as large as possible and N_0 is the starting value, or seed. The function MOD means the remainder, that is if we were to evaluate $(13)\mathrm{MOD}(9)$, the outcome is the remainder of the division $13/9$, namely 4.

The problem with such generators is that their outputs are periodic; they will start to repeat themselves with a period that is at most M. If however the parameters a and c are badly chosen, the period may be even shorter.

Consider the following example

$$N_i = (6N_{i-1} + 7)\mathrm{MOD}(5), \tag{8.75}$$

with a seed $N_0 = 2$. This generator produces the sequence $4, 1, 3, 0, 2, 4, 1, 3, 0, 2, ... \dots$, i.e., a sequence with period 5. However, increasing M may not guarantee a larger period as the following example shows

$$N_i = (27N_{i-1} + 11)\mathrm{MOD}(54), \tag{8.76}$$

which still, with $N_0 = 2$, results in $11, 38, 11, 38, 11, 38, \dots$, a period of just 2.

Typical periods for the random generators provided in the program library are of the order of $\sim 10^9$ or larger. Other random number generators which have become increasingly popular are so-called shift-register generators. In these generators each successive number depends on many preceding values (rather than the last values as in the linear congruential generator). For example, you could make a shift register generator whose lth number is the sum of the $l-i$th and $l-j$th values with modulo M,

$$N_l = (aN_{l-i} + cN_{l-j})\mathrm{MOD}(M). \tag{8.77}$$

Such a generator again produces a sequence of pseudorandom numbers but this time with a period much larger than M. It is also possible to construct more elaborate algorithms by including more than two past terms in the sum of each iteration. One example is the generator of Marsaglia and Zaman [49] which consists of two congruential relations

$$N_l = (N_{l-3} - N_{l-1})\mathrm{MOD}(2^{31} - 69), \tag{8.78}$$

followed by

$$N_l = (69069N_{l-1} + 1013904243)\mathrm{MOD}(2^{32}), \tag{8.79}$$

which according to the authors has a period larger than 2^{94}.

Moreover, rather than using modular addition, we could use the bitwise exclusive-OR ($\oplus$) operation so that

$$N_l = (N_{l-i}) \oplus (N_{l-j}) \tag{8.80}$$

where the bitwise action of $\oplus$ means that if $N_{l-i} = N_{l-j}$ the result is 0 whereas if $N_{l-i} \neq N_{l-j}$ the result is 1. As an example, consider the case where $N_{l-i} = 6$ and $N_{l-j} = 11$. The first one has a bit representation (using 4 bits only) which reads 0110 whereas the second number is 1011. Employing the $\oplus$ operator yields 1101, or $2^3 + 2^2 + 2^0 = 13$.

In Fortran90, the bitwise $\oplus$ operation is coded through the intrinsic function IEOR(m, n) where m and n are the input numbers, while in C it is given by $m \wedge n$. The program below (from Numerical Recipes, chapter 7.1) shows the function $ran0$ implements

$$N_i = (aN_{i-1})\mathrm{MOD}(M). \tag{8.81}$$

However, since a and N_{i-1} are integers and their multiplication could become greater than the standard 32 bit integer, there is a trick via Schrage's algorithm which approximates the multiplication of large integers through the factorization

$$M = aq + r,$$

where we have defined

$$q = [M/a],$$

and

$$r = M \text{ MOD } a.$$

where the brackets denote integer division. In the code below the numbers q and r are chosen so that $r < q$. To see how this works we note first that

$$(aN_{i-1})\text{MOD}(M) = (aN_{i-1} - [N_{i-1}/q]M)\text{MOD}(M), \tag{8.82}$$

since we can add or subtract any integer multiple of M from aN_{i-1}. The last term $[N_{i-1}/q]M\text{MOD}(M)$ is zero since the integer division $[N_{i-1}/q]$ just yields a constant which is multiplied with M. We can now rewrite Eq. (8.82) as

$$(aN_{i-1})\text{MOD}(M) = (aN_{i-1} - [N_{i-1}/q](aq + r))\text{MOD}(M), \tag{8.83}$$

which results in

$$(aN_{i-1})\text{MOD}(M) = (a(N_{i-1} - [N_{i-1}/q]q) - [N_{i-1}/q]r))\,\text{MOD}(M), \tag{8.84}$$

yielding

$$(aN_{i-1})\text{MOD}(M) = (a(N_{i-1}\text{MOD}(q)) - [N_{i-1}/q]r))\,\text{MOD}(M). \tag{8.85}$$

The term $[N_{i-1}/q]r$ is always smaller or equal $N_{i-1}(r/q)$ and with $r < q$ we obtain always a number smaller than N_{i-1}, which is smaller than M. And since the number $N_{i-1}\text{MOD}(q)$ is between zero and $q - 1$ then $a(N_{i-1}\text{MOD}(q)) < aq$. Combined with our definition of $q = [M/a]$ ensures that this term is also smaller than M meaning that both terms fit into a 32-bit signed integer. None of these two terms can be negative, but their difference could. The algorithm below adds M if their difference is negative. Note that the program uses the bitwise $\oplus$ operator to generate the starting point for each generation of a random number. The period of $ran0$ is $\sim 2.1 \times 10^9$. A special feature of this algorithm is that is should never be called with the initial seed set to 0.

```
    /*
     ** The function
     **            ran0()
     ** is an "Minimal" random number generator of Park and Miller
     ** (see Numerical recipe page 279). Set or reset the input value
     ** idum to any integer value (except the unlikely value MASK)
     ** to initialize the sequence; idum must not be altered between
     ** calls for sucessive deviates in a sequence.
     ** The function returns a uniform deviate between 0.0 and 1.0.
     */
double ran0(long &idum)
{
   const int a = 16807, m = 2147483647, q = 127773;
   const int r = 2836, MASK = 123459876;
   const double am = 1./m;
```

```
    long       k;
    double     ans;
    idum ^= MASK;
    k = (*idum)/q;
    idum = a*(idum - k*q) - r*k;
    // add m if negative difference
    if(idum < 0) idum += m;
    ans=am*(idum);
    idum ^= MASK;
    return ans;
} // End: function ran0()
```

The other random number generators $ran1$, $ran2$ and $ran3$ are described in detail in chapter 7.1 of Numerical Recipes. Here we limit ourselves to study selected properties of these generators.

8.3.1 Properties of selected random number generators

As mentioned previously, the underlying PDF for the generation of random numbers is the uniform distribution, meaning that the probability for finding a number x in the interval [0,1] is $p(x) = 1$.

A random number generator should produce numbers which uniformly distributed in this interval. Table 8.3 shows the distribution of $N = 10000$ random numbers generated by the functions in the program library. We note in this table that the number of points in the various intervals $0.0-0.1$, $0.1-0.2$ etc are fairly close to 1000, with some minor deviations.

Two additional measures are the standard deviation σ and the mean $\mu = \langle x \rangle$.

For the uniform distribution with N points we have that the average $\langle x^k \rangle$ is

$$\langle x^k \rangle = \frac{1}{N}\sum_{i=1}^{N} x_i^k p(x_i), \tag{8.86}$$

and taking the limit $N \to \infty$ we have

$$\langle x^k \rangle = \int_0^1 dx p(x) x^k = \int_0^1 dx x^k = \frac{1}{k+1}, \tag{8.87}$$

since $p(x) = 1$. The mean value μ is then

$$\mu = \langle x \rangle = \frac{1}{2} \tag{8.88}$$

while the standard deviation is

$$\sigma = \sqrt{\langle x^2 \rangle - \mu^2} = \frac{1}{\sqrt{12}} = 0.2886. \tag{8.89}$$

The various random number generators produce results which agree rather well with these limiting values. In the next section, in our discussion of probability distribution functions and the central limit theorem, we are to going to see that the uniform distribution evolves towards a normal distribution in the limit $N \to \infty$.

There are many other tests which can be performed. Often a picture of the numbers generated may reveal possible patterns.

Table 8.3: Number of x-values for various intervals generated by 4 random number generators, their corresponding mean values and standard deviations. All calculations have been initialized with the variable $idum = -1$.

x-bin	ran0	ran1	ran2	ran3
0.0-0.1	1013	991	938	1047
0.1-0.2	1002	1009	1040	1030
0.2-0.3	989	999	1030	993
0.3-0.4	939	960	1023	937
0.4-0.5	1038	1001	1002	992
0.5-0.6	1037	1047	1009	1009
0.6-0.7	1005	989	1003	989
0.7-0.8	986	962	985	954
0.8-0.9	1000	1027	1009	1023
0.9-1.0	991	1015	961	1026
μ	0.4997	0.5018	0.4992	0.4990
σ	0.2882	0.2892	0.2861	0.2915

Since our random numbers, which are typically generated via a linear congruential algorithm, are never fully independent, we can then define an important test which measures the degree of correlation, namely the so-called auto-correlation function C_k

$$C_k = \frac{\langle x_{i+k} x_i \rangle - \langle x_i \rangle^2}{\langle x_i^2 \rangle - \langle x_i \rangle^2}, \tag{8.90}$$

with $C_0 = 1$. Recall that $\sigma^2 = \langle x_i^2 \rangle - \langle x_i \rangle^2$. The non-vanishing of C_k for $k \neq 0$ means that the random numbers are not independent. The independence of the random numbers is crucial in the evaluation of other expectation values. If they are not independent, our assumption for approximating σ_N in Eq. (8.14) is no longer valid.

The expectation values which enter the definition of C_k are given by

$$\langle x_{i+k} x_i \rangle = \frac{1}{N-k} \sum_{i=1}^{N-k} x_i x_{i+k}. \tag{8.91}$$

Fig. 8.3 compares the auto-correlation function calculated from $ran0$ and $ran1$. As can be seen, the correlations are non-zero, but small. The fact that correlations are present is expected, since all random numbers do depend in some way on the previous numbers.

8.4 Improved Monte Carlo integration

In section 8.1 we presented a simple brute force approach to integration with the Monte Carlo method. There we sampled over a given number of points distributed uniformly in the interval $[0,1]$

$$I = \int_0^1 f(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i) = \frac{1}{N} \sum_{i=1}^{N} f(x_i) = \langle f \rangle,$$

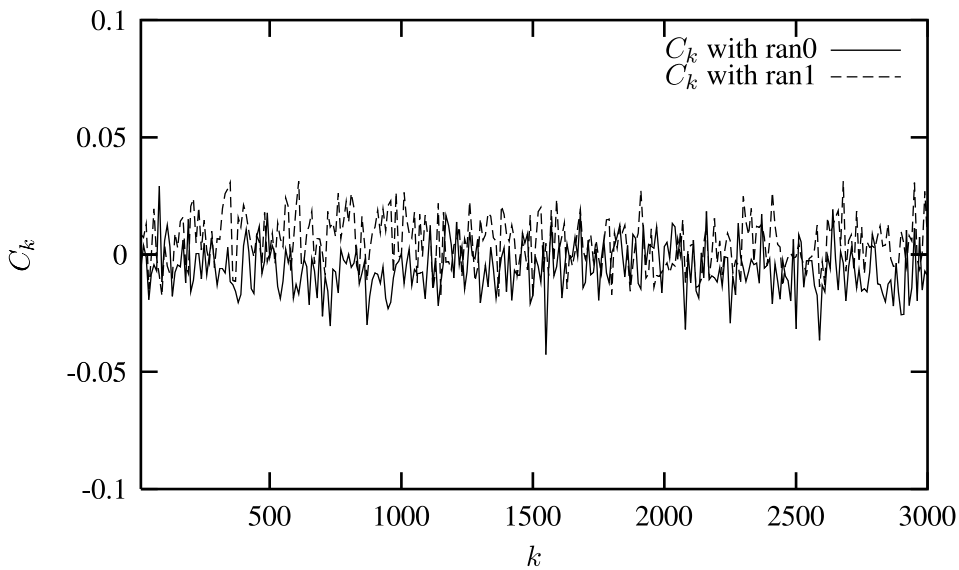


Figure 8.3: Plot of the auto-correlation function C_k for various k-values for $N = 10000$ using the random number generators $ran0$ and $ran1$.

with the weights $\omega_i = 1$.

Here we introduce two important topics which in most cases improve upon the above simple brute force approach with the uniform distribution $p(x) = 1$ for $x \in [0, 1]$. With improvements we think of a smaller variance and the need for fewer Monte Carlo samples, although each new Monte Carlo sample will most likely be more times consuming than corresponding ones of the brute force method.

– The first topic deals with change of variables, and is linked to the cumulative function $P(x)$ of a PDF $p(x)$. Obviously, not all integration limits go from $x = 0$ to $x = 1$, rather, in physics we are often confronted with integration domains like $x \in [0, \infty)$ or $x \in (-\infty, \infty)$ etc. Since all random number generators give numbers in the interval $x \in [0, 1]$, *we need a mapping from this integration interval to the explicit one under consideration.*

– The next topic deals with the shape of the integrand itself. Let us for the sake of simplicity just assume that the integration domain is again from $x = 0$ to $x = 1$. If the function to be integrated $f(x)$ has sharp peaks and is zero or small for many values of $x \in [0, 1]$, most samples of $f(x)$ give contributions to the integral I which are negligible. As a consequence we need many N samples to have a sufficient accuracy in the region where $f(x)$ is peaked. What do we do then? We try to find a new PDF $p(x)$ chosen so as to match $f(x)$ in order to render the integrand smooth. The new PDF $p(x)$ has in turn an x domain which most likely has to be mapped from the domain of the uniform distribution.

Why care at all and not be content with just a change of variables in cases where that is needed? Below we show several examples of how to improve a Monte Carlo integration through smarter choices of PDFs which render the integrand smoother. However one classic example from quantum mechanics illustrates the need for a good sampling function.

In quantum mechanics, the probability distribution function is given by $p(x) = \Psi(x)^*\Psi(x)$, where $\Psi(x)$ is the eigenfunction arising from the solution of e.g., the time-independent Schrödinger equation.

If $\Psi(x)$ is an eigenfunction, the corresponding energy eigenvalue is given by

$$H(x)\Psi(x) = E\Psi(x), \tag{8.92}$$

where $H(x)$ is the hamiltonian under consideration. The expectation value of H, assuming that the quantum mechanical PDF is normalized, is given by

$$\langle H \rangle = \int dx \Psi(x)^* H(x)\Psi(x). \tag{8.93}$$

We could insert $\Psi(x)/\Psi(x)$ right to the left of H and rewrite the last equation as

$$\langle H \rangle = \int dx \Psi(x)^* \Psi(x) \frac{H(x)}{\Psi(x)} \Psi(x), \tag{8.94}$$

or

$$\langle H \rangle = \int dx p(x) \tilde{H}(x), \tag{8.95}$$

which is on the form of an expectation value with

$$\tilde{H}(x) = \frac{H(x)}{\Psi(x)} \Psi(x). \tag{8.96}$$

The crucial point to note is that if $\Psi(x)$ is the exact eigenfunction itself with eigenvalue E, then $\tilde{H}(x)$ reduces just to the constant E and we have

$$\langle H \rangle = \int dx p(x) E = E, \tag{8.97}$$

since $p(x)$ is normalized.

However, *in most cases of interest we do not have the exact* Ψ. But if we have made a clever choice for $\Psi(x)$, the expression $\tilde{H}(x)$ exhibits a smooth behavior in the neighbourhood of the exact solution. The above example encompasses the main essence of the Monte Carlo philosophy. It is a trial approach, where intelligent guesses lead to hopefully better results.

8.4.1 Change of variables

The starting point is always the uniform distribution

$$p(x)dx = \begin{cases} dx & 0 \le x \le 1 \\ 0 & else \end{cases} \tag{8.98}$$

with $p(x) = 1$ and satisfying

$$\int_{-\infty}^{\infty} p(x)dx = 1. \tag{8.99}$$

All random number generators provided in the program library generate numbers in this domain.

When we attempt a transformation to a new variable $x \to y$ we have to conserve the probability

$$p(y)dy = p(x)dx, \tag{8.100}$$

which for the uniform distribution implies

$$p(y)dy = dx. \tag{8.101}$$

Let us assume that $p(y)$ is a PDF different from the uniform PDF $p(x) = 1$ with $x \in [0, 1]$. If we integrate the last expression we arrive at

$$x(y) = \int_0^y p(y')dy', \tag{8.102}$$

which is nothing but the cumulative distribution of $p(y)$, i.e.,

$$x(y) = P(y) = \int_0^y p(y')dy'. \tag{8.103}$$

This is an important result which has consequences for eventual improvements over the brute force Monte Carlo.

To illustrate this approach, let us look at some examples.

Example 1

Suppose we have the general uniform distribution

$$p(y)dy = \begin{cases} \frac{dy}{b-a} & a \leq y \leq b \\ 0 & else \end{cases} \tag{8.104}$$

If we wish to relate this distribution to the one in the interval $x \in [0, 1]$ we have

$$p(y)dy = \frac{dy}{b-a} = dx, \tag{8.105}$$

and integrating we obtain the cumulative function

$$x(y) = \int_a^y \frac{dy'}{b-a}, \tag{8.106}$$

yielding

$$y = a + (b-a)x, \tag{8.107}$$

a well-known result!

Example 2, the exponential distribution

Assume that

$$p(y) = e^{-y}, \tag{8.108}$$

which is the exponential distribution, important for the analysis of e.g., radioactive decay. Again, $p(x)$ is given by the uniform distribution with $x \in [0, 1]$, and with the assumption that the probability is conserved we have

$$p(y)dy = e^{-y}dy = dx, \tag{8.109}$$

which yields after integration

$$x(y) = P(y) = \int_0^y \exp{(-y')}dy' = 1 - \exp{(-y)}, \tag{8.110}$$

or

$$y(x) = -ln(1-x). \tag{8.111}$$

This gives us the new random variable y in the domain $y \in [0, \infty)$ determined through the random variable $x \in [0, 1]$ generated by functions like $ran0$.

This means that if we can factor out $\exp(-y)$ from an integrand we may have

$$I = \int_0^\infty F(y)dy = \int_0^\infty \exp(-y)G(y)dy \tag{8.112}$$

which we rewrite as

$$\int_0^\infty \exp(-y)G(y)dy = \int_0^\infty \frac{dx}{dy}G(y)dy \approx \frac{1}{N}\sum_{i=1}^{N} G(y(x_i)), \tag{8.113}$$

where x_i is a random number in the interval [0,1]. Note that in practical implementations, our random number generators for the uniform distribution never return exactly 0 or 1, but we we may come very close. We should thus in principle set $x \in (0, 1)$.

The algorithm for the last example is rather simple. In the function which sets up the integral, we simply need to call one of the random number generators like $ran0$, $ran1$, $ran2$ or $ran3$ in order to obtain numbers in the interval [0,1]. We obtain y by the taking the logarithm of $(1 - x)$. Our calling function which sets up the new random variable y may then include statements like

```
.....
idum=-1;
x=ran0(&idum);
y=-log(1.-x);
.....
```

Example 3

Another function which provides an example for a PDF is

$$p(y)dy = \frac{dy}{(a + by)^n}, \tag{8.114}$$

with $n > 1$. It is normalizable, positive definite, analytically integrable and the integral is invertible, allowing thereby the expression of a new variable in terms of the old one. The integral

$$\int_0^\infty \frac{dy}{(a + by)^n} = \frac{1}{(n-1)ba^{n-1}}, \tag{8.115}$$

gives

$$p(y)dy = \frac{(n-1)ba^{n-1}}{(a + by)^n}dy, \tag{8.116}$$

which in turn gives the cumulative function

$$x(y) = P(y) = \int_0^y \frac{(n-1)ba^{n-1}}{(a + bx)^n}dy' =, \tag{8.117}$$

resulting in

$$x(y) = 1 - \frac{1}{(1 + b/ay)^{n-1}}, \tag{8.118}$$

or

$$y = \frac{a}{b}\left((1-x)^{-1/(n-1)} - 1\right). \tag{8.119}$$

With the random variable $x \in [0, 1]$ generated by functions like $ran0$, we have again the appropriate random variable y for a new PDF.

Example 4, the normal distribution

For the normal distribution, expressed here as

$$g(x,y) = \exp{(-(x^2+y^2)/2)}dxdy. \tag{8.120}$$

it is rather difficult to find an inverse since the cumulative distribution is given by the error function $erf(x)$.

If we however switch to polar coordinates, we have for x and y

$$r = \left(x^2+y^2\right)^{1/2} \qquad \theta = tan^{-1}\frac{x}{y}, \tag{8.121}$$

resulting in

$$g(r,\theta) = r\exp{(-r^2/2)}drd\theta, \tag{8.122}$$

where the angle θ could be given by a uniform distribution in the region $[0,2\pi]$. Following example 1 above, this implies simply multiplying random numbers $x \in [0,1]$ by 2π. The variable r, defined for $r \in [0,\infty)$ needs to be related to to random numbers $x' \in [0,1]$. To achieve that, we introduce a new variable

$$u = \frac{1}{2}r^2, \tag{8.123}$$

and define a PDF

$$\exp{(-u)}du, \tag{8.124}$$

with $u \in [0,\infty)$. Using the results from example 2, we have that

$$u = -ln(1-x'), \tag{8.125}$$

where x' is a random number generated for $x' \in [0,1]$. With

$$x = rcos(\theta) = \sqrt{2u}cos(\theta), \tag{8.126}$$

and

$$y = rsin(\theta) = \sqrt{2u}sin(\theta), \tag{8.127}$$

we can obtain new random numbers x, y through

$$x = \sqrt{-2ln(1-x')}cos(\theta), \tag{8.128}$$

and

$$y = \sqrt{-2ln(1-x')}sin(\theta), \tag{8.129}$$

with $x' \in [0,1]$ and $\theta \in 2\pi[0,1]$.

A function which yields such random numbers for the normal distribution would include statements like

```
.....
idum=-1;
radius=sqrt(-2*ln(1.-ran0(idum)));
theta=2*pi*ran0(idum);
x=radius*cos(theta);
y=radius*sin(theta);
.....
```

8.4.2 Importance sampling

With the aid of the above variable transformations we address now one of the most widely used approaches to Monte Carlo integration, namely importance sampling.

Let us assume that $p(y)$ is a PDF whose behavior resembles that of a function F defined in a certain interval $[a,b]$. The normalization condition is

$$\int_a^b p(y)dy = 1. \tag{8.130}$$

We can rewrite our integral as

$$I = \int_a^b F(y)dy = \int_a^b p(y)\frac{F(y)}{p(y)}dy. \tag{8.131}$$

This integral resembles our discussion on the evaluation of the energy for a quantum mechanical system in Eq. (8.94).

Since random numbers are generated for the uniform distribution $p(x)$ with $x \in [0,1]$, we need to perform a change of variables $x \to y$ through

$$x(y) = \int_a^y p(y')dy', \tag{8.132}$$

where we used

$$p(x)dx = dx = p(y)dy. \tag{8.133}$$

If we can invert $x(y)$, we find $y(x)$ as well.

With this change of variables we can express the integral of Eq. (8.131) as

$$I = \int_a^b p(y)\frac{F(y)}{p(y)}dy = \int_a^b \frac{F(y(x))}{p(y(x))}dx, \tag{8.134}$$

meaning that a Monte Carlo evalutaion of the above integral gives

$$\int_a^b \frac{F(y(x))}{p(y(x))}dx = \frac{1}{N}\sum_{i=1}^{N}\frac{F(y(x_i))}{p(y(x_i))}. \tag{8.135}$$

The advantage of such a change of variables in case $p(y)$ follows closely F is that the integrand becomes smooth and we can sample over relevant values for the integrand. It is however not trivial to find such a function p. The conditions on p which allow us to perform these transformations are

1. p is normalizable and positive definite,
2. it is analytically integrable and
3. the integral is invertible, allowing us thereby to express a new variable in terms of the old one.

The variance is now with the definition

$$\tilde{F} = \frac{F(y(x))}{p(y(x))}, \tag{8.136}$$

given by

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}\left(\tilde{F}\right)^2 - \left(\frac{1}{N}\sum_{i=1}^{N}\tilde{F}\right)^2. \tag{8.137}$$

The algorithm for this procedure is

– Use the uniform distribution to find the random variable y in the interval [0,1]. $p(x)$ is a user provided PDF.

– Evaluate thereafter

$$I = \int_a^b F(x)dx = \int_a^b p(x)\frac{F(x)}{p(x)}dx, \quad (8.138)$$

by rewriting

$$\int_a^b p(x)\frac{F(x)}{p(x)}dx = \int_a^b \frac{F(x(y))}{p(x(y))}dy, \quad (8.139)$$

since

$$\frac{dy}{dx} = p(x). \quad (8.140)$$

– Perform then a Monte Carlo sampling for

$$\int_a^b \frac{F(x(y))}{p(x(y))}dy, \approx \frac{1}{N}\sum_{i=1}^{N}\frac{F(x(y_i))}{p(x(y_i))}, \quad (8.141)$$

with $y_i \in [0,1]$,

– and evaluate the variance as well according to Eq. (8.137).

8.4.3 Acceptance-Rejection method

This is rather simple and appealing method after von Neumann. Assume that we are looking at an interval $x \in [a,b]$, this being the domain of the PDF $p(x)$. Suppose also that the largest value our distribution function takes in this interval is M, that is

$$p(x) \le M \qquad x \in [a,b]. \quad (8.142)$$

Then we generate a random number x from the uniform distribution for $x \in [a,b]$ and a corresponding number s for the uniform distribution between $[0,M]$. If

$$p(x) \ge s, \quad (8.143)$$

we accept the new value of x, else we generate again two new random numbers x and s and perform the test in the latter equation again.

As an example, consider the evaluation of the integral

$$I = \int_0^3 \exp{(x)}dx.$$

Obviously to derive it analytically is much easier, however the integrand could pose some more difficult challenges. The aim here is simply to show how to implent the acceptance-rejection algorithm. The integral is the area below the curve $f(x) = \exp{(x)}$. If we uniformly fill the rectangle spanned by $x \in [0,3]$ and $y \in [0, \exp{(3)}]$, the fraction below the curve obatained from a uniform distribution, and multiplied by the area of the rectangle, should approximate the chosen integral. It is rather easy to implement this numerically, as shown in the following code.

Acceptance-Rejection algorithm

```
//    Loop over Monte Carlo trials n
      integral =0.;
      for ( int i = 1;   i <= n; i++){
//    Finds a random value for x in the interval [0,3]
          x = 3*ran0(&idum);
//    Finds y-value between [0,exp(3)]
          y = exp(3.0)*ran0(&idum);
//    if the value of y at exp(x) is below the curve, we accept
          if ( y  < exp(x)) s = s+ 1.0;
//    The integral is area enclosed below the line f(x)=exp(x)
     }
//   Then we multiply with the area of the rectangle and divide by the number
     of cycles
     Integral = 3.*exp(3.)*s/n
```

8.5 Monte Carlo integration of multidimensional integrals

When we deal with multidimensional integrals of the form

$$I = \int_0^1 dx_1 \int_0^1 dx_2 \dots \int_0^1 dx_d g(x_1, \dots, x_d), \tag{8.144}$$

with x_i defined in the interval $[a_i, b_i]$ we would typically need a transformation of variables of the form

$$x_i = a_i + (b_i - a_i)t_i,$$

if we were to use the uniform distribution on the interval $[0, 1]$. In this case, we need a Jacobi determinant

$$\prod_{i=1}^{d}(b_i - a_i),$$

and to convert the function $g(x_1, \dots, x_d)$ to

$$g(x_1, \dots, x_d) \to g(a_1 + (b_1 - a_1)t_1, \dots, a_d + (b_d - a_d)t_d).$$

As an example, consider the following sixth-dimensional integral

$$\int_{-\infty}^{\infty} \mathbf{dxdy} g(\mathbf{x}, \mathbf{y}), \tag{8.145}$$

where

$$g(\mathbf{x}, \mathbf{y}) = \exp{(-\mathbf{x}^2 - \mathbf{y}^2 - (\mathbf{x} - \mathbf{y})^2/2)}, \tag{8.146}$$

with $d = 6$.

We can solve this integral by employing our brute force scheme, or using importance sampling and random variables distributed according to a gaussian PDF. For the latter, if we set the mean value $\mu = 0$ and the standard deviation $\sigma = 1/\sqrt{2}$, we have

$$\frac{1}{\sqrt{\pi}} \exp{(-x^2)}, \tag{8.147}$$

and through

$$\pi^3 \int \prod_{i=1}^{6} \left(\frac{1}{\sqrt{\pi}} \exp\left(-x_i^2\right) \right) \exp\left(-(\mathbf{x}-\mathbf{y})^2/2\right) dx_1 \dots dx_6, \tag{8.148}$$

we can rewrite our integral as

$$\int f(x_1, \dots, x_d) F(x_1, \dots, x_d) \prod_{i=1}^{6} dx_i, \tag{8.149}$$

where f is the gaussian distribution.

Below we list two codes, one for the brute force integration and the other employing importance sampling with a gaussian distribution.

8.5.1 Brute force integration

`http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter08/cpp/program4.cpp`

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace std;

double brute_force_MC(double *);
//      Main function begins here
int main()
{
      int n;
      double x[6], y, fx;
      double int_mc = 0.;  double variance = 0.;
      double sum_sigma= 0. ; long idum=-1 ;
      double length=5.; // we fix the max size of the box to L=5
      double volume=pow((2*length),6);
      cout << "Read in the number of Monte-Carlo samples" << endl;
      cin >> n;
//    evaluate the integral with importance sampling
      for ( int i = 1;  i <= n; i++){
//    x[] contains the random numbers for all dimensions
        for (int j = 0; j< 6; j++) {
            x[j]=-length+2*length*ran0(&idum);
        }
        fx=brute_force_MC(x);
        int_mc += fx;
        sum_sigma += fx*fx;
      }
      int_mc = int_mc/((double) n );
      sum_sigma = sum_sigma/((double) n );
      variance=sum_sigma-int_mc*int_mc;
//    final output
       cout << setiosflags(ios::showpoint | ios::uppercase);
       cout << " Monte carlo result= " << setw(10) << setprecision(8) <<
           volume*int_mc;
```

```
     cout << " Sigma= " << setw(10) << setprecision(8) << volume*sqrt(
         variance/((double) n )) << endl;
     return 0;
}  // end of main program

// this function defines the integrand to integrate

double  brute_force_MC(double *x)
{
   double a = 1.; double b = 0.5;
// evaluate the different terms of the exponential
   double xx=x[0]*x[0]+x[1]*x[1]+x[2]*x[2];
   double yy=x[3]*x[3]+x[4]*x[4]+x[5]*x[5];
   double xy=pow((x[0]-x[3]),2)+pow((x[1]-x[4]),2)+pow((x[2]-x[5]),2);
   return exp(-a*xx-a*yy-b*xy);
} // end function for the integrand
```

8.5.2 Importance sampling

This code includes a call to the function *normal_random*, which produces random numbers from a gaussian distribution.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter08/cpp/program5.cpp

```
// importance sampling with gaussian deviates
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace std;

double gaussian_MC(double *);
double gaussian_deviate(long *);
//      Main function begins here
int main()
{
      int n;
      double x[6], y, fx;
      cout << "Read in the number of Monte-Carlo samples" << endl;
      cin >> n;
      double int_mc = 0.;  double variance = 0.;
      double sum_sigma= 0. ; long idum=-1 ;
      double length=5.; // we fix the max size of the box to L=5
      double volume=pow(acos(-1.),3.);
      double sqrt2 = 1./sqrt(2.);
//    evaluate the integral with importance sampling
      for ( int i = 1;  i <= n; i++){
//    x[] contains the random numbers for all dimensions
        for (int j = 0; j < 6; j++) {
          x[j] = gaussian_deviate(&idum)*sqrt2;
        }
        fx=gaussian_MC(x);
        int_mc += fx;
```

```
        sum_sigma += fx*fx;
      }
      int_mc = int_mc/((double) n );
      sum_sigma = sum_sigma/((double) n );
      variance=sum_sigma-int_mc*int_mc;
//    final output
       cout << setiosflags(ios::showpoint | ios::uppercase);
       cout << " Monte carlo result= " << setw(10) << setprecision(8) <<
           volume*int_mc;
       cout << " Sigma= " << setw(10) << setprecision(8) << volume*sqrt(
           variance/((double) n )) << endl;
      return 0;
}  // end of main program

// this function defines the integrand to integrate

double  gaussian_MC(double *x)
{
   double a = 0.5;
// evaluate the different terms of the exponential
   double xy=pow((x[0]-x[3]),2)+pow((x[1]-x[4]),2)+pow((x[2]-x[5]),2);
   return exp(-a*xy);
} // end function for the integrand

// random numbers with gaussian distribution
double gaussian_deviate(long * idum)
{
  static int iset = 0;
  static double gset;
  double fac, rsq, v1, v2;

  if ( idum < 0) iset =0;
  if (iset == 0) {
    do {
      v1 = 2.*ran0(idum) -1.0;
      v2 = 2.*ran0(idum) -1.0;
      rsq = v1*v1+v2*v2;
    } while (rsq >= 1.0 || rsq == 0.);
    fac = sqrt(-2.*log(rsq)/rsq);
    gset = v1*fac;
    iset = 1;
    return v2*fac;
  } else {
    iset =0;
    return gset;
  }
} // end function for gaussian deviates
```

The following table lists the results from the above two programs as function of the number of Monte Carlo samples. The suffix cr stands for the brute force approach while gd stands for the use of a Gaussian distribution function. One sees clearly that the approachwith a Gaussian distribution function yields a much improved numerical result, with fewer samples.

Table 8.4: Results for as function of number of Monte Carlo samples N. The exact answer is $I \approx 10.9626$ for the integral. The suffix cr stands for the brute force approach while gd stands for the use of a Gaussian distribution function. All calculations use ran0 as function to generate the uniform distribution.

N	I_{cr}	I_{gd}
10000	1.15247E+01	1.09128E+01
100000	1.29650E+01	1.09522E+01
1000000	1.18226E+01	1.09673E+01
10000000	1.04925E+01	1.09612E+01

8.6 Exercises and projects

Exercise 8.1: Cumulative functions

Calculate the cumulative functions $P(x)$ for the binomial and the Poisson distributions and their variances.

Exercise 8.2: Random number algorithm

Make a program which computes random numbers according to the algorithm of Marsaglia and Zaman, Eqs. (8.78) and (8.79). Compute the correlation function C_k and compare with the auto-correlation function from the function $ran0$.

Exercise 8.3: Normal distribution and random numbers

Make a function $normal\_random$ which computes random numbers for the normal distribution based on random numbers generated from the function $ran0$.

Exercise 8.4: Exponential distribution and random numbers

Make a function exp_random which computes random numbers for the exponential distribution $p(y) = e^{-\alpha y}$ based on random numbers generated from the function $ran0$.

Exercise 8.5: Monte Carlo integration

(a) Calculate the integral

$$I = \int_0^1 e^{-x^2} dx,$$

using brute force Monte Carlo with $p(x) = 1$ and importance sampling with $p(x) = ae^{-x}$ where a is a constant.

(b) Calculate the integral

$$I = \int_0^{\pi} \frac{1}{x^2 + cos^2(x)} dx,$$

with $p(x) = ae^{-x}$ where a is a constant. Determine the value of a which minimizes the variance.

Project 8.1: Decay of ^{210}Bi and ^{210}Po

In this project we are going to simulate the radioactive decay of these nuclei using sampling through random numbers. We assume that at $t = 0$ we have $N_X(0)$ nuclei of the type X which can decay radioactively. At a given time t we are left with $N_X(t)$ nuclei. With a transition rate ω_X, which is the probability that the system will make a transition to another state during a time step of one second, we get the following differential equation

$$dN_X(t) = -\omega_X N_X(t)dt,$$

whose solution is

$$N_X(t) = N_X(0)e^{-\omega_X t},$$

and where the mean lifetime of the nucleus X is

$$\tau = \frac{1}{\omega_X}.$$

If the nucleus X decays to Y, which can also decay, we get the following coupled equations

$$\frac{dN_X(t)}{dt} = -\omega_X N_X(t),$$

and

$$\frac{dN_Y(t)}{dt} = -\omega_Y N_Y(t) + \omega_X N_X(t).$$

We assume that at $t = 0$ we have $N_Y(0) = 0$. In the beginning we will have an increase of N_Y nuclei, however, they will decay thereafter. In this project we let the nucleus ^{210}Bi represent X. It decays through β-decay to ^{210}Po, which is the Y nucleus in our case. The latter decays through emision of an α-particle to ^{206}Pb, which is a stable nucleus. ^{210}Bi has a mean lifetime of 7.2 days while ^{210}Po has a mean lifetime of 200 days.

a) Find analytic solutions for the above equations assuming continuous variables and setting the number of ^{210}Po nuclei equal zero at $t = 0$.

b) Make a program which solves the above equations. What is a reasonable choice of timestep Δt? You could use the program on radioactive decay from the web-page of the course as an example and make your own for the decay of two nuclei. Compare the results from your program with the exact answer as function of $N_X(0) = 10$, 100 and 1000. Make plots of your results.

c) When ^{210}Po decays it produces an α particle. At what time does the production of α particles reach its maximum? Compare your results with the analytic ones for $N_X(0) = 10$, 100 and 1000.

Project 8.2: Numerical integration of the correlation energy of the helium atom

The task of this project is to integrate in a brute force manner a six-dimensional integral which is used to determine the ground state correlation energy between two electrons in a helium atom. We will employ both Gauss-Legendre quadrature and Monte-Carlo integration. Furthermore, you will need to parallelize your code for the Monte-Carlo integration.

We assume that the wave function of each electron can be modelled like the single-particle wave function of an electron in the hydrogen atom. The single-particle wave function for an electron i in the $1s$ state is given in terms of a dimensionless variable (the wave function is not properly normalized)

$$\mathbf{r}_i = x_i\mathbf{e}_x + y_i\mathbf{e}_y + z_i\mathbf{e}_z,$$

as

$$\psi_{1s}(\mathbf{r}_i) = e^{-\alpha r_i},$$

where α is a parameter and

$$r_i = \sqrt{x_i^2 + y_i^2 + z_i^2}.$$

We will fix $\alpha = 2$, which should correspond to the charge of the helium atom $Z = 2$.

The ansatz for the wave function for two electrons is then given by the product of two $1s$ wave functions as

$$\Psi(\mathbf{r}_1, \mathbf{r}_2) = e^{-\alpha(r_1+r_2)}.$$

Note that it is not possible to find an analytic solution to Schrödinger's equation for two interacting electrons in the helium atom.

The integral we need to solve is the quantum mechanical expectation value of the correlation energy between two electrons, namely

$$\langle \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \rangle = \int_{-\infty}^{\infty} d\mathbf{r}_1 d\mathbf{r}_2 e^{-2\alpha(r_1+r_2)} \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|}. \qquad (8.150)$$

Note that our wave function is not normalized. There is a normalization factor missing, but for this project we don't need to worry about that.

a) Use Gauss-Legendre quadrature and compute the integral by integrating for each variable x_1, y_1, z_1, x_2, y_2, z_2 from $-\infty$ to ∞. How many mesh points do you need before the results converges at the level of the fourth leading digit? Hint: the single-particle wave function $e^{-\alpha r_i}$ is more or less zero at $r_i \approx 10 - 15$. You can therefore replace the integration limits $-\infty$ and ∞ with -10 and 10, respectively. You need to check that this approximation is satisfactory.

b) Compute the same integral but now with brute force Monte Carlo and compare your results with those from the previous point. Discuss the differences. With bruce force we mean that you should use the uniform distribution.

c) Improve your brute force Monte Carlo calculation by using importance sampling. Hint: use the exponential distribution. Does the variance decrease? Does the CPU time used compared with the brute force Monte Carlo decrease in order to achieve the same accuracy? Comment your results.

d) Parallelize your code from the previous point and compare the CPU time needed with that from point [c)]. Do you achieve a good speedup?

e) The integral of Eq. (8.150) has an analytical expression. Can you find it?

Chapter 9

Random walks and the Metropolis algorithm

Nel mezzo del cammin di nostra vita, mi ritrovai per una selva oscura, ché la diritta via era smarrita. (Divina Commedia, Inferno, Canto I, 1-3)*Dante Alighieri*

The way that can be spoken of is not the constant way. (Tao Te Ching, Book I, I.1)*Lao Tzu*

9.1 Motivation

In the previous chapter we discussed technical aspects of Monte Carlo integration such as algorithms for generating random numbers and integration of multidimensional integrals. The latter topic served to illustrate two key topics in Monte Carlo simulations, namely a proper selection of variables and importance sampling. An intelligent selection of variables, good sampling techniques and guiding functions can be crucial for the outcome of our Monte Carlo simulations. Examples of this will be demonstrated in the chapters on statistical and quantum physics applications. Here we make a detour however from this main area of applications. The focus is on diffusion and random walks. The rationale for this is that the tricky part of an actual Monte Carlo simulation resides in the appropriate selection of random states, and thereby numbers, according to the probability distribution (PDF) at hand. With appropriate there is however much more to the picture than meets the eye.

Suppose our PDF is given by the well-known normal distribution. Think of for example the velocity distribution of an ideal gas in a container. In our simulations we could then accept or reject new moves with a probability proportional to the normal distribution. This would parallel our example on the sixth dimensional integral in the previous chapter. However, in this case we would end up rejecting basically all moves since the probabilities are exponentially small in most cases. The result would be that we barely moved from the initial position. Our statistical averages would then be significantly biased and most likely not very reliable.

Instead, all Monte Carlo schemes used are based on Markov processes in order to generate new random states. A Markov process is a random walk with a selected probability for making a move. The new move is independent of the previous history of the system. The Markov process is used repeatedly in Monte Carlo simulations in order to generate new random states. The reason for choosing a Markov process is that when it is run for a long enough time starting with a random state, we will eventually reach the most likely state of the system. In thermodynamics, this means that after a certain number of

Markov processes we reach an equilibrium distribution. This mimicks the way a real system reaches its most likely state at a given temperature of the surroundings.

To reach this distribution, the Markov process needs to obey two important conditions, that of ergodicity and detailed balance. These conditions impose then constraints on our algorithms for accepting or rejecting new random states. The Metropolis algorithm discussed here abides to both these constraints and is discussed in more detail in Section 9.5. The Metropolis algorithm is widely used in Monte Carlo simulations of physical systems and the understanding of it rests within the interpretation of random walks and Markov processes. However, before we do that we discuss the intimate link between random walks, Markov processes and the diffusion equation. In section 9.3 we show that a Markov process is nothing but the discretized version of the diffusion equation. Diffusion and random walks are discussed from a more experimental point of view in the next section. There we show also a simple algorithm for random walks and discuss eventual physical implications. We end this chapter with a discussion of one of the most used algorithms for generating new steps, namely the Metropolis algorithm. This algorithm, which is based on Markovian random walks satisfies both the ergodicity and detailed balance requirements and is widely in applications of Monte Carlo simulations in the natural sciences. The Metropolis algorithm is used in our studies of phase transitions in statistical physics and the simulations of quantum mechanical systems.

9.2 Diffusion equation and random walks

Physical systems subject to random influences from the ambient have a long history, dating back to the famous experiments by the British Botanist R. Brown on pollen of different plants dispersed in water. This lead to the famous concept of Brownian motion. In general, small fractions of any system exhibit the same behavior when exposed to random fluctuations of the medium. Although apparently non-deterministic, the rules obeyed by such Brownian systems are laid out within the framework of diffusion and Markov chains. The fundamental works on Brownian motion were developed by A. Einstein at the turn of the last century.

Diffusion and the diffusion equation are central topics in both Physics and Mathematics, and their ranges of applicability span from stellar dynamics to the diffusion of particles governed by Schrödinger's equation. The latter is, for a free particle, nothing but the diffusion equation in complex time!

Let us consider the one-dimensional diffusion equation. We study a large ensemble of particles performing Brownian motion along the x-axis. There is no interaction between the particles.

We define $w(x,t)dx$ as the probability of finding a given number of particles in an interval of length dx in $x \in [x, x+dx]$ at a time t. This quantity is our probability distribution function (PDF). The quantum physics equivalent of $w(x,t)$ is the wave function itself. This diffusion interpretation of Schrödinger's equation forms the starting point for diffusion Monte Carlo techniques in quantum physics.

Good overview texts are the books of Robert and Casella and Karatsas, see Refs. [46, 50].

9.2.1 Diffusion equation

From experiment there are strong indications that the flux of particles $j(x,t)$, viz., the number of particles passing x at a time t is proportional to the gradient of $w(x,t)$. This proportionality is expressed mathematically through

$$j(x,t) = -D\frac{\partial w(x,t)}{\partial x}, \tag{9.1}$$

where D is the so-called diffusion constant, with dimensionality length^2 per time. If the number of particles is conserved, we have the continuity equation

$$\frac{\partial j(x,t)}{\partial x} = -\frac{\partial w(x,t)}{\partial t}, \tag{9.2}$$

which leads to

$$\frac{\partial w(x,t)}{\partial t} = D\frac{\partial^2 w(x,t)}{\partial x^2}, \tag{9.3}$$

which is the diffusion equation in one dimension.

With the probability distribution function $w(x,t)dx$ we can use the results from the previous chapter to compute expectation values such as the mean distance

$$\langle x(t)\rangle = \int_{-\infty}^{\infty} xw(x,t)dx, \tag{9.4}$$

or

$$\langle x^2(t)\rangle = \int_{-\infty}^{\infty} x^2 w(x,t)dx, \tag{9.5}$$

which allows for the computation of the variance $\sigma^2 = \langle x^2(t)\rangle - \langle x(t)\rangle^2$. Note well that these expectation values are time-dependent. In a similar way we can also define expectation values of functions $f(x,t)$ as

$$\langle f(x,t)\rangle = \int_{-\infty}^{\infty} f(x,t)w(x,t)dx. \tag{9.6}$$

Since $w(x,t)$ is now treated as a PDF, it needs to obey the same criteria as discussed in the previous chapter. However, the normalization condition

$$\int_{-\infty}^{\infty} w(x,t)dx = 1 \tag{9.7}$$

imposes significant constraints on $w(x,t)$. These are

$$w(x=\pm\infty,t) = 0 \qquad \frac{\partial^n w(x,t)}{\partial x^n}|_{x=\pm\infty} = 0, \tag{9.8}$$

implying that when we study the time-derivative $\partial\langle x(t)\rangle/\partial t$, we obtain after integration by parts and using Eq. (9.3)

$$\frac{\partial\langle x\rangle}{\partial t} = \int_{-\infty}^{\infty} x\frac{\partial w(x,t)}{\partial t}dx = D\int_{-\infty}^{\infty} x\frac{\partial^2 w(x,t)}{\partial x^2}dx, \tag{9.9}$$

leading to

$$\frac{\partial\langle x\rangle}{\partial t} = Dx\frac{\partial w(x,t)}{\partial x}|_{x=\pm\infty} - D\int_{-\infty}^{\infty} \frac{\partial w(x,t)}{\partial x}dx, \tag{9.10}$$

implying that

$$\frac{\partial\langle x\rangle}{\partial t} = 0. \tag{9.11}$$

This means in turn that $\langle x\rangle$ is independent of time. If we choose the initial position $x(t=0) = 0$, the average displacement $\langle x\rangle = 0$. If we link this discussion to a random walk in one dimension with equal probability of jumping to the left or right and with an initial position $x = 0$, then our probability distribution remains centered around $\langle x\rangle = 0$ as function of time. However, the variance is not necessarily 0. Consider first

$$\frac{\partial\langle x^2\rangle}{\partial t} = Dx^2\frac{\partial w(x,t)}{\partial x}|_{x=\pm\infty} - 2D\int_{-\infty}^{\infty} x\frac{\partial w(x,t)}{\partial x}dx, \tag{9.12}$$

where we have performed an integration by parts as we did for $\frac{\partial\langle x\rangle}{\partial t}$. A further integration by parts results in

$$\frac{\partial\langle x^2\rangle}{\partial t} = -Dxw(x,t)|_{x=\pm\infty} + 2D\int_{-\infty}^{\infty} w(x,t)dx = 2D, \tag{9.13}$$

leading to

$$\langle x^2\rangle = 2Dt, \tag{9.14}$$

and the variance as

$$\langle x^2\rangle - \langle x\rangle^2 = 2Dt. \tag{9.15}$$

The root mean square displacement after a time t is then

$$\sqrt{\langle x^2\rangle - \langle x\rangle^2} = \sqrt{2Dt}. \tag{9.16}$$

This should be contrasted to the displacement of a free particle with initial velocity v_0. In that case the distance from the initial position after a time t is $x(t) = vt$ whereas for a diffusion process the root mean square value is $\sqrt{\langle x^2\rangle - \langle x\rangle^2} \propto \sqrt{t}$. Since diffusion is strongly linked with random walks, we could say that a random walker escapes much more slowly from the starting point than would a free particle. We can vizualize the above in the following figure. In Fig. 9.1 we have assumed that our distribution is given by a normal distribution with variance $\sigma^2 = 2Dt$, centered at $x = 0$. The distribution reads

$$w(x,t)dx = \frac{1}{\sqrt{4\pi Dt}}\exp{(-\frac{x^2}{4Dt})}dx. \tag{9.17}$$

At a time $t = 2$s the new variance is $\sigma^2 = 4D$s, implying that the root mean square value is $\sqrt{\langle x^2\rangle - \langle x\rangle^2} = 2\sqrt{D}$. At a further time $t = 8$ we have $\sqrt{\langle x^2\rangle - \langle x\rangle^2} = 4\sqrt{D}$. While time has elapsed by a factor of 4, the root mean square has only changed by a factor of 2. Fig. 9.1 demonstrates the spreadout of the distribution as time elapses. A typical example can be the diffusion of gas molecules in a container or the distribution of cream in a cup of coffee. In both cases we can assume that the the initial distribution is represented by a normal distribution.

9.2.2 Random walks

Consider now a random walker in one dimension, with probability R of moving to the right and L for moving to the left. At $t = 0$ we place the walker at $x = 0$, as indicated in Fig. 9.2. The walker can then jump, with the above probabilities, either to the left or to the right for each time step. Note that in principle we could also have the possibility that the walker remains in the same position. This is not implemented in this example. Every step has length $\Delta x = l$. Time is discretized and we have a jump either to the left or to the right at every time step. Let us now assume that we have equal probabilities for jumping to the left or to the right, i.e., $L = R = 1/2$. The average displacement after n time steps is

$$\langle x(n)\rangle = \sum_{i}^{n}\Delta x_i = 0 \qquad \Delta x_i = \pm l, \tag{9.18}$$

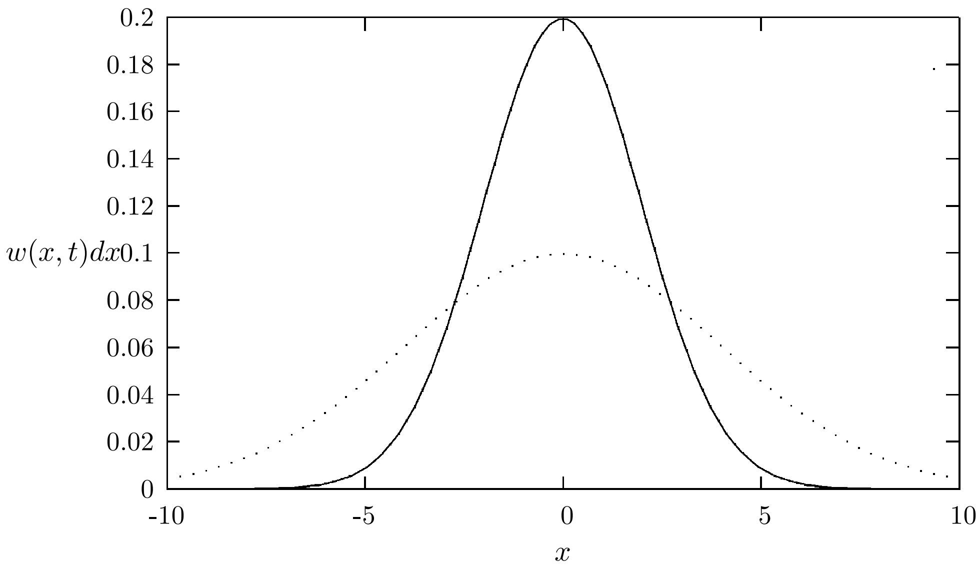


Figure 9.1: Time development of a normal distribution with variance $\sigma^2 = 2Dt$ and with $D = 1\text{m}^2/\text{s}$. The solid line represents the distribution at $t = 2\text{s}$ while the dotted line stands for $t = 8\text{s}$.

Figure 9.2: One-dimensional walker which can jump either to the left or to the right. Every step has length $\Delta x = l$.

since we have an equal probability of jumping either to the left or to right. The value of $\langle x(n)^2\rangle$ is

$$\langle x(n)^2\rangle = \left(\sum_i^n \Delta x_i\right)^2 = \sum_i^n \Delta x_i^2 + \sum_{i\neq j}^n \Delta x_i \Delta x_j = l^2 n. \tag{9.19}$$

For many enough steps the non-diagonal contribution is

$$\sum_{i\neq j}^N \Delta x_i \Delta x_j = 0, \tag{9.20}$$

since $\Delta x_{i,j} = \pm l$. The variance is then

$$\langle x(n)^2\rangle - \langle x(n)\rangle^2 = l^2 n. \tag{9.21}$$

It is also rather straightforward to compute the variance for $L \neq R$. The result is

$$\langle x(n)^2\rangle - \langle x(n)\rangle^2 = 4LRl^2 n. \tag{9.22}$$

In Eq. (9.21) the variable n represents the number of time steps. If we define $n = t/\Delta t$, we can then couple the variance result from a random walk in one dimension with the variance from the diffusion equation of Eq. (9.15) by defining the diffusion constant as

$$D = \frac{l^2}{\Delta t}. \tag{9.23}$$

In the next section we show in detail that this is the case.

The program below demonstrates the simplicity of the one-dimensional random walk algorithm. It is straightforward to extend this program to two or three dimensions as well. The input is the number of time steps, the probability for a move to the left or to the right and the total number of Monte Carlo samples. It computes the average displacement and the variance for one random walker for a given number of Monte Carlo samples. Each sample is thus to be considered as one experiment with a given number of walks. The interesting part of the algorithm is described in the function mc_sampling. The other functions read or write the results from screen or file and are similar in structure to programs discussed previously. The main program reads the name of the output file from screen and sets up the arrays containing the walker's position after a given number of steps. The corresponding program for a two-dimensional random walk (not listed in the main text) is found under programs/chapter9/program2.cpp

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter09/cpp/program1.cpp

```
/*
  1-dim random walk program.
  A walker makes several trials steps with
  a given number of walks per trial
*/
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace std;

// Function to read in data from screen, note call by reference
```

```
void initialise(int&, int&, double&) ;
// The Mc sampling for random walks
void  mc_sampling(int, int, double, int *, int *);
// prints to screen the results of the calculations
void  output(int, int, int *, int *);

int main()
{
  int max_trials, number_walks;
  double move_probability;
  // Read in data
  initialise(max_trials, number_walks, move_probability) ;
  int *walk_cumulative = new int [number_walks+1];
  int *walk2_cumulative = new int [number_walks+1];
  for (int walks = 1; walks <= number_walks; walks++){
    walk_cumulative[walks] = walk2_cumulative[walks] = 0;
  } // end initialization of vectors
  // Do the mc sampling
  mc_sampling(max_trials, number_walks, move_probability,
              walk_cumulative, walk2_cumulative);
  // Print out results
  output(max_trials, number_walks, walk_cumulative,
         walk2_cumulative);
  delete [] walk_cumulative; // free memory
  delete [] walk2_cumulative;
  return 0;
} // end main function
```

The input and output functions are

```
void initialise(int& max_trials, int& number_walks, double& move_probability
   )
{
  cout << "Number of Monte Carlo trials =";
  cin >> max_trials;
  cout << "Number of attempted walks=";
  cin >> number_walks;
  cout << "Move probability=";
  cin >> move_probability;
}  // end of function initialise


void output(int max_trials, int number_walks,
            int *walk_cumulative, int *walk2_cumulative)
{
  ofstream ofile("testwalkers.dat");
  for( int  i = 1; i <=  number_walks; i++){
    double xaverage = walk_cumulative[i]/((double) max_trials);
    double x2average = walk2_cumulative[i]/((double) max_trials);
    double variance = x2average - xaverage*xaverage;
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << setw(6) << i;
    ofile << setw(15) << setprecision(8) << xaverage;
```

```
    ofile << setw(15) << setprecision(8) << variance << endl;
  }
  ofile.close();
}  // end of function output
```

The algorithm is in the function mc_sampling and tests the probability of moving to the left or to the right by generating a random number.

```
void mc_sampling(int max_trials, int number_walks,
                 double move_probability, int *walk_cumulative,
                 int *walk2_cumulative)
{
  long idum;
  idum=-1;  // initialise random number generator
  for (int trial=1; trial <= max_trials; trial++){
    int position = 0;
    for (int walks = 1; walks <= number_walks; walks++){
      if (ran0(&idum) <= move_probability) {
        position += 1;
      }
      else {
        position -= 1;
      }
      walk_cumulative[walks] += position;
      walk2_cumulative[walks] += position*position;
    }  // end of loop over walks
  } // end of loop over trials
}   // end mc_sampling function
```

Fig. 9.3 shows that the variance increases linearly as function of the number of time steps, as expected from the analytic results. Similarly, the mean displacement in Fig. 9.4 oscillates around zero.

9.3 Microscopic derivation of the diffusion equation

When solving partial differential equations such as the diffusion equation numerically, the derivatives are always discretized. Recalling our discussions from Chapter 3, we can rewrite the time derivative as

$$\frac{\partial w(x,t)}{\partial t} \approx \frac{w(i,n+1)-w(i,n)}{\Delta t}, \tag{9.24}$$

whereas the gradient is approximated as

$$D\frac{\partial^2 w(x,t)}{\partial x^2} \approx D\frac{w(i+1,n)+w(i-1,n)-2w(i,n)}{(\Delta x)^2}, \tag{9.25}$$

resulting in the discretized diffusion equation

$$\frac{w(i,n+1)-w(i,n)}{\Delta t} = D\frac{w(i+1,n)+w(i-1,n)-2w(i,n)}{(\Delta x)^2}, \tag{9.26}$$

where n represents a given time step and i a step in the x-direction. We will come back to the solution of such equations in our chapter on partial differential equations, see Chapter 15. The aim here is to show

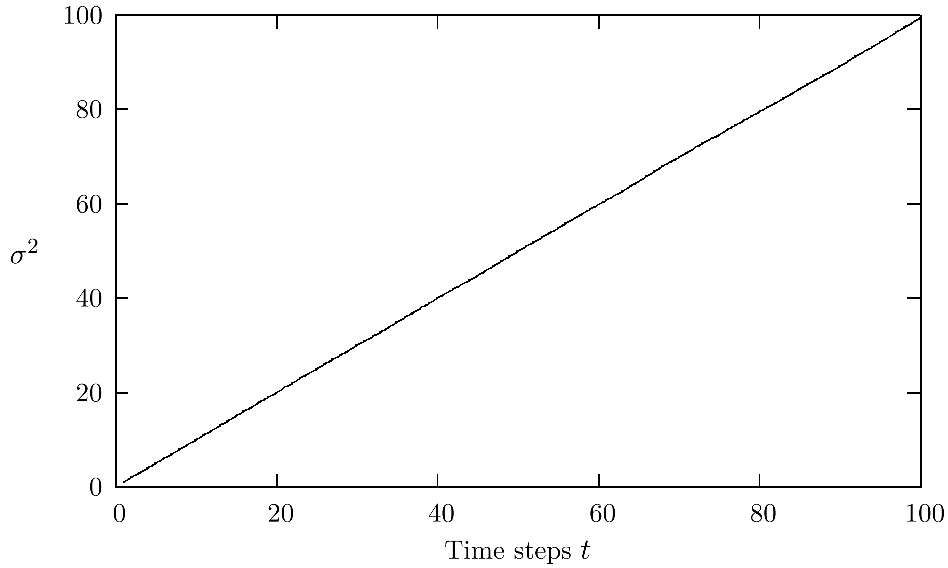


Figure 9.3: Time development of σ^2 for a random walker. 100000 Monte Carlo samples were used with the function ran1 and a seed set to -1.

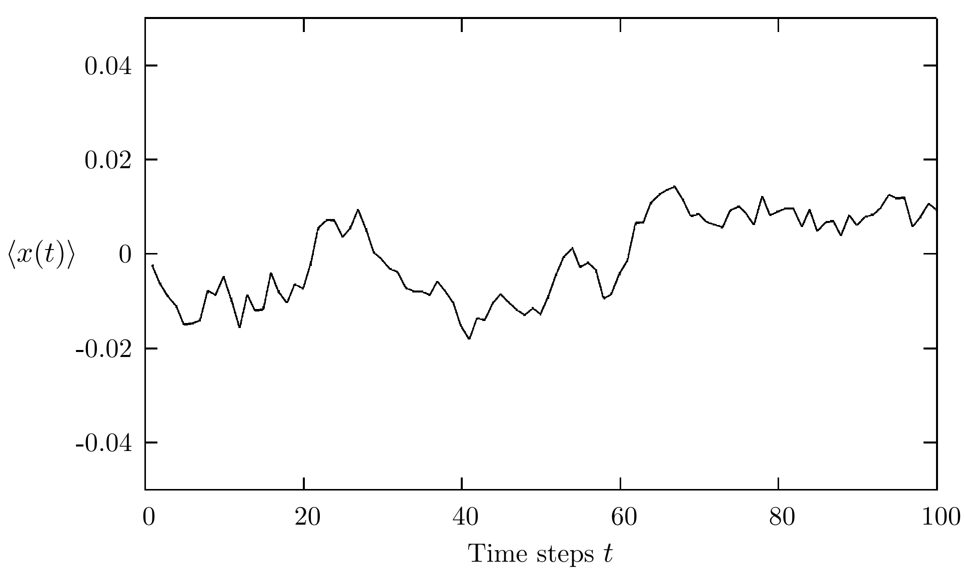


Figure 9.4: Time development of $\langle x(t) \rangle$ for a random walker. 100000 Monte Carlo samples were used with the function ran1 and a seed set to -1.

that we can derive the discretized diffusion equation from a Markov process and thereby demonstrate the close connection between the important physical process diffusion and random walks. Random walks allow for an intuitive way of picturing the process of diffusion. In addition, as demonstrated in the previous section, it is easy to simulate a random walk.

9.3.1 Discretized diffusion equation and Markov chains

A Markov process allows in principle for a microscopic description of Brownian motion. As with the random walk studied in the previous section, we consider a particle which moves along the x-axis in the form of a series of jumps with step length $\Delta x = l$. Time and space are discretized and the subsequent moves are statistically indenpendent, i.e., the new move depends only on the previous step and not on the results from earlier trials. We start at a position $x = jl = j\Delta x$ and move to a new position $x = i\Delta x$ during a step $\Delta t = \epsilon$, where $i \geq 0$ and $j \geq 0$ are integers. The original probability distribution function (PDF) of the particles is given by $w_i(t = 0)$ where i refers to a specific position on the grid in Fig. 9.2, with $i = 0$ representing $x = 0$. The function $w_i(t = 0)$ is now the discretized version of $w(x,t)$. We can regard the discretized PDF as a vector. For the Markov process we have a transition probability from a position $x = jl$ to a position $x = il$ given by

$$W_{ij}(\epsilon) = W(il - jl, \epsilon) = \begin{cases} \frac{1}{2} & |i-j| = 1 \\ 0 & \text{else} \end{cases} \tag{9.27}$$

We call W_{ij} for the transition probability and we can represent it, see below, as a matrix. Our new PDF $w_i(t = \epsilon)$ is now related to the PDF at $t = 0$ through the relation

$$w_i(t = \epsilon) = W(j \to i) w_j(t = 0). \tag{9.28}$$

This equation represents the discretized time-development of an original PDF. It is a microscopic way of representing the process shown in Fig. 9.1. Since both W and w represent probabilities, they have to be normalized, i.e., we require that at each time step we have

$$\sum_i w_i(t) = 1, \tag{9.29}$$

and

$$\sum_j W(j \to i) = 1. \tag{9.30}$$

The further constraints are $0 \leq W_{ij} \leq 1$ and $0 \leq w_j \leq 1$. Note that the probability for remaining at the same place is in general not necessarily equal zero. In our Markov process we allow only for jumps to the left or to the right.

The time development of our initial PDF can now be represented through the action of the transition probability matrix applied n times. At a time $t_n = n\epsilon$ our initial distribution has developed into

$$w_i(t_n) = \sum_j W_{ij}(t_n) w_j(0), \tag{9.31}$$

and defining

$$W(il - jl, n\epsilon) = (W^n(\epsilon))_{ij} \tag{9.32}$$

we obtain

$$w_i(n\epsilon) = \sum_j (W^n(\epsilon))_{ij} w_j(0), \tag{9.33}$$

or in matrix form
$$\hat{w}(n\epsilon) = \hat{W}^n(\epsilon)\hat{w}(0). \tag{9.34}$$
The matrix $\hat{W}$ can be written in terms of two matrices
$$\hat{W} = \frac{1}{2}\left(\hat{L} + \hat{R}\right), \tag{9.35}$$
where $\hat{L}$ and $\hat{R}$ represent the transition probabilities for a jump to the left or the right, respectively. For a 4×4 case we could write these matrices as
$$\hat{R} = \left(\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right), \tag{9.36}$$
and
$$\hat{L} = \left(\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right). \tag{9.37}$$
However, in principle these are infinite dimensional matrices since the number of time steps are very large or infinite. For the infinite case we can write these matrices $R_{ij} = \delta_{i,(j+1)}$ and $L_{ij} = \delta_{(i+1),j}$, implying that
$$\hat{L}\hat{R} = \hat{R}\hat{L} = I, \tag{9.38}$$
and
$$\hat{L} = \hat{R}^{-1} \tag{9.39}$$
To see that $\hat{L}\hat{R} = \hat{R}\hat{L} = 1$, perform e.g., the matrix multiplication
$$\hat{L}\hat{R} = \sum_k \hat{L}_{ik}\hat{R}_{kj} = \sum_k \delta_{(i+1),k}\delta_{k,(j+1)} = \delta_{i+1,j+1} = \delta_{i,j}, \tag{9.40}$$
and only the diagonal matrix elements are different from zero.

For the first time step we have thus
$$\hat{W} = \frac{1}{2}\left(\hat{L} + \hat{R}\right), \tag{9.41}$$
and using the properties in Eqs. (9.38) and (9.39) we have after two time steps
$$\hat{W}^2(2\epsilon) = \frac{1}{4}\left(\hat{L}^2 + \hat{R}^2 + 2\hat{R}\hat{L}\right), \tag{9.42}$$
and similarly after three time steps
$$\hat{W}^3(3\epsilon) = \frac{1}{8}\left(\hat{L}^3 + \hat{R}^3 + 3\hat{R}\hat{L}^2 + 3\hat{R}^2\hat{L}\right). \tag{9.43}$$
Using the binomial formula
$$\sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array}\right) \hat{a}^k\hat{b}^{n-k} = (a+b)^n, \tag{9.44}$$

we have that the transition matrix after n time steps can be written as

$$\hat{W}^n(n\epsilon)) = \frac{1}{2^n}\sum_{k=0}^{n}\left(\begin{array}{c} n \\ k \end{array}\right)\hat{R}^k\hat{L}^{n-k}, \tag{9.45}$$

or

$$\hat{W}^n(n\epsilon)) = \frac{1}{2^n}\sum_{k=0}^{n}\left(\begin{array}{c} n \\ k \end{array}\right)\hat{L}^{n-2k} = \frac{1}{2^n}\sum_{k=0}^{n}\left(\begin{array}{c} n \\ k \end{array}\right)\hat{R}^{2k-n}, \tag{9.46}$$

and using $R_{ij}^m = \delta_{i,(j+m)}$ and $L_{ij}^m = \delta_{(i+m),j}$ we arrive at

$$W(il-jl, n\epsilon) = \left\{ \begin{array}{cc} \frac{1}{2^n}\left(\begin{array}{c} n \\ \frac{1}{2}(n+i-j) \end{array}\right) & |i-j| \leq n \\ 0 & \text{else} \end{array} \right., \tag{9.47}$$

and $n+i-j$ has to be an even number. We note that the transition matrix for a Markov process has three important properties:

- It depends only on the difference in space $i-j$, it is thus homogenous in space.
- It is also isotropic in space since it is unchanged when we go from (i,j) to $(-i,-j)$.
- It is homogenous in time since it depends only the difference between the initial time and final time.

If we place the walker at $x = 0$ at $t = 0$ we can represent the initial PDF with $w_i(0) = \delta_{i,0}$. Using Eq. (9.34) we have

$$w_i(n\epsilon) = \sum_j (W^n(\epsilon))_{ij} w_j(0) = \sum_j \frac{1}{2^n}\left(\begin{array}{c} n \\ \frac{1}{2}(n+i-j) \end{array}\right)\delta_{j,0}, \tag{9.48}$$

resulting in

$$w_i(n\epsilon) = \frac{1}{2^n}\left(\begin{array}{c} n \\ \frac{1}{2}(n+i) \end{array}\right) \qquad |i| \leq n \tag{9.49}$$

Using the recursion relation for the binomials

$$\left(\begin{array}{c} n+1 \\ \frac{1}{2}(n+1+i)) \end{array}\right) = \left(\begin{array}{c} n \\ \frac{1}{2}(n+i+1) \end{array}\right) + \left(\begin{array}{c} n \\ \frac{1}{2}(n+i)-1 \end{array}\right) \tag{9.50}$$

we obtain, defining $x = il$, $t = n\epsilon$ and setting

$$w(x,t) = w(il, n\epsilon) = w_i(n\epsilon), \tag{9.51}$$

$$w(x, t+\epsilon) = \frac{1}{2}w(x+l, t) + \frac{1}{2}w(x-l, t), \tag{9.52}$$

and adding and subtracting $w(x,t)$ and multiplying both sides with l^2/ϵ we have

$$\frac{w(x,t+\epsilon) - w(x,t)}{\epsilon} = \frac{l^2}{2\epsilon}\frac{w(x+l,t) - 2w(x,t) + w(x-l,t)}{l^2}, \tag{9.53}$$

and identifying $D = l^2/2\epsilon$ and letting $l = \Delta x$ and $\epsilon = \Delta t$ we see that this is nothing but the discretized version of the diffusion equation. Taking the limits $\Delta x \to 0$ and $\Delta t \to 0$ we recover

$$\frac{\partial w(x,t)}{\partial t} = D\frac{\partial^2 w(x,t)}{\partial x^2},$$

the diffusion equation.

An illustrative example

The following simple example may help in understanding the meaning of the transition matrix $\hat{W}$ and the vector $\hat{w}$. Consider the 3×3 matrix $\hat{W}$

$$\hat{W} = \begin{pmatrix} 1/4 & 1/8 & 2/3 \\ 3/4 & 5/8 & 0 \\ 0 & 1/4 & 1/3 \end{pmatrix},$$

and we choose our initial state as

$$\hat{w}(t=0) = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

We note that both the vector and the matrix are properly normalized. Summing the vector elements gives one and summing over columns for the matrix results also in one. We act then on $\hat{w}$ with $\hat{W}$. The first iteration is

$$w_i(t=\epsilon) = W(j \to i)w_j(t=0),$$

resulting in

$$\hat{w}(t=\epsilon) = \begin{pmatrix} 1/4 \\ 3/4 \\ 0 \end{pmatrix}.$$

The next iteration results in

$$w_i(t=2\epsilon) = W(j \to i)w_j(t=\epsilon),$$

resulting in

$$\hat{w}(t=2\epsilon) = \begin{pmatrix} 5/23 \\ 21/32 \\ 6/32 \end{pmatrix}.$$

Note that the vector $\hat{w}$ is always normalized to 1. We find the steady state of the system by solving the linear set of equations

$$\mathbf{w}(t=\infty) = \mathbf{W}\mathbf{w}(t=\infty).$$

This linear set of equations reads

$$\begin{aligned} W_{11}w_1(t=\infty) + W_{12}w_2(t=\infty) + W_{13}w_3(t=\infty) &= \quad w_1(t=\infty) \\ W_{21}w_1(t=\infty) + W_{22}w_2(t=\infty) + W_{23}w_3(t=\infty) &= \quad w_2(t=\infty) \\ W_{31}w_1(t=\infty) + W_{32}w_2(t=\infty) + W_{33}w_3(t=\infty) &= \quad w_3(t=\infty) \end{aligned} \tag{9.54}$$

with the constraint that

$$\sum_i w_i(t=\infty) = 1,$$

yielding as solution

$$\hat{w}(t=\infty) = \begin{pmatrix} 4/15 \\ 8/15 \\ 3/15 \end{pmatrix}.$$

Table 9.1: Convergence to the steady state as function of number of iterations.

Iteration	w_1	w_2	w_3
0	1.00000	0.00000	0.00000
1	0.25000	0.75000	0.00000
2	0.15625	0.62625	0.18750
3	0.24609	0.52734	0.22656
4	0.27848	0.51416	0.20736
5	0.27213	0.53021	0.19766
6	0.26608	0.53548	0.19844
7	0.26575	0.53424	0.20002
8	0.26656	0.53321	0.20023
9	0.26678	0.53318	0.20005
10	0.26671	0.53332	0.19998
11	0.26666	0.53335	0.20000
12	0.26666	0.53334	0.20000
13	0.26667	0.53333	0.20000
$\hat{w}(t=\infty)$	0.26667	0.53333	0.20000

Table 9.1 demonstrates the convergence as a function of the number of iterations or time steps. We have after t-steps

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^{\mathbf{t}}\hat{\mathbf{w}}(0),$$

with $\hat{\mathbf{w}}(0)$ the distribution at $t=0$ and $\hat{\mathbf{W}}$ representing the transition probability matrix. We can always expand $\hat{\mathbf{w}}(0)$ in terms of the right eigenvectors $\hat{\mathbf{v}}$ of $\hat{\mathbf{W}}$ as

$$\hat{\mathbf{w}}(0) = \sum_i \alpha_i \hat{\mathbf{v}}_i,$$

resulting in

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^t\hat{\mathbf{w}}(0) = \hat{\mathbf{W}}^t \sum_i \alpha_i \hat{\mathbf{v}}_i = \sum_i \lambda_i^t \alpha_i \hat{\mathbf{v}}_i,$$

with λ_i the i^{th} eigenvalue corresponding to the eigenvector $\hat{\mathbf{v}}_i$.

If we assume that λ_0 is the largest eigenvector we see that in the limit $t \to \infty$, $\hat{\mathbf{w}}(t)$ becomes proportional to the corresponding eigenvector $\hat{\mathbf{v}}_0$. This is our steady state or final distribution.

9.3.2 Continuous equations

Hitherto we have considered discretized versions of all equations. Our initial probability distribution function was then given by

$$w_i(0) = \delta_{i,0},$$

and its time-development after a given time step $\Delta t = \epsilon$ is

$$w_i(t) = \sum_j W(j \to i) w_j(t=0).$$

The continuous analog to $w_i(0)$ is

$$w(\mathbf{x}) \to \delta(\mathbf{x}), \tag{9.55}$$

where we now have generalized the one-dimensional position x to a generic-dimensional vector $\mathbf{x}$. The Kroenecker δ function is replaced by the δ distribution function $\delta(\mathbf{x})$ at $t = 0$.

The transition from a state j to a state i is now replaced by a transition to a state with position $\mathbf{y}$ from a state with position $\mathbf{x}$. The discrete sum of transition probabilities can then be replaced by an integral and we obtain the new distribution at a time $t + \Delta t$ as

$$w(\mathbf{y}, t + \Delta t) = \int W(\mathbf{y}, \mathbf{x}, \Delta t) w(\mathbf{x}, t) d\mathbf{x}, \tag{9.56}$$

and after m time steps we have

$$w(\mathbf{y}, t + m\Delta t) = \int W(\mathbf{y}, \mathbf{x}, m\Delta t) w(\mathbf{x}, t) d\mathbf{x}. \tag{9.57}$$

When equilibrium is reached we have

$$w(\mathbf{y}) = \int W(\mathbf{y}, \mathbf{x}, t) w(\mathbf{x}) d\mathbf{x}. \tag{9.58}$$

We can solve the equation for $w(\mathbf{y}, t)$ by making a Fourier transform to momentum space. The PDF $w(\mathbf{x}, t)$ is related to its Fourier transform $\tilde{w}(\mathbf{k}, t)$ through

$$w(\mathbf{x}, t) = \int_{-\infty}^{\infty} d\mathbf{k} \exp{(i\mathbf{k}\mathbf{x})} \tilde{w}(\mathbf{k}, t), \tag{9.59}$$

and using the definition of the δ-function

$$\delta(\mathbf{x}) = \frac{1}{2\pi} \int_{-\infty}^{\infty} d\mathbf{k} \exp{(i\mathbf{k}\mathbf{x})}, \tag{9.60}$$

we see that

$$\tilde{w}(\mathbf{k}, 0) = 1/2\pi. \tag{9.61}$$

We can then use the Fourier-transformed diffusion equation

$$\frac{\partial \tilde{w}(\mathbf{k}, t)}{\partial t} = -D\mathbf{k}^2 \tilde{w}(\mathbf{k}, t), \tag{9.62}$$

with the obvious solution

$$\tilde{w}(\mathbf{k}, t) = \tilde{w}(\mathbf{k}, 0) \exp{\left[-(D\mathbf{k}^2 t)\right)} = \frac{1}{2\pi} \exp{\left[-(D\mathbf{k}^2 t)\right]}. \tag{9.63}$$

Using Eq. (9.59) we obtain

$$w(\mathbf{x}, t) = \int_{-\infty}^{\infty} d\mathbf{k} \exp{[i\mathbf{k}\mathbf{x}]} \frac{1}{2\pi} \exp{\left[-(D\mathbf{k}^2 t)\right]} = \frac{1}{\sqrt{4\pi D t}} \exp{\left[-(\mathbf{x}^2/4Dt)\right]}, \tag{9.64}$$

with the normalization condition

$$\int_{-\infty}^{\infty} w(\mathbf{x}, t) d\mathbf{x} = 1. \tag{9.65}$$

It is rather easy to verify by insertion that Eq. (9.64) is a solution of the diffusion equation. The solution represents the probability of finding our random walker at position $\mathbf{x}$ at time t if the initial distribution was placed at $\mathbf{x} = 0$ at $t = 0$.

There is another interesting feature worth observing. The discrete transition probability W itself is given by a binomial distribution, see Eq. (9.47). The results from the central limit theorem, see Sect. 8.2.2, state that transition probability in the limit $n \to \infty$ converges to the normal distribution. It is then possible to show that

$$W(il - jl, n\epsilon) \to W(\mathbf{y}, \mathbf{x}, \Delta t) = \frac{1}{\sqrt{4\pi D\Delta t}} \exp\left[-((\mathbf{y} - \mathbf{x})^2/4D\Delta t)\right], \tag{9.66}$$

and that it satisfies the normalization condition and is itself a solution to the diffusion equation.

9.3.3 Numerical simulation

In the two previous subsections we have given evidence that a Markov process actually yields in the limit of infinitely many steps the diffusion equation. It links therefore in a physical intuitive way the fundamental process of diffusion with random walks. It could therefore be of interest to visualize this connection through a numerical experiment. We saw in the previous subsection that one possible solution to the diffusion equation is given by a normal distribution. In addition, the transition rate for a given number of steps develops from a binomial distribution into a normal distribution in the limit of infinitely many steps. To achieve this we construct in addition a histogram which contains the number of times the walker was in a particular position x. This is given by the variable probability, which is normalized in the output function. We have omitted the initialization function, since this identical to program1.cpp or program2.cpp of this chapter. The array probability extends from −number_walks to +number_walks

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter09/cpp/program2.cpp

```
/*
  1-dim random walk program.
  A walker makes several trials steps with
  a given number of walks per trial
*/
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace  std;

// Function to read in data from screen, note call by reference
void initialise(int&, int&, double&) ;
// The Mc sampling for random walks
void  mc_sampling(int , int , double , int *, int *, int *);
// prints to screen the results of the calculations
void  output(int , int , int *, int *, int *);

int main()
{
  int max_trials , number_walks;
  double move_probability;
  // Read in data
  initialise(max_trials , number_walks, move_probability) ;
```

```
  int *walk_cumulative = new int [number_walks+1];
  int *walk2_cumulative = new int [number_walks+1];
  int *probability = new int [2*(number_walks+1)];
  for (int walks = 1; walks <= number_walks; walks++){
    walk_cumulative[walks] = walk2_cumulative[walks] = 0;
  }
  for (int walks = 0; walks <= 2*number_walks; walks++){
    probability[walks] = 0;
  } // end initialization of vectors
  // Do the mc sampling
  mc_sampling(max_trials, number_walks, move_probability,
              walk_cumulative, walk2_cumulative, probability);
  // Print out results
  output(max_trials, number_walks, walk_cumulative,
         walk2_cumulative, probability);
  delete [] walk_cumulative; // free memory
  delete [] walk2_cumulative; delete [] probability;
  return 0;
} // end main function
```

The output function contains now the normalization of the probability as well and writes this to its own file.

```
void output(int max_trials, int number_walks,
            int *walk_cumulative, int *walk2_cumulative, int * probability)
{
  ofstream ofile("testwalkers.dat");
  ofstream probfile("probability.dat");
  for( int  i = 1; i <=  number_walks; i++){
    double xaverage = walk_cumulative[i]/((double) max_trials);
    double x2average = walk2_cumulative[i]/((double) max_trials);
    double variance = x2average - xaverage*xaverage;
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << setw(6) << i;
    ofile << setw(15) << setprecision(8) << xaverage;
    ofile << setw(15) << setprecision(8) << variance << endl;
  }
  ofile.close();
  // find norm of probability
  double norm = 0.;
  for( int  i = -number_walks; i <=  number_walks; i++){
    norm += (double) probability[i+number_walks];
  }
  // write probability
  for( int  i = -number_walks; i <=  number_walks; i++){
    double histogram = probability[i+number_walks]/norm;
    probfile << setiosflags(ios::showpoint | ios::uppercase);
    probfile << setw(6) << i;
    probfile << setw(15) << setprecision(8) << histogram << endl;
  }
  probfile.close();
}  // end of function output
```

The sampling part is still done in the same function, but contains now the setup of a histogram containing the number of times the walker visited a given position x.

```
void mc_sampling(int max_trials, int number_walks,
                 double move_probability, int *walk_cumulative,
                 int *walk2_cumulative, int *probability)
{
  long idum;
  idum=-1;  // initialise random number generator
  for (int trial=1; trial <= max_trials; trial++){
    int position = 0;
    for (int walks = 1; walks <= number_walks; walks++){
      if (ran0(&idum) <= move_probability) {
        position += 1;
      }
      else {
        position -= 1;
      }
      walk_cumulative[walks] += position;
      walk2_cumulative[walks] += position*position;
      probability[position+number_walks] += 1;
    }  // end of loop over walks
  } // end of loop over trials
}   // end mc_sampling function
```

Fig. 9.5 shows the resulting probability distribution after n steps In Fig. 9.5 we have plotted the probability distribution function after a given number of time steps. Do you recognize the shape of the probabiliy distributions?

9.4 Entropy and Equilibrium Features

We use this section to motivate, in a physically intuitive way, the importance of the ergodic hypothesis via a discussion of how a Markovian process reaches an equilibrium situation after a given number of random walks. It serves then the scope of bridging the gap between a Markovian process and our discussion of the Metropolis algorithm in the next section.

To achieve this, we will use the program from the previous section, see programs/chapter9/program3.cpp and introduce the concept of entropy S. We discuss the thermodynamical meaning of the entropy and its link with the second law of thermodynamics in the next chapter. Here it will suffice to state that the entropy is a measure of the disorder of the system, thus a system which is fully ordered and stays in its fundamental state (ground state) has zero entropy, while a disordered system has a large and nonzero entropy.

The definition of the entropy S (as a dimensionless quantity here) is

$$S = -\sum_i w_i ln(w_i), \tag{9.67}$$

where w_i is the probability of finding our system in a state i. For our one-dimensional random walk case discussed in the previous sections it represents the probability for being at position $i = i\Delta x$ after a given number of time steps. In order to test this, we start with the previous program but assume now that we have N random walkers at $i = 0$ and $t = 0$ and let these random walkers diffuse as function of time. This

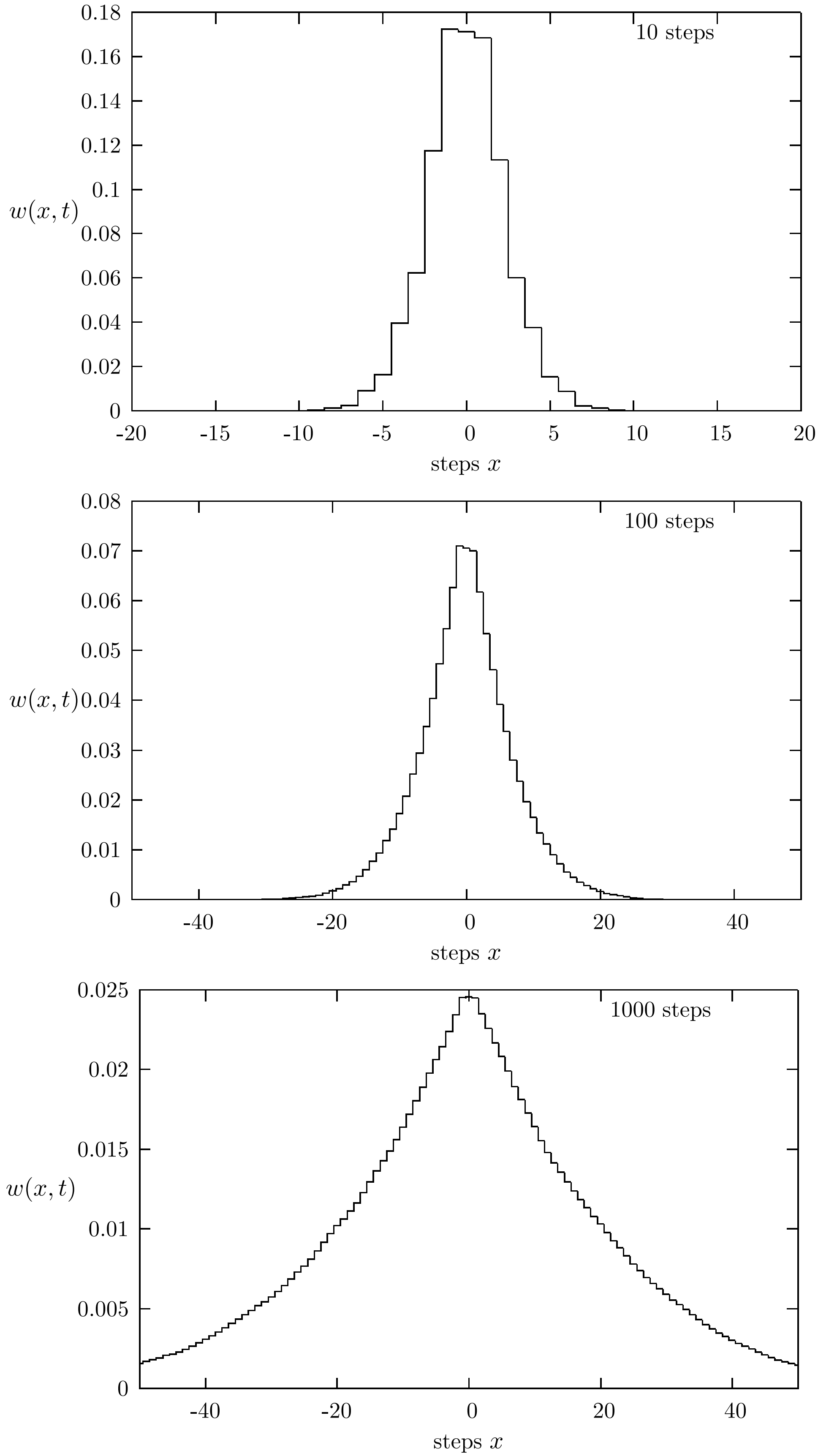


Figure 9.5: Probability distribution for one walker after 10, 100 and 1000 steps.

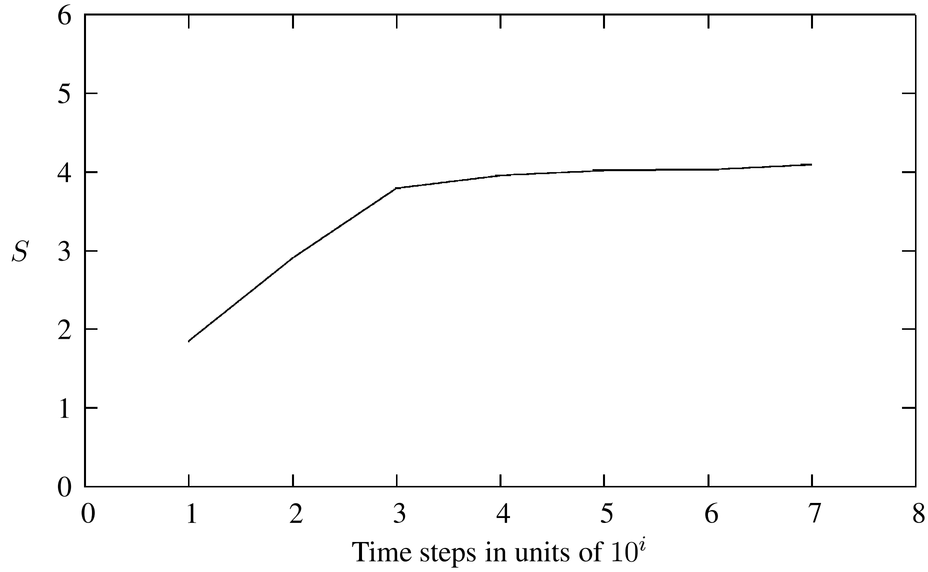


Figure 9.6: Entropy S_j as function of number of time steps j for a random walk in one dimension. Here we have used 100 walkers on a lattice of length from $L = -50$ to $L = 50$ employing periodic boundary conditions meaning that if a walker reaches the point $x = L$ it is shifted to $x = -L$ and if $x = -L$ it is shifted to $x = L$.

means simply an additional loop. We compute then, as in the previous program example, the probability distribution for N walkers after a given number of steps i along x and time steps j. We can then compute an entropy S_j for a given number of time steps by summing over all probabilities i. We show this in Fig. 9.6. The code used to compute these results is in programs/chapter9/program4.cpp. Here we have used 100 walkers on a lattice of length from $L = -50$ to $L = 50$ employing periodic boundary conditions meaning that if a walker reaches the point $x = L$ it is shifted to $x = -L$ and if $x = -L$ it is shifted to $x = L$. We see from Fig. 9.6 that for small time steps, where all particles N are in the same position or close to the initial position, the entropy is very small, reflecting the fact that we have an ordered state. As time elapses, the random walkers spread out in space (here in one dimension) and the entropy increases as there are more states, that is positions accesible to the system. We say that the system shows an increased degree of disorder. After several time steps, we see that the entropy reaches a constant value, a situation called a steady state. This signals that the system has reached its equilibrium situation and that the random walkers spread out to occupy all possible available states. At equilibrium it means thus that all states are equally probable and this is not baked into any dynamical equations such as Newton's law of motion. It occurs because the system is allowed to explore all possibilities. An important hypothesis, which has never been proven rigorously but for certain systems, is the ergodic hypothesis which states that in equilibrium all available states of a closed system have equal probability. This hypothesis states also that if we are able to simulate long enough, then one should be able to trace through all possible paths in the space of available states to reach the equilibrium situation. Our Markov process should be able to reach any state of the system from any other state if we run for long enough. Markov processes fullfil the requirement of ergodicity since all new steps are independent of the previous ones and the random walkers can thus explore with equal probability all possible positions. In general however, we know that physical processes are not independent of each other. The relation between ergodicity and physical systems is an unsettled topic.

The Metropolis algorithm which we discuss in the next section is based on a Markovian process and fullfils the requirement of ergodicity. In addition, in the next section we impose the criterion of detailed balance.

9.5 The Metropolis algorithm and detailed balance

Let us recapitulate some of our results about Markov chains and random walks.

- The time development of our PDF $w(t)$, after one time-step from $t = 0$ is given by

  $$w_i(t = \epsilon) = W(j \to i) w_j(t = 0).$$

  This equation represents the discretized time-development of an original PDF. We can rewrite this as a

  $$w_i(t = \epsilon) = W_{ij} w_j(t = 0).$$

  with the transition matrix W for a random walk given by

  $$W_{ij}(\epsilon) = W(il - jl, \epsilon) = \begin{cases} \frac{1}{2} & |i - j| = 1 \\ 0 & \text{else} \end{cases}$$

  We call W_{ij} for the transition probability and we represent it as a matrix.

- Both W and w represent probabilities and they have to be normalized, meaning that that at each time step we have

  $$\sum_i w_i(t) = 1,$$

  and

  $$\sum_j W(j \to i) = 1.$$

  The further constraints are $0 \leq W_{ij} \leq 1$ and $0 \leq w_j \leq 1$.

- We can thus write the action of W as

  $$w_i(t+1) = \sum_j W_{ij} w_j(t), \tag{9.68}$$

  or as vector-matrix relation

  $$\hat{\mathbf{w}}(t+1) = \hat{\mathbf{W}} \hat{\mathbf{w}}(t), \tag{9.69}$$

  and if we have that $||\hat{\mathbf{w}}(t+1) - \hat{\mathbf{w}}(t)|| \to 0$, we say that we have reached the most likely state of the system, the so-called steady state or equilibrium state. Another way of phrasing this is

  $$\mathbf{w}(t = \infty) = \mathbf{W} \mathbf{w}(t = \infty). \tag{9.70}$$

An important condition we require that our Markov chain should satisfy is that of detailed balance. In statistical physics this condition ensures that it is e.g., the Boltzmann distribution which is generated when equilibrium is reached. The definition for being in equilibrium is that the rates at which a system makes a transition to or from a given state i have to be equal, that is

$$\sum_i W(j \to i)w_j = \sum_i W(i \to j)w_i. \tag{9.71}$$

However, the condition that the rates should equal each other is in general not sufficient to guarantee that we, after many simulations, generate the correct distribution. We therefore introduce an additional condition, namely that of detailed balance

$$W(j \to i)w_j = W(i \to j)w_i. \tag{9.72}$$

At equilibrium detailed balance gives thus

$$\frac{W(j \to i)}{W(i \to j)} = \frac{w_i}{w_j}. \tag{9.73}$$

We introduce the Boltzmann distribution

$$w_i = \frac{\exp\left(-\beta(E_i)\right)}{Z}, \tag{9.74}$$

which states that probability of finding the system in a state i with energy E_i at an inverse temperature $\beta = 1/k_BT$ is $w_i \propto \exp\left(-\beta(E_i)\right)$. The denominator Z is a normalization constant which ensures that the sum of all probabilities is normalized to one. It is defined as the sum of probabilities over all microstates j of the system

$$Z = \sum_j \exp\left(-\beta(E_i)\right). \tag{9.75}$$

From the partition function we can in principle generate all interesting quantities for a given system in equilibrium with its surroundings at a temperature T. This is demonstrated in the next chapter.

With the probability distribution given by the Boltzmann distribution we are now in the position where we can generate expectation values for a given variable A through the definition

$$\langle A \rangle = \sum_j A_j w_j = \frac{\sum_j A_j \exp\left(-\beta(E_j)\right)}{Z}. \tag{9.76}$$

In general, most systems have an infinity of microstates making thereby the computation of Z practically impossible and a brute force Monte Carlo calculation over a given number of randomly selected microstates may therefore not yield those microstates which are important at equilibrium. To select the most important contributions we need to use the condition for detailed balance. Since this is just given by the ratios of probabilities, we never need to evaluate the partition function Z. For the Boltzmann distribution, detailed balance results in

$$\frac{w_i}{w_j} = \exp\left(-\beta(E_i - E_j)\right). \tag{9.77}$$

Let us now specialize to a system whose energy is defined by the orientation of single spins. Consider the state i, with given energy E_i represented by the following N spins

$$\begin{array}{ccccccccc} \uparrow & \uparrow & \uparrow & \dots & \uparrow & \downarrow & \uparrow & \dots & \uparrow & \downarrow \\ 1 & 2 & 3 & \dots & k-1 & k & k+1 & \dots & N-1 & N \end{array}$$

We are interested in the transition with one single spinflip to a new state j with energy E_j

$$\begin{array}{cccccccccc} \uparrow & \uparrow & \uparrow & \dots & \uparrow & \uparrow & \uparrow & \dots & \uparrow & \downarrow \\ 1 & 2 & 3 & \dots & k-1 & k & k+1 & \dots & N-1 & N \end{array}$$

This change from one microstate i (or spin configuration) to another microstate j is the configuration space analogue to a random walk on a lattice. Instead of jumping from one place to another in space, we 'jump' from one microstate to another.

However, the selection of states has to generate a final distribution which is the Boltzmann distribution. This is again the same we saw for a random walker, for the discrete case we had always a binomial distribution, whereas for the continuous case we had a normal distribution. The way we sample configurations should result in, when equilibrium is established, in the Boltzmann distribution. Else, our algorithm for selecting microstates has to be wrong.

Since we do not know the analytic form of the transition rate, we are free to model it as

$$W(i \to j) = g(i \to j)A(i \to j), \tag{9.78}$$

where g is a selection probability while A is the probability for accepting a move. It is also called the acceptance ratio. The selection probability should be same for all possible spin orientations, namely

$$g(i \to j) = \frac{1}{N}. \tag{9.79}$$

With detailed balance this gives

$$\frac{g(j \to i)A(j \to i)}{g(i \to j)A(i \to j)} = \exp\left(-\beta(E_i - E_j)\right), \tag{9.80}$$

but since the selection ratio is the same for both transitions, we have

$$\frac{A(j \to i)}{A(i \to j)} = \exp\left(-\beta(E_i - E_j)\right) \tag{9.81}$$

In general, we are looking for those spin orientations which correspond to the average energy at equilibrium.

We are in this case interested in a new state E_j whose energy is lower than E_i, viz., $\Delta E = E_j - E_i \leq 0$. A simple test would then be to accept only those microstates which lower the energy. Suppose we have ten microstates with energy $E_0 \leq E_1 \leq E_2 \leq E_3 \leq \cdots \leq E_9$. Our desired energy is E_0. At a given temperature T we start our simulation by randomly choosing state E_9. Flipping spins we may then find a path from $E_9 \to E_8 \to E_7 \cdots \to E_1 \to E_0$. This would however lead to biased statistical averages since it would violate the ergodic hypothesis discussed in the previous section. This principle states that it should be possible for any Markov process to reach every possible state of the system from any starting point if the simulations is carried out for a long enough time.

Any state in a Boltzmann distribution has a probability different from zero and if such a state cannot be reached from a given starting point, then the system is not ergodic. This means that another possible path to E_0 could be $E_9 \to E_7 \to E_8 \cdots \to E_9 \to E_5 \to E_0$ and so forth. Even though such a path could have a negligible probability it is still a possibility, and if we simulate long enough it should be included in our computation of an expectation value.

Thus, we require that our algorithm should satisfy the principle of detailed balance and be ergodic. One possible way is the Metropolis algorithm, which reads

$$A(j \to i) = \begin{cases} \exp\left(-\beta(E_i - E_j)\right) & E_i - E_j > 0 \\ 1 & else \end{cases} \tag{9.82}$$

This algorithm satisfies the condition for detailed balance and ergodicity. It is implemented as follows:

- Establish an initial energy E_b
- Do a random change of this initial state by e.g., flipping an individual spin. This new state has energy E_t. Compute then $\Delta E = E_t - E_b$
- If $\Delta E \leq 0$ accept the new configuration.
- If $\Delta E > 0$, compute $w = e^{-(\beta\Delta E)}$.
- Compare w with a random number r. If $r \leq w$ accept, else keep the old configuration.
- Compute the terms in the sums $\sum A_s w_s$.
- Repeat the above steps in order to have a large enough number of microstates
- For a given number of MC cycles, compute then expectation values.

The application of this algorithm will be discussed in detail in the next two chapters.

9.6 Exercises and projects

Exercise 9.1: Two dimensional randow walk

Extend the first program discussed in this chapter to a two-dimensional random walk with probability $1/4$ for a move to the right, left, up or down. Compute the variance for both the x and y directions and the total variance.

Exercise 9.1: Two dimensional randow walk

Use the second program to fit the computed probability distribution with a normal distribution using your calculated values of σ^2 and $\langle x \rangle$.

Project 9.1: simulation of the Boltzmann distribution

In this project the aim is to show that the Metropolis algorithm generates the Boltzmann distribution

$$P(\beta) = \frac{e^{-\beta E}}{Z}, \tag{9.83}$$

with $\beta = 1/kT$ being the inverse temperature, E is the energy of the system and Z is the partition function. The only functions you will need are those to generate random numbers.

We are going to study one single particle in equilibrium with its surroundings, the latter modeled via a large heat bath with temperature T.

The model used to describe this particle is that of an ideal gas in **one** dimension and with velocity $-v$ or v. We are interested in finding $P(v)dv$, which expresses the probability for finding the system with a given velocity $v \in [v, v + dv]$. The energy for this one-dimensional system is

$$E = \frac{1}{2}kT = \frac{1}{2}v^2, \tag{9.84}$$

with mass $m = 1$. In order to simulate the Boltzmann distribution, your program should contain the following ingredients:

– Reads in the temperature T, the number of Monte Carlo cycles, and the initial velocity. You should also read in the change in velocity δv used in every Monte Carlo step. Let the temperature have dimension energy.

– Thereafter you choose a maximum velocity given by e.g., $v_{max} \sim 10\sqrt{T}$. Then you construct a velocity interval defined by v_{max} and divided it in small intervals through v_{max}/N, with $N \sim 100 - 1000$. For each of these intervals your task is to find out how many times a given velocity during the Monte Carlo sampling appears in each specific interval.

– The number of times a given velocity appears in a specific interval is used to construct a histogram representing $P(v)dv$. To achieve this you should construct a vector $P[N]$ which contains the number of times a given velocity appears in the subinterval $v, v + dv$.

In order to find the number of velocities appearing in each interval we will employ the Metropolis algorithm. A pseudocode for this is

```
for( montecarlo_cycles=1; Max_cycles; montecarlo_cycles++) {
   ...
   // change speed as function of delta v
   v_change = (2*ran1(&idum) -1 )* delta_v;
   v_new = v_old+v_change;
   // energy change
   delta_E = 0.5*(v_new*v_new - v_old*v_old) ;
   ......
   // Metropolis algorithm begins here
     if ( ran1(&idum) <= exp(-beta*delta_E)  ) {
          accept_step = accept_step + 1 ;
          v_old = v_new ;
          .....
     }
   // thereafter we must fill in  P[N] as a function of
   // the new speed
     P[?] = ...

   // upgrade mean velocity, energy and variance
      ...
   }
```

a) Make your own algorithm which sets up the histogram $P(v)dv$, find the mean velocity, the energy, the energy variance and the number of accepted steps for a given temperature. Study the change of the number of accepted moves as a function of δv. Compare the final energy with the analytic result $E = kT/2$ for one dimension. Use $T = 4$ and set the intial velocity to zero, i.e., $v_0 = 0$. Try different values of δv. A possible start value is $\delta v = 4$. Check the final result for the energy as a function of the number of Monte Carlo cycles.

It can be useful to check your results against the analytic solutions. These can be obtained by computing the partition function of the system of interest. In our case it is given by

$$Z = \int_{-\infty}^{+\infty} e^{-\beta v^2/2} dv = \sqrt{2\pi}\beta^{-1/2}$$

From the partition function we can in turn compute the expectation value of the mean velocity and the variance. The mean velocity is given by

$$\langle v \rangle = \int_{-\infty}^{+\infty} v e^{-\beta v^2/2} dv = 0$$

The above expression holds as the integrand is an odd function of v. The mean energy and energy variance can be easily calculated. The expressions for $\langle E \rangle$ and σ_E assume the following form:

$$\langle E \rangle = \int_{-\infty}^{+\infty} \frac{v^2}{2} e^{-\beta v^2/2} dv = -\frac{1}{Z}\frac{\partial Z}{\partial \beta} = \frac{1}{2}\beta^{-1} = \frac{1}{2}T$$

$$\langle E^2 \rangle = \int_{-\infty}^{+\infty} \frac{v^4}{4} e^{-\beta v^2/2} dv = \frac{1}{Z}\frac{\partial^2 Z}{\partial \beta^2} = \frac{3}{4}\beta^{-2} = \frac{3}{4}T^2$$

and

$$\sigma_E = \langle E^2 \rangle - \langle E \rangle^2 = \frac{1}{2}T^2$$

an expected results. It is useful to compare these results with those from your program.

b) Make thereafter a plot of $ln(P(v))$ as function of E and see if you get a straight line. Comment the result.

Project 9.2: Random Walk in two dimensions

For this project you can build upon program programs/chapter9/program2.cpp (or the f90 version). You will need to compute the expectation values $\langle x(N) \rangle$, $\langle y(N) \rangle$ and

$$\langle \Delta R^2(N) \rangle = \langle x^2(N) \rangle + \langle y^2(N) \rangle - \langle x(N) \rangle^2 - \langle y(N) \rangle^2$$

where N is the number of time steps.

a) Enumerate all random walks on a square lattice for $N = 2$ and obtain exact results for $\langle x(N) \rangle$, $\langle y(N) \rangle$ and $\langle \Delta R^2(N) \rangle$. Verify your results by comparing your Monte Carlo simulations with the exact results. Assume that all four directions are equally probable.

b) Do a Monte Carlo simulation to estimate $\langle \Delta R^2(N) \rangle$ for $N = 10, 40, 60$ and 100 using a reasonable number of trials for each N. Assume that we have the asymptotic behavior

$$\langle \Delta R^2(N) \rangle \sim N^{2\nu},$$

and estimate the exponent ν from a log-log plot of $\langle \Delta R^2(N) \rangle$ versus N. If $\nu \approx 1/2$, estimate the magnutide of the self-diffusion coefficient D given by

$$\langle \Delta R^2(N) \rangle \sim 2dDN,$$

with d the dimension of the system.

c) Compute now the quantities $\langle x(N) \rangle$, $\langle y(N) \rangle$, $\langle \Delta R^2(N) \rangle$ and

$$\langle R^2(N) \rangle = \langle x^2(N) \rangle + \langle y^2(N) \rangle,$$

for the same values of N as in the previous case but now with the step probabilities $2/3$, $1/6$, $1/6$ and $1/6$ corresponding to right, left, up and down, respectively. This choice corresponds to a biased random walk with a drift to the right. What is the interpretation of $\langle x(N) \rangle$ in this case? What is the dependence of $\langle \Delta R^2(N) \rangle$ on N and does $\langle R^2(N) \rangle$ depend simply on N?

d) Consider now a random walk that starts at a site that is a distance $y = h$ above a horisontal line (ground). If the probability of a step down towards the ground is bigger than the probability of a step up, we expect that the walker will eventually reach a horisontal line. This walk is a simple model of the fall of a rain drop in the presence of a random breeze. Assume that the probabilities are 0.1, 0.6, $0, 15$ and 0.15 corresponding to up, down, right and left, respectively. Do a Monte Carlo simulation to determine the mean time τ for the walker to reach any site on the line at $x = 0$. Find the functional dependence of τ on h. Can you define a velocity in the vertical direction? Since the walker does not always move vertically, it suffers a net displacement Δx in the horizontal direction. Compute $\langle \Delta x^2 \rangle$ and find its dependence on h and τ.

Chapter 10

Monte Carlo methods in statistical physics

When you are solving a problem, don't worry. Now, after you have solved the problem, then that's the time to worry. *Richard Feynman*

10.1 Introduction and motivation

The aim of this chapter is to present examples from the physical sciences where Monte Carlo methods are widely applied. Here we focus on examples from statistical physics. and discuss one of the most studied systems, the Ising model for the interaction among classical spins. This model exhibits both first and second order phase transitions and is perhaps one of the most studied system in statistical physics with respect to simulations of phase transitions. The Norwegian-born chemist Lars Onsager`http://nobelprize.org/nobel_prizes/chemistry/laureates/1968/onsager-bio.html`, 1903-1976, developed in 1944 an ingenious mathematical description of the Ising model meant to simulate a two-dimensional model of a magnet composed of many small atomic magnets. This work proved later useful in analyzing other complex systems, such as gases sticking to solid surfaces, and hemoglobin molecules that absorb oxygen. He got the Nobel prize in chemistry in 1968 for his studies of non-equilibrium thermodynamics. Many people argue he should have received the Nobel prize in physics as well for his work on the Ising model. Another model we discuss at the end of this chapter is the so-called class of Potts models, which exhibits both first and second order type of phase transitions. Both the Ising model and the Potts model have been used to model phase transitions in solid state physics, with a particular emphasis on ferromagnetism and antiferromagnetism.

Metals like iron, nickel, cobalt and some of the rare earths (gadolinium, dysprosium) exhibit a unique magnetic behavior which is called ferromagnetism because iron (ferrum in Latin) is the most common and most dramatic example. Ferromagnetic materials exhibit a long-range ordering phenomenon at the atomic level which causes the unpaired electron spins to line up parallel with each other in a region called a domain. The long range order which creates magnetic domains in ferromagnetic materials arises from a quantum mechanical interaction at the atomic level. This interaction is remarkable in that it locks the magnetic moments of neighboring atoms into a rigid parallel order over a large number of atoms in spite of the thermal agitation which tends to randomize any atomic-level order. Sizes of domains range from a 0.1 mm to a few mm. When an external magnetic field is applied, the domains already aligned in the direction of this grow at the expense of their neighbors. For a given ferromagnetic material the long range order abruptly disappears at a certain temperature which is called the Curie temperature for the material. The Curie temperature of iron is about 1043 K while metals like cobalt and nickel have a Curie

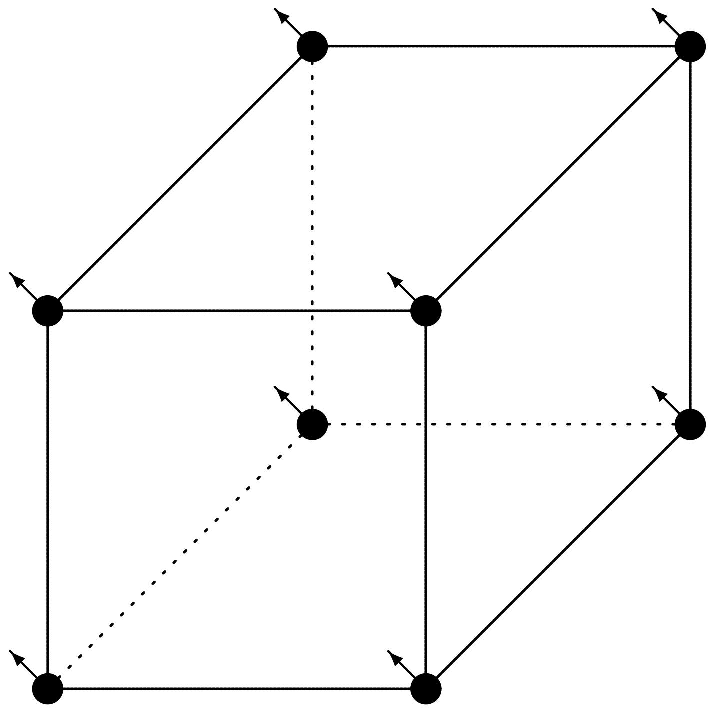


Figure 10.1: Example of a cubic lattice with atoms at each corner. Each atom has a finite magnetic moment which points in a particular direction.
.

temperature of 1388 K and 627 K, respectively, and some of the rare earth metals like gadolinium and dysprosium have 293 K and 85 k. We could think of an actual metal as composed of for example a cubic lattice with atoms at each corner with a resulting magnetic moment pointing in a particular direction, as portrayed in Fig. 10.1. In many respects, these atomic magnets are like ordinary magnets and can be thought of in terms of little magnet vectors pointing from south to north poles. The Ising model provides a simple way of describing how a magnetic material responds to thermal energy and an external magnetic field. In this model, each domain has a corresponding spin of north or south. The spins can be thought of as the poles of a bar magnet. The model assigns a value of +1 or -1 to the spins north and south respectively. The direction of the spins influences the total potential energy of the system.

Another physical case where the application of the Ising model enjoys considerable success is the description of antiferromagnetism. This is a type of magnetism where adjacent ions spontaneously align themselves at relatively low temperatures into opposite, or antiparallel, arrangements throughout the material so that it exhibits almost no gross external magnetism. In antiferromagnetic materials, which include certain metals and alloys in addition to some ionic solids, the magnetism from magnetic atoms or ions oriented in one direction is canceled out by the set of magnetic atoms or ions that are aligned in the reverse direction.

This spontaneous antiparallel coupling of atomic magnets is disrupted by heating and disappears entirely above a certain temperature, called the Néel temperature, characteristic of each antiferromagnetic material. (The Néel temperature is named for Louis Néel, French physicist, who in 1936 gave one of the first explanations of antiferromagnetism.) Some antiferromagnetic materials have Néel temperatures at, or even several hundred degrees above, room temperature, but usually these temperatures are lower. The

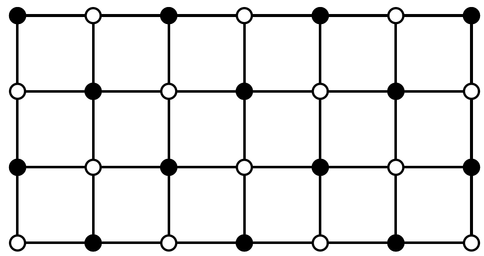


Figure 10.2: The open (white) circles at each lattice point can represent a vacant site, while the black circles can represent the absorption of an atom on a metal surface.

.

Néel temperature for manganese oxide, for example, is 122 K.

Antiferromagnetic solids exhibit special behaviour in an applied magnetic field depending upon the temperature. At very low temperatures, the solid exhibits no response to the external field, because the antiparallel ordering of atomic magnets is rigidly maintained. At higher temperatures, some atoms break free of the orderly arrangement and align with the external field. This alignment and the weak magnetism it produces in the solid reach their peak at the Néel temperature. Above this temperature, thermal agitation progressively prevents alignment of the atoms with the magnetic field, so that the weak magnetism produced in the solid by the alignment of its atoms continuously decreases as temperature is increased. For further discussion of magnetic properties and solid state physics, see for example the text of Ashcroft and Mermin [51].

As mentioned above, spin models like the Ising and Potts models can be used to model other systems as well, such as gases sticking to solid surfaces, and hemoglobin molecules that absorb oxygen. We sketch such an application in Fig. 10.2.

However, before we present the Ising model, we feel it is appropriate to refresh some important quantities in statistical physics, such as various definitions of statistical ensembles, their partition functions and relevant variables.

10.2 Review of Statistical Physics

In statistical physics the concept of an ensemble is one of the cornerstones in the definition of thermodynamical quantities. An ensemble is a collection of microphysics systems from which we derive expectations values and thermodynamical properties related to experiment. As an example, the specific heat (which is a measurable quantity in the laboratory) of a system of infinitely many particles, can be derived from the basic interactions between the microscopic constituents. The latter can span from electrons to atoms and molecules or a system of classical spins. All these microscopic constituents interact via a well-defined interaction. We say therefore that statistical physics bridges the gap between the microscopic world and the macroscopic world. Thermodynamical quantities such as the specific heat or net magnetization of a system can all be derived from a microscopic theory.

There are several types of ensembles, with their pertinent expectaction values and potentials. Table 10.1 lists the most used ensembles in statistical physics together with frequently arising extensive (depend on the size of the systems such as the number of particles) and intensive variables (apply to all components of a system), in addition to associated potentials.

Table 10.1: Overview of the most common ensembles and their variables. Here we have define $\mathcal{M}$ - to be the magnetization, $\mathcal{D}$ - the electric dipole moment, $\mathcal{H}$ - the magnetic field and $\mathcal{E}$ - to be the electric field. The last two replace the pressure as an intensive variable, while the magnetisation and the dipole moment play the same role as volume, viz they are extensive variables. The invers temperatur β regulates the mean energy while the chemical potential μ regulates the mean number of particles.

	Microcanonical	Canonical	Grand canonical	Pressure canonical
Exchange of heat with the environment	no	yes	yes	yes
Exchange of particles with the environemt	no	no	yes	no
Thermodynamical parameters	$V,\mathcal{M},\mathcal{D}$ E N	$V,\mathcal{M},\mathcal{D}$ T N	$V,\mathcal{M},\mathcal{D}$ T μ	$P,\mathcal{H},\mathcal{E}$ T N
Potential	Entropy	Helmholtz	PV	Gibbs
Energy	Internal	Internal	Internal	Enthalpy

10.2.1 Microcanonical Ensemble

The microcanonical ensemble represents an hypothetically isolated system such as a nucleus which does not exchange energy or particles via the environment. The thermodynamical quantity of interest is the entropy S which is related to the logarithm of the number of possible microscopic states $\Omega(E)$ at a given energy E that the system can access. The relation is

$$S = k_B ln\Omega. \tag{10.1}$$

When the system is in its ground state the entropy is zero since there is only one possible ground state. For excited states, we can have a higher degeneracy than one and thus an entropy which is larger than zero. We may therefore loosely state that the entropy measures the degree of order in a system. At low energies, we expect that we have only few states which are accessible and that the system prefers a specific ordering. At higher energies, more states become accessible and the entropy increases. The entropy can be used to compute observables such as the temperature

$$\frac{1}{k_BT} = \left(\frac{\partial ln\Omega}{\partial E}\right)_{N,V}, \tag{10.2}$$

the pressure

$$\frac{p}{k_BT} = \left(\frac{\partial ln\Omega}{\partial V}\right)_{N,E}, \tag{10.3}$$

or the chemical potential.

$$\frac{\mu}{k_BT} = -\left(\frac{\partial ln\Omega}{\partial N}\right)_{V,E}. \tag{10.4}$$

It is very difficult to compute the density of states $\Omega(E)$ and thereby the partition function in the microcanonical ensemble at a given energy E, since this requires the knowledge of all possible microstates at a given energy. This means that calculations are seldomly done in the microcanonical ensemble. In addition, since the microcanonical ensemble is an isolated system, it is hard to give a physical meaning to a quantity like the microcanonical temperature.

10.2.2 Canonical Ensemble

One of the most used ensembles is the canonical one, which is related to the microcanonical ensemble via a Legendre transformation. The temperature is an intensive variable in this ensemble whereas the energy follows as an expectation value. In order to calculate expectation values such as the mean energy $\langle E\rangle$ at a given temperature, we need a probability distribution. It is given by the Boltzmann distribution

$$P_i(\beta) = \frac{e^{-\beta E_i}}{Z} \tag{10.5}$$

with $\beta = 1/k_BT$ being the inverse temperature, k_B is the Boltzmann constant, E_i is the energy of a microstate i while Z is the partition function for the canonical ensemble defined as

$$Z = \sum_{i=1}^{M} e^{-\beta E_i}, \tag{10.6}$$

where the sum extends over all microstates M. The potential of interest in this case is Helmholtz' free energy. It relates the expectation value of the energy at a given temperatur T to the entropy at the same temperature via

$$F = -k_BTlnZ = \langle E\rangle - TS. \tag{10.7}$$

Helmholtz' free energy expresses the struggle between two important principles in physics, namely the strive towards an energy minimum and the drive towards higher entropy as the temperature increases. A higher entropy may be interpreted as a larger degree of disorder. When equilibrium is reached at a given temperature, we have a balance between these two principles. The numerical expression is Helmholtz' free energy. The creation of a macroscopic magnetic field from a bunch of atom-sized mini-magnets, as shown in Fig. 10.1 results from a careful balance between these two somewhat opposing principles in physics, order vs. disorder.

In the canonical ensemble the entropy is given by

$$S = k_BlnZ + k_BT\left(\frac{\partial lnZ}{\partial T}\right)_{N,V}, \tag{10.8}$$

and the pressure by

$$p = k_BT\left(\frac{\partial lnZ}{\partial V}\right)_{N,T}. \tag{10.9}$$

Similarly we can compute the chemical potential as

$$\mu = -k_B T \left(\frac{\partial lnZ}{\partial N} \right)_{V,T}. \tag{10.10}$$

For a system described by the canonical ensemble, the energy is an expectation value since we allow energy to be exchanged with the surroundings (a heat bath with temperature T).

This expectation value, the mean energy, can be calculated using

$$\langle E \rangle = k_B T^2 \left(\frac{\partial lnZ}{\partial T} \right)_{V,N} \tag{10.11}$$

or using the probability distribution P_i as

$$\langle E \rangle = \sum_{i=1}^{M} E_i P_i(\beta) = \frac{1}{Z} \sum_{i=1}^{M} E_i e^{-\beta E_i}. \tag{10.12}$$

The energy is proportional to the first derivative of the potential, Helmholtz' free energy. The corresponding variance is defined as

$$\sigma_E^2 = \langle E^2 \rangle - \langle E \rangle^2 = \frac{1}{Z} \sum_{i=1}^{M} E_i^2 e^{-\beta E_i} - \left(\frac{1}{Z} \sum_{i=1}^{M} E_i e^{-\beta E_i} \right)^2. \tag{10.13}$$

If we divide the latter quantity with kT^2 we obtain the specific heat at constant volume

$$C_V = \frac{1}{k_B T^2} \left(\langle E^2 \rangle - \langle E \rangle^2 \right), \tag{10.14}$$

which again can be related to the second derivative of Helmholtz' free energy. Using the same prescription, we can also evaluate the mean magnetization through

$$\langle \mathcal{M} \rangle = \sum_{i}^{M} \mathcal{M}_i P_i(\beta) = \frac{1}{Z} \sum_{i}^{M} \mathcal{M}_i e^{-\beta E_i}, \tag{10.15}$$

and the corresponding variance

$$\sigma_{\mathcal{M}}^2 = \langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2 = \frac{1}{Z} \sum_{i=1}^{M} \mathcal{M}_i^2 e^{-\beta E_i} - \left(\frac{1}{Z} \sum_{i=1}^{M} \mathcal{M}_i e^{-\beta E_i} \right)^2. \tag{10.16}$$

This quantity defines also the susceptibility χ

$$\chi = \frac{1}{k_B T} \left(\langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2 \right). \tag{10.17}$$

10.2.3 Grand Canonical and Pressure Canonical

Two other ensembles which are much used in statistical physics and thermodynamics are the grand canonical and pressure canonical ensembles. In the first we allow the system (in contact with a large heat bath) to exchange both heat and particles with the environment. The potential is, with a partition function $\Xi(V, T, \mu)$ with variables V, T and μ,

$$pV = k_B T ln\Xi, \tag{10.18}$$

and the entropy is given by

$$S = k_B ln\Xi + k_B T \left(\frac{\partial ln\Xi}{\partial T}\right)_{V,\mu}, \tag{10.19}$$

while the mean number of particles is

$$\langle N \rangle = k_B T \left(\frac{\partial ln\Xi}{\partial \mu}\right)_{V,T}. \tag{10.20}$$

The pressure is determined as

$$p = k_B T \left(\frac{\partial ln\Xi}{\partial V}\right)_{\mu,T}. \tag{10.21}$$

In the pressure canonical ensemble we employ with Gibbs' free energy as the potential. It is related to Helmholtz' free energy via $G = F + pV$. The partition function is $\Delta(N, p, T)$, with temperature, pressure and the number of particles as variables. The pressure and volume term can be replaced by other external potentials, such as an external magnetic field (or a gravitational field) which performs work on the system. Gibbs' free energy reads

$$G = -k_B T ln\Delta, \tag{10.22}$$

and the entropy is given by

$$S = k_B ln\Delta + k_B T \left(\frac{\partial ln\Delta}{\partial T}\right)_{p,N}. \tag{10.23}$$

We can compute the volume as

$$V = -k_B T \left(\frac{\partial ln\Delta}{\partial p}\right)_{N,T}, \tag{10.24}$$

and finally the chemical potential

$$\mu = -k_B T \left(\frac{\partial ln\Delta}{\partial N}\right)_{p,T}. \tag{10.25}$$

In this chapter we work with the canonical ensemble only.

10.3 Ising model and phase transitions in magnetic systems

10.3.1 Theoretical background

The model we will employ in our studies of phase transitions at finite temperature for magnetic systems is the so-called Ising model. In its simplest form the energy is expressed as

$$E = -J \sum_{<kl>}^{N} s_k s_l - \mathcal{B} \sum_{k}^{N} s_k, \tag{10.26}$$

with $s_k = \pm 1$, N is the total number of spins, J is a coupling constant expressing the strength of the interaction between neighboring spins and $\mathcal{B}$ is an external magnetic field interacting with the magnetic moment set up by the spins. The symbol $< kl >$ indicates that we sum over nearest neighbors only. Notice that for $J > 0$ it is energetically favorable for neighboring spins to be aligned. This feature leads to, at low enough temperatures, a cooperative phenomenon called spontaneous magnetization. That is, through interactions between nearest neighbors, a given magnetic moment can influence the alignment of spins that are separated from the given spin by a macroscopic distance. These long range correlations between spins are associated with a long-range order in which the lattice has a net magnetization in the absence of a magnetic field. In our further studies of the Ising model, we will mostly limit the attention to cases with $\mathcal{B} = 0$ only.

In order to calculate expectation values such as the mean energy $\langle E \rangle$ or magnetization $\langle \mathcal{M} \rangle$ in statistical physics at a given temperature, we need a probability distribution

$$P_i(\beta) = \frac{e^{-\beta E_i}}{Z} \tag{10.27}$$

with $\beta = 1/kT$ being the inverse temperature, k the Boltzmann constant, E_i is the energy of a state i while Z is the partition function for the canonical ensemble defined as

$$Z = \sum_{i=1}^{M} e^{-\beta E_i}, \tag{10.28}$$

where the sum extends over all microstates M. P_i expresses the probability of finding the system in a given configuration i.

The energy for a specific configuration i is given by

$$E_i = -J \sum_{<kl>}^{N} s_k s_l. \tag{10.29}$$

To better understand what is meant with a configuration, consider first the case of the one-dimensional Ising model with $\mathcal{B} = 0$. In general, a given configuration of N spins in one dimension may look like

$$\begin{array}{ccccccccccc} \uparrow & \uparrow & \uparrow & \dots & \uparrow & \downarrow & \uparrow & \dots & \uparrow & \downarrow \\ 1 & 2 & 3 & \dots & i-1 & i & i+1 & \dots & N-1 & N \end{array}$$

In order to illustrate these features let us further specialize to just two spins.

With two spins, since each spin takes two values only, it means that in total we have $2^2 = 4$ possible arrangements of the two spins. These four possibilities are

$$1 = \uparrow\uparrow \qquad 2 = \uparrow\downarrow \qquad 3 = \downarrow\uparrow \qquad 4 = \downarrow\downarrow$$

What is the energy of each of these configurations?

For small systems, the way we treat the ends matters. Two cases are often used.

1. In the first case we employ what is called free ends. For the one-dimensional case, the energy is then written as a sum over a single index

$$E_i = -J \sum_{j=1}^{N-1} s_j s_{j+1}, \tag{10.30}$$

If we label the first spin as s_1 and the second as s_2 we obtain the following expression for the energy

$$E = -J s_1 s_2. \tag{10.31}$$

The calculation of the energy for the one-dimensional lattice with free ends for one specific spin-configuration can easily be implemented in the following lines

```
for ( j=1; j < N; j++) {
    energy += spin[j]*spin[j+1];
}
```

where the vector $spin[]$ contains the spin value $s_k = \pm 1$. For the specific state E_1, we have chosen all spins up. The energy of this configuration becomes then

$$E_1 = E_{\uparrow\uparrow} = -J.$$

The other configurations give

$$E_2 = E_{\uparrow\downarrow} = +J,$$

$$E_3 = E_{\downarrow\uparrow} = +J,$$

and

$$E_4 = E_{\downarrow\downarrow} = -J.$$

2. We can also choose so-called periodic boundary conditions. This means that if $i = N$, we set the spin number to $i = 1$. In this case the energy for the one-dimensional lattice reads

$$E_i = -J \sum_{j=1}^{N} s_j s_{j+1}, \tag{10.32}$$

and we obtain the following expression for the two-spin case

$$E = -J(s_1 s_2 + s_2 s_1). \tag{10.33}$$

In this case the energy for E_1 is different, we obtain namely

$$E_1 = E_{\uparrow\uparrow} = -2J.$$

The other cases do also differ and we have

$$E_2 = E_{\uparrow\downarrow} = +2J,$$

$$E_3 = E_{\downarrow\uparrow} = +2J,$$

and

$$E_4 = E_{\downarrow\downarrow} = -2J.$$

If we choose to use periodic boundary conditions we can code the above expression as

```
jm=N;
for ( j=1; j <=N ; j++) {
    energy += spin[j]*spin[jm];
    jm = j ;
}
```

Table 10.2: Energy and magnetization for the one-dimensional Ising model with $N = 2$ spins with free ends (FE) and periodic boundary conditions (PBC).

State	Energy (FE)	Energy (PBC)	Magnetization
$1 = \uparrow\uparrow$	$-J$	$-2J$	2
$2 = \uparrow\downarrow$	J	$2J$	0
$3 = \downarrow\uparrow$	J	$2J$	0
$4 = \downarrow\downarrow$	$-J$	$-2J$	-2

Table 10.3: Degeneracy, energy and magnetization for the one-dimensional Ising model with $N = 2$ spins with free ends (FE) and periodic boundary conditions (PBC).

Number spins up	Degeneracy	Energy (FE)	Energy (PBC)	Magnetization
2	1	$-J$	$-2J$	2
1	2	J	$2J$	0
0	1	$-J$	$-2J$	-2

The magnetization is however the same, defined as

$$\mathcal{M}_i = \sum_{j=1}^{N} s_j, \tag{10.34}$$

where we sum over all spins for a given configuration i.

Table 10.2 lists the energy and magnetization for both free ends and periodic boundary conditions.

We can reorganize Table 10.2 according to the number of spins pointing up, as shown in Table 10.3. It is worth noting that for small dimensions of the lattice, the energy differs depending on whether we use periodic boundary conditions or free ends. This means also that the partition functions will be different, as discussed below. In the thermodynamic limit however, $N \to \infty$, the final results do not depend on the kind of boundary conditions we choose.

For a one-dimensional lattice with periodic boundary conditions, each spin sees two neighbors. For a two-dimensional lattice each spin sees four neighboring spins. How many neighbors does a spin see in three dimensions?

In a similar way, we could enumerate the number of states for a two-dimensional system consisting of two spins, i.e., a 2×2 Ising model on a square lattice with *periodic boundary conditions*. In this case we have a total of $2^4 = 16$ states. Some examples of configurations with their respective energies are listed here

$$E = -8J \quad \begin{array}{cc} \uparrow & \uparrow \\ \uparrow & \uparrow \end{array} \qquad E = 0 \quad \begin{array}{cc} \uparrow & \uparrow \\ \uparrow & \downarrow \end{array} \qquad E = 0 \quad \begin{array}{cc} \downarrow & \downarrow \\ \uparrow & \downarrow \end{array} \qquad E = -8J \quad \begin{array}{cc} \downarrow & \downarrow \\ \downarrow & \downarrow \end{array}$$

In the Table 10.4 we group these configurations according to their total energy and magnetization.

Exercise 10.1
Convince yourself that the values listed in Table 10.4 are correct.

Table 10.4: Energy and magnetization for the two-dimensional Ising model with $N = 2 \times 2$ spins with periodic boundary conditions.

Number spins up	Degeneracy	Energy	Magnetization
4	1	$-8J$	4
3	4	0	2
2	4	0	0
2	2	$8J$	0
1	4	0	-2
0	1	$-8J$	-4

For the one-dimensional Ising model we can compute rather easily the exact partition function for a system of N spins. Let us consider first the case with free ends. The energy reads

$$E = -J \sum_{j=1}^{N-1} s_j s_{j+1}.$$

The partition function for N spins is given by

$$Z_N = \sum_{s_1=\pm 1} \cdots \sum_{s_N=\pm 1} \exp\left(\beta J \sum_{j=1}^{N-1} s_j s_{j+1}\right), \tag{10.35}$$

and since the last spin occurs only once in the last sum in the exponential, we can single out the last spin as follows

$$\sum_{s_N=\pm 1} \exp\left(\beta J s_{N-1} s_N\right) = 2cosh(\beta J). \tag{10.36}$$

The partition function consists then of a part from the last spin and one from the remaining spins resulting in

$$Z_N = Z_{N-1} 2cosh(\beta J). \tag{10.37}$$

We can repeat this process and obtain

$$Z_N = (2cosh(\beta J))^{N-2} Z_2, \tag{10.38}$$

with Z_2 given by

$$Z_2 = \sum_{s_1=\pm 1} \sum_{s_2=\pm 1} \exp\left(\beta J s_1 s_2\right) = 4cosh(\beta J), \tag{10.39}$$

resulting in

$$Z_N = 2(2cosh(\beta J))^{N-1}. \tag{10.40}$$

In the thermodynamical limit where we let $N \to \infty$, the way we treat the ends does not matter. However, since our computations will always be carried out with a limited value of N, we need to consider other boundary conditions as well. Here we limit the attention to periodic boundary conditions.

If we use periodic boundary conditions, the partition function is given by

$$Z_N = \sum_{s_1=\pm 1} \cdots \sum_{s_N=\pm 1} \exp\left(\beta J \sum_{j=1}^{N} s_j s_{j+1}\right), \tag{10.41}$$

where the sum in the exponential runs from 1 to N since the energy is defined as

$$E = -J \sum_{j=1}^{N} s_j s_{j+1}.$$

We can then rewrite the partition function as

$$Z_N = \sum_{\{s_i=\pm 1\}} \prod_{i=1}^{N} \exp\left(\beta J s_i s_{i+1}\right), \tag{10.42}$$

where the first sum is meant to represent all lattice sites. Introducing the matrix $\hat{\mathbf{T}}$ (the so-called transfer matrix)

$$\hat{\mathbf{T}} = \begin{pmatrix} e^{\beta J} & e^{-\beta J} \\ e^{-\beta J} & e^{\beta J} \end{pmatrix}, \tag{10.43}$$

with matrix elements $t_{11} = e^{\beta J}$, $t_{1-1} = e^{-\beta J}$, $t_{-11} = e^{\beta J}$ and $t_{-1-1} = e^{\beta J}$ we can rewrite the partition function as

$$Z_N = \sum_{\{s_i=\pm 1\}} \hat{\mathbf{T}}_{s_1 s_2} \hat{\mathbf{T}}_{s_2 s_3} \dots \hat{\mathbf{T}}_{s_N s_1} = Tr \hat{\mathbf{T}}^N. \tag{10.44}$$

The 2×2 matrix $\hat{\mathbf{T}}$ is easily diagonalized with eigenvalues $\lambda_1 = 2cosh(\beta J)$ and $\lambda_2 = 2sinh(\beta J)$. Similarly, the matrix $\hat{\mathbf{T}}^N$ has eigenvalues λ_1^N and λ_2^N and the trace of $\hat{\mathbf{T}}^N$ is just the sum over eigenvalues resulting in a partition function

$$Z_N = \lambda_1^N + \lambda_2^N = 2^N \left([cosh(\beta J)]^N + [sinh(\beta J)]^N \right). \tag{10.45}$$

In the limit $N \to \infty$ the two partition functions with free ends and periodic boundary conditions agree, see below for a demonstration.

In the development phase of an algorithm and its pertinent code it is always useful to test the numerics against analytic results. It is therefore instructive to compute properties like the internal energy and the specific heat for these two cases and test the results against those produced by our code. We can then calculate the mean energy with free ends from the above formula for the partition function using

$$\langle E \rangle = -\frac{\partial lnZ}{\partial \beta} = -(N-1) J tanh(\beta J). \tag{10.46}$$

Helmholtz's free energy is given by

$$F = -k_B T ln Z_N = -N k_B T ln \left(2cosh(\beta J)\right). \tag{10.47}$$

If we take our simple system with just two spins in one-dimension, we see immediately that the above expression for the partition function is correct. Using the definition of the partition function we have

$$Z_2 = \sum_{i=1}^{2} e^{-\beta E_i} = 2e^{-\beta J} + 2e^{\beta J} = 4cosh(\beta J) \tag{10.48}$$

If we take the limit $T \to 0$ ($\beta \to \infty$) and set $N = 2$, we obtain

$$\lim_{\beta\to\infty} \langle E \rangle = -J \frac{e^{J\beta} - e^{-J\beta}}{e^{J\beta} + e^{-J\beta}} = -J, \tag{10.49}$$

which is the energy where all spins point in the same direction. At low T, the system tends towards a state with the highest possible degree of order.

The specific heat in one-dimension with free ends is

$$C_V = \frac{1}{kT^2} \frac{\partial^2}{\partial \beta^2} lnZ_N = (N-1)k \left(\frac{\beta J}{cosh(\beta J)} \right)^2. \tag{10.50}$$

Note well that this expression for the specific heat from the one-dimensional Ising model does not diverge or exhibits discontinuities, as can be seen from Fig. 10.3.

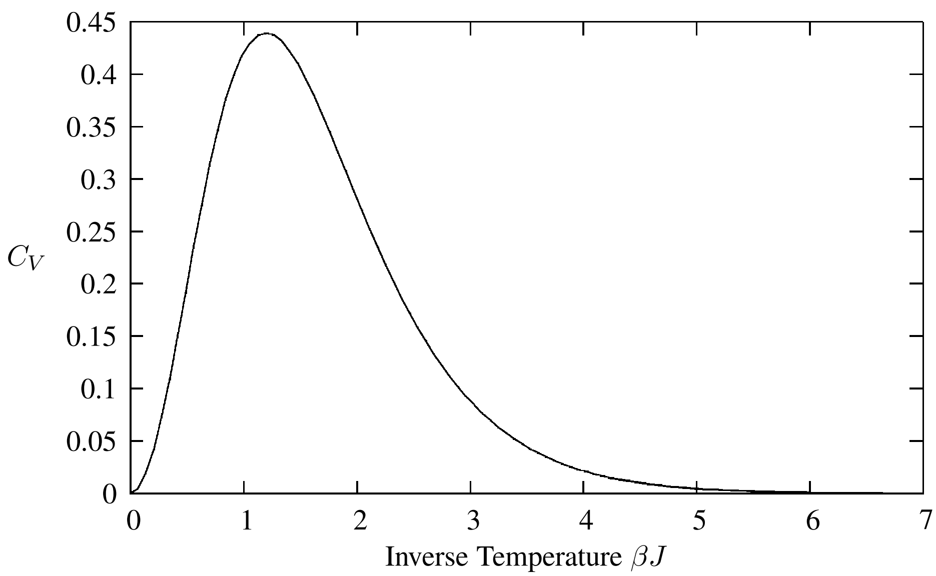


Figure 10.3: Heat capacity per spin ($C_V/(N-1)k_B$ as function of inverse temperature β for the one-dimensional Ising model.

In one dimension we do not have a second order phase transition, although this is predicted by mean field models [52].

We can repeat this exercise for the case with periodic boundary conditions as well. Helmholtz's free energy is in this case

$$F = -k_B T ln(\lambda_1^N + \lambda_2^N) = -k_B T \left\{ N ln(\lambda_1) + ln \left(1 + (\frac{\lambda_2}{\lambda_1})^N \right) \right\}, \tag{10.51}$$

which in the limit $N \to \infty$ results in $F = -k_B T N ln(\lambda_1)$ as in the case with free ends. Since other thermodynamical quantities are related to derivatives of the free energy, all observables become identical in the thermodynamic limit.

Exercise 10.2
Calculate the internal energy and heat capacity of the one-dimensional Ising model using periodic boundary conditions and compare the results with those for free ends in the limit $N \to \infty$.

Hitherto we have limited ourselves to studies of systems with zero external magnetic field, viz $\mathcal{B} = 0$. We will mostly study systems which exhibit a spontaneous magnitization. It is however instructive to extend the one-dimensional Ising model to $\mathcal{B} \neq \prime$, yielding a partition function (with periodic boundary conditions)

$$Z_N = \sum_{s_1=\pm 1} \cdots \sum_{s_N=\pm 1} \exp{(\beta \sum_{j=1}^{N} (J s_j s_{j+1} + \frac{\mathcal{B}}{2}(s_j + s_{j+1}))}, \tag{10.52}$$

which yields a new transfer matrix with matrix elements $t_{11} = e^{\beta(J+\mathcal{B})}$, $t_{1-1} = e^{-\beta J}$, $t_{-11} = e^{\beta J}$ and $t_{-1-1} = e^{\beta(J-\mathcal{B})}$ with eigenvalues

$$\lambda_1 = e^{\beta J} cosh(\beta J) + (e^{2\beta J} sinh^2(\beta \mathcal{B}) + e^{-2\beta J})^{1/2}, \tag{10.53}$$

and

$$\lambda_1 = e^{\beta J} cosh(\beta J) - (e^{2\beta J} sinh^2(\beta \mathcal{B}) + e^{-2\beta J})^{1/2}. \tag{10.54}$$

The partition function is given by $Z_N = \lambda_1^N + \lambda_2^N$ and in the thermodynamic limit we obtain the following free energy

$$F = -N k_B T ln \left(e^{\beta J} cosh(\beta J) + (e^{2\beta J} sinh^2(\beta \mathcal{B}) + e^{-2\beta J})^{1/2} \right). \tag{10.55}$$

It is now useful to compute the expectation value of the magnetisation per spin

$$\langle \mathcal{M}/N \rangle = \frac{1}{NZ} \sum_{i}^{M} \mathcal{M}_i e^{-\beta E_i} = -\frac{1}{N} \frac{\partial F}{\partial \mathcal{B}}, \tag{10.56}$$

resulting in

$$\langle \mathcal{M}/N \rangle = \frac{sinh(\beta \mathcal{B})}{\left(sinh^2(\beta \mathcal{B}) + e^{-2\beta J})^{1/2} \right)}. \tag{10.57}$$

We see that for $\mathcal{B} = 0$ the magnetisation is zero. This means that for a one-dimensional Ising model we cannot have a spontaneous magnetization. For the two-dimensional model however, see the discussion below, the Ising model exhibits both a spontaneous magnetisation and a specific heat and susceptibility which are discontinuous or even diverge. However, except for the simplest case such as 2×2 lattice of spins, with the following partition function

$$Z = 2e^{-8J\beta} + 2e^{8J\beta} + 12, \tag{10.58}$$

and resulting mean energy

$$\langle E \rangle = -\frac{1}{Z} \left(16 e^{8J\beta} - 16 e^{-8J\beta} \right), \tag{10.59}$$

it is a highly non-trivial task to find the analytic expression for Z_N in the thermodynamic limit. The analytic expression for the Ising model in two dimensions was obtained in 1944 by the Norwegian chemist

Lars Onsager [53]. The exact partition function for N spins in two dimensions and with zero magnetic field $\mathcal{B}$ is given by

$$Z_N = \left[2cosh(\beta J)e^I\right]^N, \tag{10.60}$$

with

$$I = \frac{1}{2\pi}\int_0^{\pi} d\phi ln\left[\frac{1}{2}\left(1+(1-\kappa^2 sin^2\phi)^{1/2}\right)\right], \tag{10.61}$$

and

$$\kappa = 2sinh(2\beta J)/cosh^2(2\beta J). \tag{10.62}$$

The resulting energy is given by

$$\langle E\rangle = -Jcoth(2\beta J)\left[1+\frac{2}{\pi}(2tanh^2(2\beta J)-1)K_1(q)\right], \tag{10.63}$$

with $q = 2sinh(2\beta J)/cosh^2(2\beta J)$ and the complete elliptic integral of the first kind

$$K_1(q) = \int_0^{\pi/2}\frac{d\phi}{\sqrt{1-q^2 sin^2\phi}}. \tag{10.64}$$

Differentiating once more with respect to temperature we obtain the specific heat given by

$$C_V = \frac{4k_B}{\pi}(\beta Jcoth(2\beta J))^2\left\{K_1(q)-K_2(q)-(1-tanh^2(2\beta J))\left[\frac{\pi}{2}+(2tanh^2(2\beta J)-1)K_1(q)\right]\right\}, \tag{10.65}$$

with

$$K_2(q) = \int_0^{\pi/2} d\phi\sqrt{1-q^2 sin^2\phi}. \tag{10.66}$$

is the complete elliptic integral of the second kind. Near the critical temperature T_C the specific heat behaves as

$$C_V \approx -\frac{2}{\pi}\left(\frac{2J}{k_B T_C}\right)^2 ln\left|1-\frac{T}{T_C}\right| + \text{const}. \tag{10.67}$$

In theories of critical phenomena one has that

$$C_V \sim \left|1-\frac{T}{T_C}\right|^{-\alpha}, \tag{10.68}$$

and Onsager's result is a special case of this power law behavior. The limiting form of the function

$$lim_{\alpha\to 0}\frac{1}{\alpha}(Y^{-\alpha}-1) = -lnY, \tag{10.69}$$

meaning that the analytic result is a special case of the power law singularity with $\alpha = 0$. To compute the spontaneous magnetisation per spin is also highly non-trivial. Here we list the result

$$\langle\mathcal{M}(T)/N\rangle = \left[1-\frac{(1-tanh^2(\beta J))^4}{16tanh^4(\beta J)}\right]^{1/8}, \tag{10.70}$$

for $T < T_C$ and 0 for $T > T_C$. The behavior is thus as $T \to T_C$ from below

$$\langle\mathcal{M}(T)/N\rangle \sim (T_C - T)^{1/8}. \tag{10.71}$$

The susceptibility behaves as

$$\chi(T) \sim |T_C - T|^{-7/4}. \tag{10.72}$$

10.3.2 Phase Transitions

The Ising model in two dimensions and with $\mathcal{B} = 0$ undergoes a phase transition of second order. What it actually means is that below a given critical temperature T_C, the Ising model exhibits a spontaneous magnetization with $\langle \mathcal{M} \rangle \neq 0$. Above T_C the average magnetization is zero. The one-dimensional Ising model does not predict any spontaneous magnetization at any finite temperature. The physical reason for this can be understood from the following simple consideration. Assume that the ground state for an N-spin system in one dimension is characterized by the following configuration

$$\begin{array}{ccccccccccc} \uparrow & \uparrow & \uparrow & \dots & \uparrow & \uparrow & \uparrow & \dots & \uparrow & \uparrow \\ 1 & 2 & 3 & \dots & i-1 & i & i+1 & \dots & N-1 & N \end{array}$$

which has a total energy $-NJ$ and magnetization N, where we used periodic boundary conditions. If we flip half of the spins we arrive and special to a configuration where the first half of the spins point upwards and last haf points downwards we arrive at the configuration

$$\begin{array}{cccccccccc} \uparrow & \uparrow & \uparrow & \dots & \uparrow & \uparrow & \downarrow & \dots & \downarrow & \downarrow \\ 1 & 2 & 3 & \dots & N/2-1 & N/2 & N/2+1 & \dots & N-1 & N \end{array}$$

with energy $(-N+4)J$ and net magnetization zero. This state is an example of a possible disordered state with net magnetization zero. The change in energy is however too small to stabilize the disordered state. There are many other such states with net magnetization zero with energies slightly larger than the above case. But it serves to demonstrate our point, we can namely build states at low energies compared with the ordered state with net magnetization zero. And the energy difference between the ground state is too small to stabilize the system. In two dimensions however the excitation energy to a disordered state is much higher, and this difference can be sufficient to stabilize the system. In fact, the Ising model exhibits a phase transition to a disordered phase both in two and three dimensions.

For the two-dimensional case, we move from a phase with finite magnetization $\langle \mathcal{M} \rangle \neq 0$ to a paramagnetic phase with $\langle \mathcal{M} \rangle = 0$ at a critical temperature T_C. At the critical temperature, quantities like the heat capacity C_V and the susceptibility χ are discontinuous or diverge at the critical point in the thermodynamic limit, i.e., with an infinitely large lattice. This means that the variance in energy and magnetization are discontinuous or diverge. For a finite lattice however, the variance will always scale as $\sim 1/\sqrt{M}$, M being e.g., the number of configurations which in our case is proportional with L, the number of spins in a the x and y directions. The total number of spins is $N = L \times L$ resulting in a total of $M = 2^N$ microstates. Since our lattices will always be of a finite dimensions, the calculated C_V or χ will not exhibit a diverging behavior. We will however notice a broad maximum in e.g., C_V near T_C. This maximum, as discussed below, becomes sharper and sharper as L is increased.

Near T_C we can characterize the behavior of many physical quantities by a power law behavior. As an example, we demonstrated in the previous section that the mean magnetization is given by

$$\langle \mathcal{M}(T) \rangle \sim (T - T_C)^{\beta}, \tag{10.73}$$

where $\beta = 1/8$ is a so-called critical exponent. A similar relation applies to the heat capacity

$$C_V(T) \sim |T_C - T|^{-\gamma}, \tag{10.74}$$

and the susceptibility

$$\chi(T) \sim |T_C - T|^{-\alpha}, \tag{10.75}$$

with $\alpha = 0$ and $\gamma = -7/4$. Another important quantity is the correlation length, which is expected to be of the order of the lattice spacing for $T >> T_C$. Because the spins become more and more correlated as T approaches T_C, the correlation length increases as we get closer to the critical temperature. The discontinuous behavior of ξ near T_C is

$$\xi(T) \sim |T_C - T|^{-\nu}. \tag{10.76}$$

A second-order phase transition is characterized by a correlation length which spans the whole system. Since we are always limited to a finite lattice, ξ will be proportional with the size of the lattice.

Through finite size scaling relations [54–56] it is possible to relate the behavior at finite lattices with the results for an infinitely large lattice. The critical temperature scales then as

$$T_C(L) - T_C(L = \infty) \sim aL^{-1/\nu}, \tag{10.77}$$

with a a constant and ν defined in Eq. (10.76). The correlation length is given by

$$\xi(T) \sim L \sim |T_C - T|^{-\nu}. \tag{10.78}$$

and if we set $T = T_C$ one obtains

$$\langle \mathcal{M}(T) \rangle \sim (T - T_C)^{\beta} \rightarrow L^{-\beta/\nu}, \tag{10.79}$$

$$C_V(T) \sim |T_C - T|^{-\gamma} \rightarrow L^{\alpha/\nu}, \tag{10.80}$$

and

$$\chi(T) \sim |T_C - T|^{-\alpha} \rightarrow L^{\gamma/\nu}. \tag{10.81}$$

10.4 The Metropolis algorithm and the two-dimensional Ising Model

The algorithm of choice for solving the Ising model is the approach proposed by Metropolis *et al.* [57] in 1953. As discussed in chapter 9, new configurations are generated from a previous one using a transition probability which depends on the energy difference between the initial and final states.

In our case we have as the Monte Carlo sampling function the probability for finding the system in a state s given by

$$P_s = \frac{e^{-(\beta E_s)}}{Z},$$

with energy E_s, $\beta = 1/kT$ and Z is a normalization constant which defines the partition function in the canonical ensemble. As discussed above

$$Z(\beta) = \sum_s e^{-(\beta E_s)}$$

is difficult to compute since we need all states. In a calculation of the Ising model in two dimensions, the number of configurations is given by 2^N with $N = L \times L$ the number of spins for a lattice of length L. Fortunately, the Metropolis algorithm considers only ratios between probabilities and we do not need to compute the partition function at all. The algorithm goes as follows

1. Establish an initial state with energy E_b by positioning yourself at a random position in the lattice

2. Change the initial configuration by flipping e.g., one spin only. Compute the energy of this trial state E_t.

3. Calculate $\Delta E = E_t - E_b$. The number of values ΔE is limited to five for the Ising model in two dimensions, see the discussion below.

4. If $\Delta E \leq 0$ we accept the new configuration, meaning that the energy is lowered and we are hopefully moving towards the energy minimum at a given temperature. Go to step 7.

5. If $\Delta E > 0$, calculate $w = e^{-(\beta \Delta E)}$.

6. Compare w with a random number r. If
$$r \leq w,$$
then accept the new configuration, else we keep the old configuration.

7. The next step is to update various expectations values.

8. The steps (2)-(7) are then repeated in order to obtain a sufficently good representation of states.

9. Each time you sweep through the lattice, i.e., when you have summed over all spins, constitutes what is called a Monte Carlo cycle. You could think of one such cycle as a measurement. At the end, you should divide the various expectation values with the total number of cycles. You can choose whether you wish to divide by the number of spins or not. If you divide with the number of spins as well, your result for e.g., the energy is now the energy per spin.

The crucial step is the calculation of the energy difference and the change in magnetization. This part needs to be coded in an as efficient as possible way since the change in energy is computed many times. In the calculation of the energy difference from one spin configuration to the other, we will limit the change to the flipping of one spin only. For the Ising model in two dimensions it means that there will only be a limited set of values for ΔE. Actually, there are only five possible values. To see this, select first a random spin position x, y and assume that this spin and its nearest neighbors are all pointing up. The energy for this configuration is $E = -4J$. Now we flip this spin as shown below. The energy of the new configuration is $E = 4J$, yielding $\Delta E = 8J$.

$$E = -4J \qquad \begin{array}{ccc} & \uparrow & \\ \uparrow & \uparrow & \uparrow \\ & \uparrow & \end{array} \qquad \Longrightarrow \qquad E = 4J \qquad \begin{array}{ccc} & \uparrow & \\ \uparrow & \downarrow & \uparrow \\ & \uparrow & \end{array}$$

The four other possibilities are as follows

$$E = -2J \qquad \begin{array}{ccc} & \uparrow & \\ \downarrow & \uparrow & \uparrow \\ & \uparrow & \end{array} \qquad \Longrightarrow \qquad E = 2J \qquad \begin{array}{ccc} & \uparrow & \\ \downarrow & \downarrow & \uparrow \\ & \uparrow & \end{array}$$

with $\Delta E = 4J$,

$$E = 0 \qquad \begin{array}{ccc} & \uparrow & \\ \downarrow & \uparrow & \uparrow \\ & \downarrow & \end{array} \qquad \Longrightarrow \qquad E = 0 \qquad \begin{array}{ccc} & \uparrow & \\ \downarrow & \downarrow & \uparrow \\ & \downarrow & \end{array}$$

with $\Delta E = 0$,

$$E = 2J \qquad \begin{array}{ccc} & \downarrow & \\ \downarrow & \uparrow & \uparrow \\ & \downarrow & \end{array} \qquad \Longrightarrow \qquad E = -2J \qquad \begin{array}{ccc} & \downarrow & \\ \downarrow & \downarrow & \uparrow \\ & \downarrow & \end{array}$$

with $\Delta E = -4J$ and finally

$$E = 4J \qquad \begin{array}{ccc} & \downarrow & \\ \downarrow & \uparrow & \downarrow \\ & \downarrow & \end{array} \qquad \Longrightarrow \qquad E = -4J \qquad \begin{array}{ccc} & \downarrow & \\ \downarrow & \downarrow & \downarrow \\ & \downarrow & \end{array}$$

with $\Delta E = -8J$. This means in turn that we could construct an array which contains all values of $e^{\beta \Delta E}$ before doing the Metropolis sampling. Else, we would have to evaluate the exponential at each Monte Carlo sampling. For the two-dimensional Ising model there are only five possible values. It is rather easy to convice oneself that for the one-dimensional Ising model we have only three possible values. The main part of the Ising model program is shown here (there is also a corresponding Fortran program).

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter10/cpp/ising_2dim.cpp

```
/*
   Program to solve the two-dimensional Ising model
   The coupling constant J = 1
   Boltzmann's constant = 1, temperature has thus dimension energy
   Metropolis sampling is used. Periodic boundary conditions.
*/
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace  std;
ofstream ofile;
// inline function for periodic boundary conditions
inline int periodic(int i, int limit, int add) {
  return (i+limit+add) % (limit);
}
// Function to read in data from screen
void read_input(int&, int&, double&, double&, double&);
// Function to initialise energy and magnetization
void initialize(int, double, int **, double&, double&);
// The metropolis algorithm
void Metropolis(int, long&, int **, double&, double&, double *);
// prints to file the results of the calculations
void output(int, int, double, double *);

//  main program
int main(int argc, char* argv[])
{
  char *outfilename;
  long idum;
  int **spin_matrix, n_spins, mcs;
  double w[17], average[5], initial_temp, final_temp, E, M, temp_step;

  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
```

```
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  //    Read in initial values such as size of lattice, temp and cycles
  read_input(n_spins, mcs, initial_temp, final_temp, temp_step);
  spin_matrix = (int**) matrix(n_spins, n_spins, sizeof(int));
  idum = -1; // random starting point
  for ( double temp = initial_temp; temp <= final_temp; temp+=temp_step){
    //    initialise energy and magnetization
    E = M = 0.;
    // setup array for possible energy changes
    for( int de =-8; de <= 8; de++) w[de+8] = 0;
    for( int de =-8; de <= 8; de+=4) w[de+8] = exp(-de/temp);
    // initialise array for expectation values
    for( int i = 0; i < 5; i++) average[i] = 0.;
    initialize(n_spins, double temp, spin_matrix, E, M);
    // start Monte Carlo computation
    for (int cycles = 1; cycles <= mcs; cycles++){
      Metropolis(n_spins, idum, spin_matrix, E, M, w);
      // update expectation values
      average[0] += E;    average[1] += E*E;
      average[2] += M;    average[3] += M*M; average[4] += fabs(M);
    }
    // print results
    output(n_spins, mcs, temp, average);
  }
  free_matrix((void **) spin_matrix); // free memory
  ofile.close();  // close output file
  return 0;
}
```

The array $w[17]$ contains values of ΔE spanning from $-8J$ to $8J$ and it is precalculated in the main part for every new temperature. The program takes as input the initial temperature, final temperature, a temperature step, the number of spins in one direction (we force the lattice to be a square lattice, meaning that we have the same number of spins in the x and the y directions) and the number of Monte Carlo cycles. For every Monte Carlo cycle we run through all spins in the lattice in the function metropolis and flip one spin at the time and perform the Metropolis test. However, every time we flip a spin we need to compute the actual energy difference ΔE in order to access the right element of the array which stores $e^{\beta \Delta E}$. This is easily done in the Ising model since we can exploit the fact that only one spin is flipped, meaning in turn that all the remaining spins keep their values fixed. The energy difference between a state E_1 and a state E_2 with zero magnetic field is

$$\Delta E = E_2 - E_1 = J \sum_{<kl>}^{N} s_k^1 s_l^1 - J \sum_{<kl>}^{N} s_k^2 s_l^2, \tag{10.82}$$

which we can rewrite as

$$\Delta E = -J \sum_{<kl>}^{N} s_k^2 (s_l^2 - s_l^1), \tag{10.83}$$

where the sum now runs only over the nearest neighbors k of the spin Since the spin to be flipped takes only two values, $s_l^1 = \pm 1$ and $s_l^2 = \pm 1$, it means that if $s_l^1 = 1$, then $s_l^2 = -1$ and if $s_l^1 = -1$, then

$s_l^2 = 1$. The other spins keep their values, meaning that $s_k^1 = s_k^2$. If $s_l^1 = 1$ we must have $s_l^1 - s_l^2 = 2$, and if $s_l^1 = -1$ we must have $s_l^1 - s_l^2 = -2$. From these results we see that the energy difference can be coded efficiently as

$$\Delta E = 2Js_l^1 \sum_{<k>}^{N} s_k, \tag{10.84}$$

where the sum runs only over the nearest neighbors k of spin l. We can compute the change in magnetisation by flipping one spin as well. Since only spin l is flipped, all the surrounding spins remain unchanged. The difference in magnetisation is therefore only given by the difference $s_l^1 - s_l^2 = \pm 2$, or in a more compact way as

$$M_2 = M_1 + 2s_l^2, \tag{10.85}$$

where M_1 and M_2 are the magnetizations before and after the spin flip, respectively. Eqs. (10.84) and (10.85) are implemented in the function metropolis shown here

```
void Metropolis(int n_spins, long& idum, int **spin_matrix, double& E,
    double&M, double *w)
{
  // loop over all spins
  for(int y =0; y < n_spins; y++) {
    for (int x= 0; x < n_spins; x++){
      // Find random position
      int ix = (int) (ran1(&idum)*(double)n_spins);
      int iy = (int) (ran1(&idum)*(double)n_spins);
      int deltaE =  2*spin_matrix[iy][ix]*
        (spin_matrix[iy][periodic(ix,n_spins,-1)]+
         spin_matrix[periodic(iy,n_spins,-1)][ix] +
         spin_matrix[iy][periodic(ix,n_spins,1)] +
         spin_matrix[periodic(iy,n_spins,1)][ix]);
      // Here we perform the Metropolis test
      if ( ran1(&idum) <= w[deltaE+8] ) {
        spin_matrix[iy][ix] *= -1;  // flip one spin and accept new spin
            config
        // update energy and magnetization
        M += (double) 2*spin_matrix[iy][ix];
        E += (double) deltaE;
      }
    }
  }
} // end of Metropolis sampling over spins
```

Note that we loop over all spins but that we choose the lattice positions x and y randomly. If the move is accepted after performing the Metropolis test, we update the energy and the magnetisation. The new values are used to update the averages computed in the main function.

When setting up the values of the spins it can be useful to have a visualization of the lattice, as shown for the 7×7 lattice of Fig. 10.4.

Another important function is the function initialize. This function sets up the initial energy, magnetisation and spin values for the different lattice positions. The latter sets all spins equal one if the temperature is low, which for the two-dimensional Ising model means temperatures $T < 1.5$. Else, it keeps the value from the preceeding temperature. We have built up a code where we run over a larger temperature span, typically with values $T \in [1.0, 3.0]$.

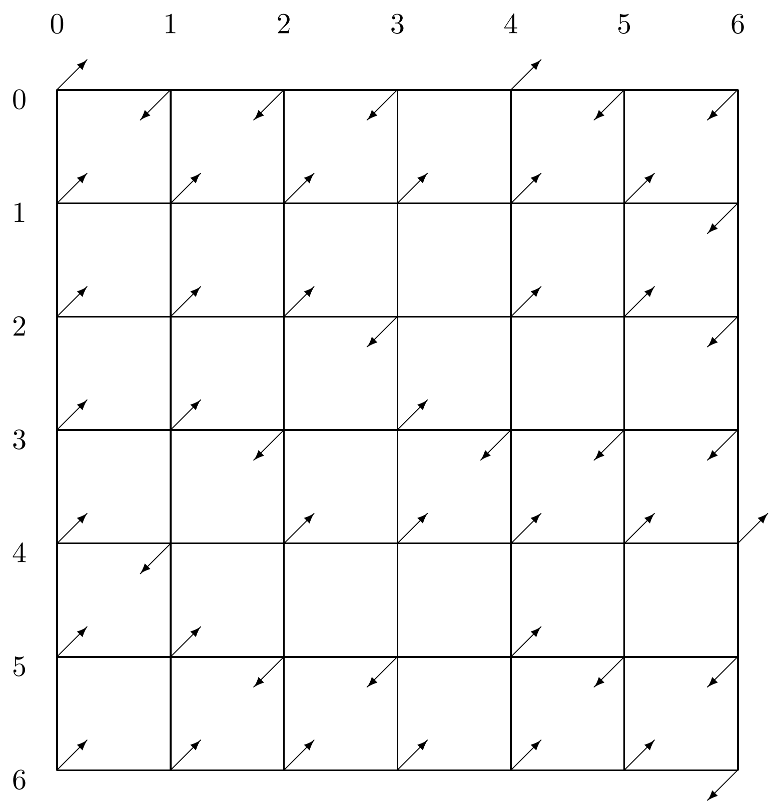


Figure 10.4: Example of a two-dimensional 7×7 lattice with spins pointing either up or down. The variable spin_matrix [0][1] takes the value +1 while spin_matrix [6][0] is -1.

```
// function to initialise energy, spin matrix and magnetization
void initialize(int n_spins, double temp, int **spin_matrix,
                double& E, double& M)
{
  // setup spin matrix and intial magnetization
  for(int y =0; y < n_spins; y++) {
    for (int x= 0; x < n_spins; x++){
      if (temp < 1.5) spin_matrix[y][x] = 1; // spin orientation for the
          ground state
      M +=  (double) spin_matrix[y][x];
    }
  }
  // setup initial energy
  for(int y =0; y < n_spins; y++) {
    for (int x= 0; x < n_spins; x++){
      E -=  (double) spin_matrix[y][x]*
        (spin_matrix[periodic(y,n_spins,-1)][x] +
         spin_matrix[y][periodic(x,n_spins,-1)]);
    }
  }
}// end function initialise
```

In the function output we print the final results, spanning from the mean energy to the susceptibility. Note that we divide by all spins. All the thermodynamical variables we compute are so-called extensive ones meaning that they depend linearly on the number of spins. Since our results will depend on the size of the lattice, we need to divide by the total number of spins in order to see whether quantities like the energy or the heat capacity stabilise or not as functions of increasing lattice size.

```
void output(int n_spins, int mcs, double temp, double *average)
{
  double total_spins  = 1/(n_spins*n_spins);  // divided by total number of
       spins
  double norm = 1/((double) (mcs));  // divided by total number of cycles
  double Eaverage = average[0]*norm;
  double E2average = average[1]*norm;
  double Maverage = average[2]*norm;
  double M2average = average[3]*norm;
  double Mabsaverage = average[4]*norm;
  // all expectation values are per spin, divide by 1/n_spins/n_spins
  double Evariance = (E2average- Eaverage*Eaverage)/total_spins;
  double Mvariance = (M2average - Mabsaverage*Mabsaverage)/total_spins;
  ofile << setiosflags(ios::showpoint | ios::uppercase);
  ofile << setw(15) << setprecision(8) << temp;
  ofile << setw(15) << setprecision(8) << Eaverage/total_spins;
  ofile << setw(15) << setprecision(8) << Evariance/temp/temp;
  ofile << setw(15) << setprecision(8) << Maverage/total_spins;
  ofile << setw(15) << setprecision(8) << Mvariance/temp;
  ofile << setw(15) << setprecision(8) << Mabsaverage/total_spins << endl;
} // end output function
```

10.5 Selected results for the Ising model

In Figs. 10.5-10.8 we display selected results from the program discussed in the previous section. The results have all been obtained with one million Monte Carlo cycles and the Metropolis algorithm for different two-dimensional lattices. A temperature step of $\Delta T = 0.1$ was used for all lattices except the 100×100 results. For the latter we single out a smaller temperature region close to the critical temperature and used $\Delta T = 0.05$. Fig. 10.5 shows the energy to stabilize as function of lattice size. We note that the numerics indicates a smooth and continuous curve for the energy, although there is a larger increase close to the critical temperature $T_C \approx 2.269$.

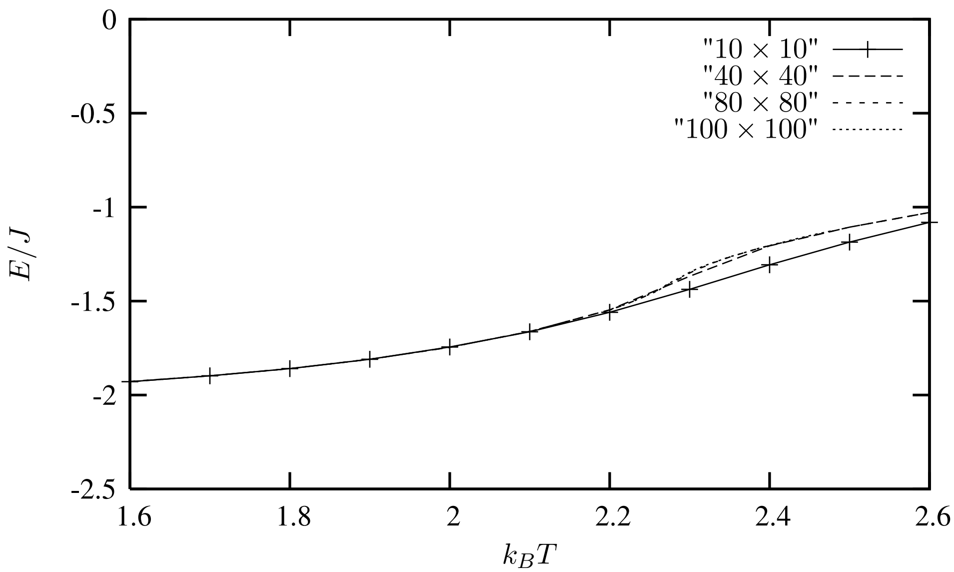


Figure 10.5: Average energy per spin as function of the lattice size for the two-dimensional Ising model.

We mentioned previously that the two-dimensional Ising model with zero external magnetic field exhibits a second-order phase transition and a spontaneous magnetization below T_C. Fig. 10.6 shows the absolute value of the magnetisation as function of the number of spins. We note that with increasing lattice size we approach a steeper line and the transition from a smaller magnetisation to a larger one becomes sharper. This is a possible sign of a phase transition, where we move from a state where all spins (or most of them) align in a specific direction (high degree of order) to a phase where both spin directions are equally probable (high degree of disorder) and result in zero net magnetisation. The ordered phase at low temperatures is called for a ferromagnetic phase while the disordered phase is called the paramagnetic phase, with zero net magnetisation. Since we are plotting the absolute value, our net magnetisation will always be above zero since we are taking the average of a number which is never negative.

The reason we choose to plot the average absolute value instead of the net magnetisation is that slightly below T_C, the net magnetisation may oscillate between negative and positive values since the system, as function of the number of Monte Carlo cycles is likely to have its spins pointing up or down. This means that after a given number of cycles, the net spin may be slightly positive but could then occasionaly jump

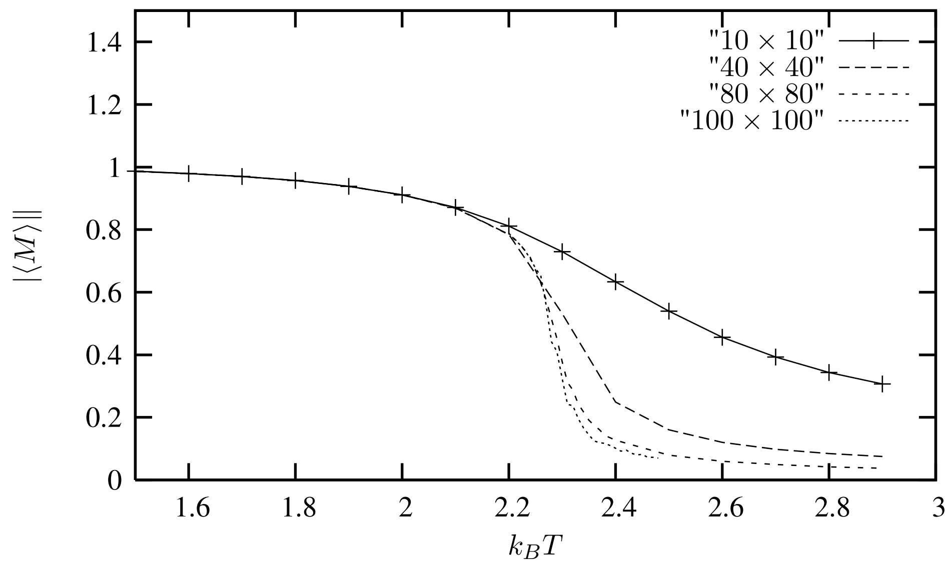


Figure 10.6: Absolute value of the average magnetization per spin as function of the lattice size for the two-dimensional Ising model.

to a negative value and stay there for a given number of Monte Carlo cycles. Above the phase transition the net magnetisation is always zero.

The fact that the system exhibits a spontaneous magnetization (no external field applied) below T_C leads to the definition of the magnetisation as an order parameter. The order parameter is a quantity which is zero on one side of a critical temperature and non-zero on the other side. Since the magnetisation is a continuous quantity at T_C, with the analytic results

$$\left[1 - \frac{(1 - tanh^2(\beta J))^4}{16 tanh^4(\beta J)}\right]^{1/8},$$

for $T < T_C$ and 0 for $T > T_C$, our transition is defined as a continuous one or as a second order phase transition. From Ehrenftest's definition of a phase transition we have that a second order or continuous phase transition exhibits second derivatives of Helmholtz' free energy (the potential in this case) with respect to e.g., temperature that are discontinuous or diverge at T_C. The specific heat for the two-dimensional Ising model exhibits a power-law behavior around T_C with a logarithmic divergence. In Fig. 10.7 we show the corresponding specific heat.

We see from this figure that as the size of the lattice is increased, the specific heat develops a sharper and shaper peak centered around the critical temperature. A similar case happens for the susceptibility, with an even sharper peak, as can be seen from Fig. 10.8.

The Metropolis algorithm is not very efficient close to the critical temperature. Other algorihms such as the heat bath algorithm, the Wolff algorithm and other clustering algorithms, the Swendsen-Wang algorithm, or the multi-histogram method [58, 59] are much more efficient in simulating properties near

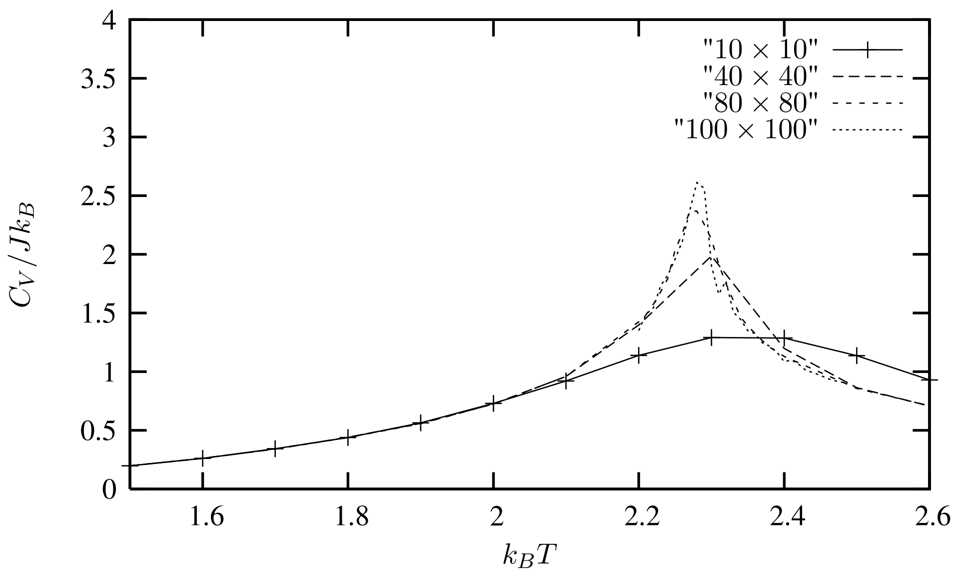


Figure 10.7: Heat capacity per spin as function of the lattice size for the two-dimensional Ising model.

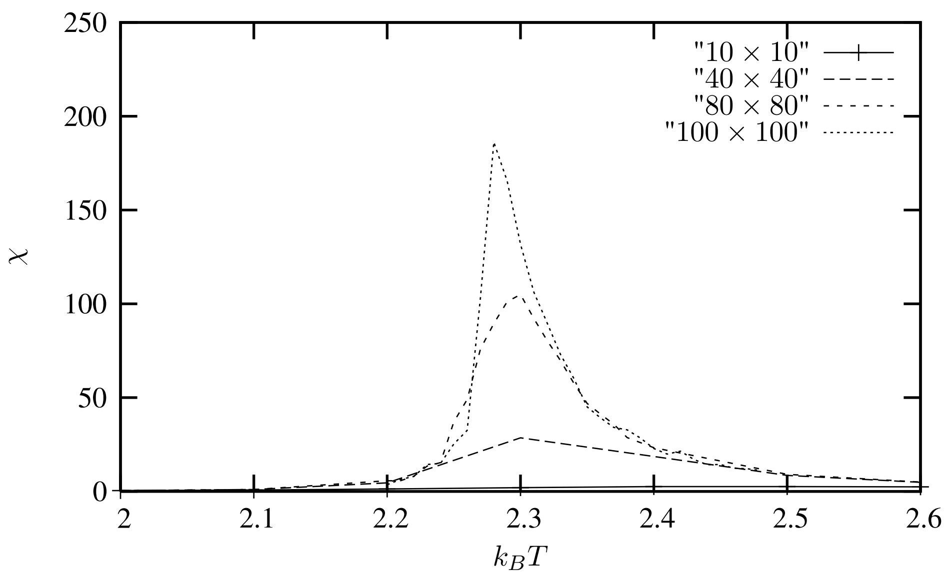


Figure 10.8: Susceptibility per spin as function of the lattice size for the two-dimensional Ising model.

the critical temperature. For spin models like the class of higher-order Potts models discussed in section 10.7, the efficiency of the Metropolis algorithm is simply inadequate. These topics are discussed in depth in the textbooks of Newman and Barkema [56] and Landau and Binder [60] and in chapter 18.

10.6 Correlation functions and further analysis of the Ising model

10.6.1 Thermalization

In the code discussed above we have assumed that one performs a calculation starting with low temperatures, typically well below T_C. For the Ising model this means to start with an ordered configuration. The final set of configurations that define the established equilibrium at a given T, will then be dominated by those configurations where most spins are aligned in one specific direction. For a calculation starting at low T, it makes sense to start with an initial configuration where all spins have the same value, whereas if we were to perform a calculation at high T, for example well above T_C, it would most likely be more meaningful to have a randomly assigned value for the spins. In our code example we use the final spin configuration from a lower temperature to define the initial spin configuration for the next temperature.

In many other cases we may have a limited knowledge on the suitable initial configurations at a given T. This means in turn that if we guess wrongly, we may need a certain number of Monte Carlo cycles before we reach the most likely equilibrium configurations. When equilibrium is established, various observable such as the mean energy and magnetization oscillate around their mean values. A parallel is the particle in the box example discussed in chapter 8. There we considered a box divided into two equal halves separated by a wall. At the beginning, time $t = 0$, there are N particles on the left side. A small hole in the wall is then opened and one particle can pass through the hole per unit time. After some time the system reaches its equilibrium state with equally many particles in both halves, $N/2$. Thereafter, the mean number of particles oscillates around $N/2$.

The number of Monte Carlo cycles needed to reach this equilibrium position is referred to as the thermalization time, or equilibration time t_{eq}. We should then discard the contributions to various expectation values till we have reached equilibrium. How to determine the thermalization time can be done in a brute force way, as demonstrated in Figs. 10.9 and 10.10. In Fig. 10.9 the calculations have been performed with a 40×40 lattice for a temperature $k_BT/J = 2.4$, which corresponds to a case close to a disordered system. We compute the absolute value of the magnetization after each sweep over the lattice. Two starting configurations were used, one with a random orientation of the spins and one with an ordered orientation, the latter corresponding to the ground state of the system. As expected, a disordered configuration as start configuration brings us closer to the average value at the given temperature, while more cycles are needed to reach the steady state with an ordered configuration. Guided by the eye, we could obviously make such plots and discard a given number of samples. However, such a rough guide hides several interesting features. Before we switch to a more detailed analysis, let us also study a case where we start with the 'correct' configuration for the relevant temperature.

Fig. 10.10 displays the absolute value of the mean magnetisation as function of time t for a 100×100 lattice for temperatures $k_BT/J = 1.5$ and $k_BT/J = 2.4$. For the lowest temperature, an ordered start configuration was chosen, while for the temperature close to the critical temperature, a disordered configuration was used. We notice that for the low temperature case the system reaches rather quickly the expected value, while for

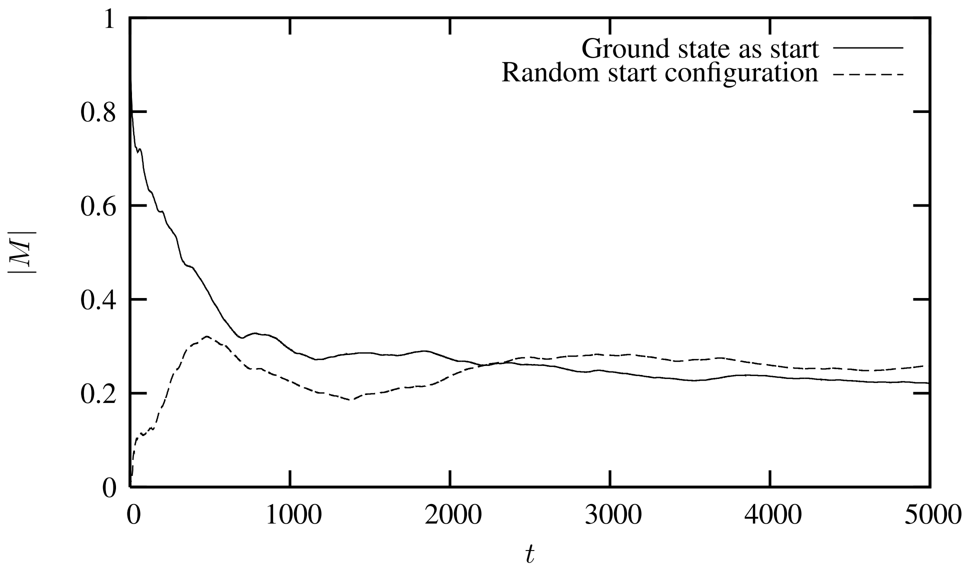


Figure 10.9: Absolute value of the mean magnetisation as function of time t. Time is represented by the number of Monte Carlo cycles. The calculations have been performed with a 40×40 lattice for a temperature $k_B T/J = 2.4$. Two start configurations were used, one with a random orientation of the spins and one with an ordered orientation, which corresponds to the ground state of the system.

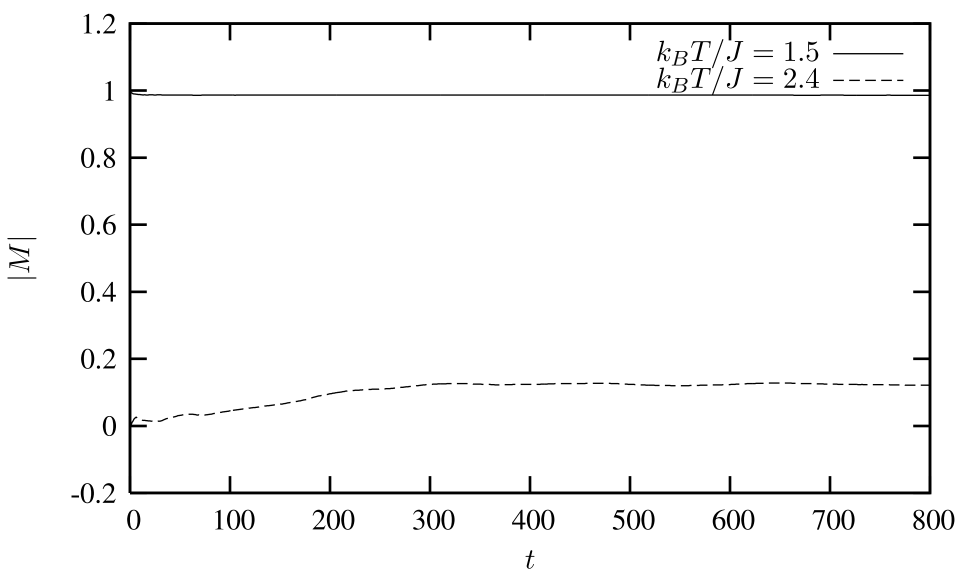


Figure 10.10: Absolute value of the mean magnetisation as function of time t. Time is represented by the number of Monte Carlo cycles. The calculations were performed with a 100×100 lattice for temperatures $k_B T/J = 1.5$ and $k_B T/J = 2.4$. For the lowest temperature, an ordered start configuration was chosen, while for the temperature close to T_C, a disordered configuration was used.

the temperature close to $k_B T_C/J \approx 2.269$ it takes more time to reach the actual steady state.

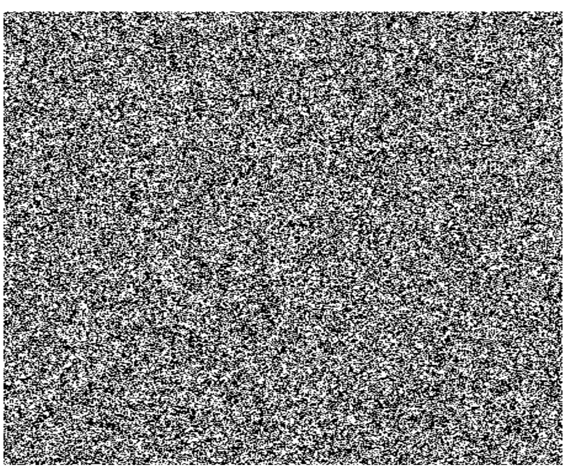
It seems thus that the time needed to reach a steady state is longer for temperatures close to the critical temperature than for temperatures away. In the next subsection we will define more rigorously the equilibration time t_{eq} in terms of the so-called correlation time τ. The correlation time represents the typical time by which the correlation function discussed in the next subsection falls off. There are a number of ways to estimate the correlation time τ. One that is often used is to set it equal the equilibration time $\tau = t_{\text{eq}}$. The correlation time is a measure of how long it takes the system to get from one state to another one that is significantly different from the first. Normally the equilibration time is longer than the correlation time, mainly because two states close to the steady state are more similar in structure than a state far from the steady state.

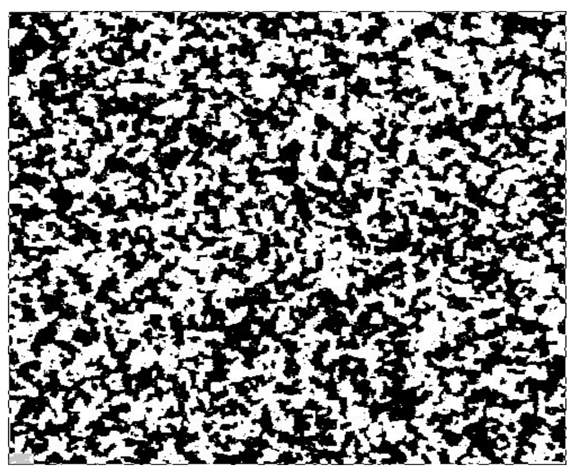
Here we mention also that one can show, using scaling relations [56], that at the critical temperature the correlation time τ relates to the lattice size L as

$$\tau \sim L^{d+z},$$

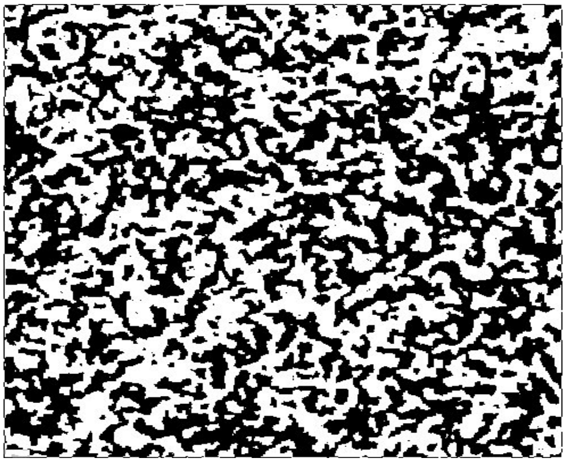
with d the dimensionality of the system. For the Metropolis algorithm based on a single spin-flip process, Nightingale and Blöte obtained $z = 2.1665 \pm 0.0012$ [61]. This is a rather high value, meaning that our algorithm is not the best choice when studying properties of the Ising model near T_C.

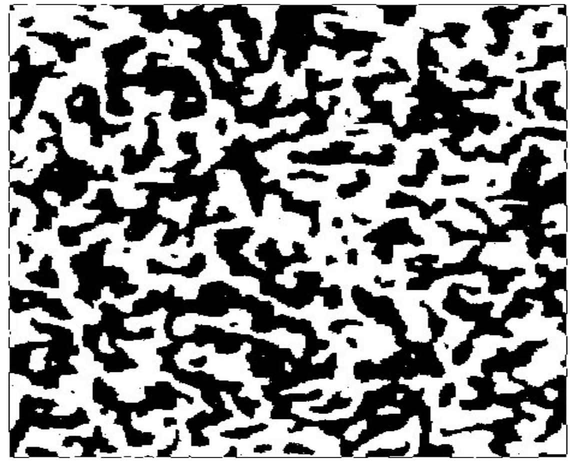
We can understand this behavior by studying the development of the two-dimensional Ising model as function of temperature. The first figure to the left shows the start of a simulation of a 40×40 lattice at a high temperature. Black dots stand for spin down or -1 while white dots represent spin up $(+1)$. As the system cools down, we see in the picture to the right that it starts developing domains with several spins pointing in one particular direction.





Cooling the system further we observe clusters pervading larger areas of the lattice, as seen in the next two pictures. The rightmost picture is the one with T close to the critical temperature. The reason for the large correlation time (and the parameter z) for the single-spin flip Metropolis algorithm is the development of these large domains or clusters with all spins pointing in one direction. It is quite difficult for the algorithm to flip over one of these large domains because it has to do it spin by spin, with each move having a high probability of being rejected due to the ferromagnetic interaction between spins.





Since all spins point in the same direction, the chance of performing the flip

$$E=-4J \qquad \begin{array}{ccc} & \uparrow & \\ \uparrow & \uparrow & \uparrow \\ & \uparrow & \end{array} \qquad \Longrightarrow \qquad E=4J \qquad \begin{array}{ccc} & \uparrow & \\ \uparrow & \downarrow & \uparrow \\ & \uparrow & \end{array}$$

leads to an energy difference of $\Delta E = 8J$. Using the exact critical temperature $k_BT_C/J \approx 2.2.69$, we obtain a probability $\exp -(8/2.269) = 0.029429$ which is rather small. The increase in large correlation times due to increasing lattices can be diminished by using so-called cluster algorithms, such as that introduced by Ulli Wolff in 1989 [62] and the Swendsen-Wang [63] algorithm from 1987. The two-dimensional Ising model with the Wolff or Swendsen-Wang algorithms exhibits a much smaller correlation time, with the variable $z = 0.25 \pm 001$. Here, instead of flipping a single spin, one flips an entire cluster of spins pointing in the same direction. We defer the discussion of these methods to chapter 18.

10.6.2 Time-correlation functions

The so-called time-displacement autocorrelation $\phi(t)$ for the magnetization is given by[1]

$$\phi(t) = \int dt' \left[\mathcal{M}(t') - \langle\mathcal{M}\rangle\right]\left[\mathcal{M}(t'+t) - \langle\mathcal{M}\rangle\right], \tag{10.86}$$

which can be rewritten as

$$\phi(t) = \int dt' \left[\mathcal{M}(t')\mathcal{M}(t'+t) - \langle\mathcal{M}\rangle^2\right], \tag{10.87}$$

where $\langle\mathcal{M}\rangle$ is the average value of the magnetization and $\mathcal{M}(t)$ its instantaneous value. We can discretize this function as follows, where we used our set of computed values $\mathcal{M}(t)$ for a set of discretized times (our Monte Carlo cycles corresponding to a sweep over the lattice)

$$\phi(t) = \frac{1}{t_{\max}-t}\sum_{t'=0}^{t_{\max}-t}\mathcal{M}(t')\mathcal{M}(t'+t) - \frac{1}{t_{\max}-t}\sum_{t'=0}^{t_{\max}-t}\mathcal{M}(t') \times \frac{1}{t_{\max}-t}\sum_{t'=0}^{t_{\max}-t}\mathcal{M}(t'+t). \tag{10.88}$$

One should be careful with times close to $t_{\max}$, the upper limit of the sums becomes small and we end up integrating over a rather small time interval. This means that the statistical error in $\phi(t)$ due to the random nature of the fluctuations in $\mathcal{M}(t)$ can become large. Note also that we could replace the magnetization with the mean energy, or any other expectation values of interest.

[1] We follow closely chapter 3 of Ref. [56].

The time-correlation function for the magnetization gives a measure of the correlation between the magnetization at a time t' and a time $t'+t$. If we multiply the magnetizations at these two different times, we will get a positive contribution if the magnetizations are fluctuating in the same direction, or a negative value if they fluctuate in the opposite direction. If we then integrate over time, or use the discretized version of Eq. (10.88), the time correlation function $\phi(t)$ should take a non-zero value if the fluctuations are correlated, else it should gradually go to zero. For times a long way apart the magnetizations are most likely uncorrelated and $\phi(t)$ should be zero. Fig. 10.11 exhibits the time-correlation function for the magnetization for the same lattice and temperatures discussed in Fig. 10.10.

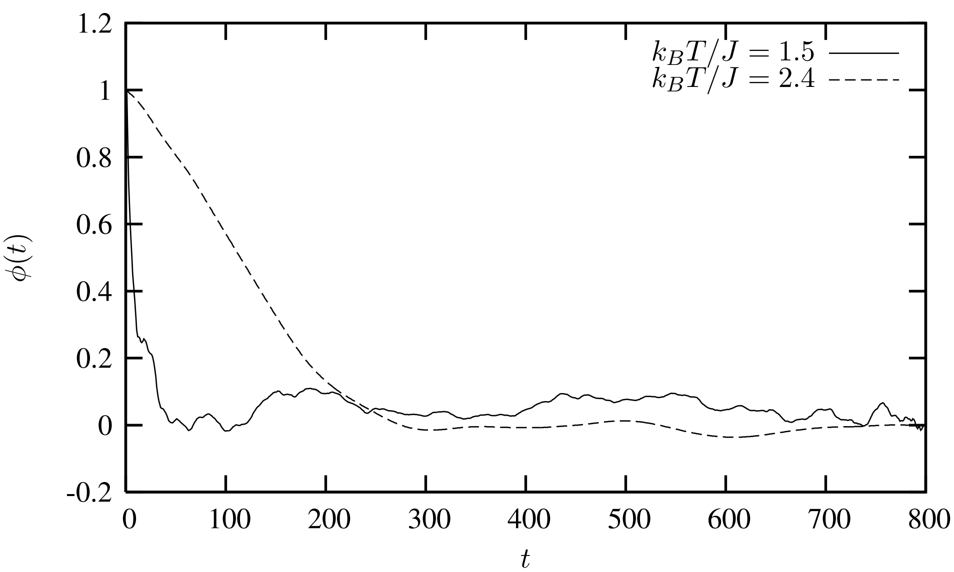


Figure 10.11: Time-autocorrelation function with time t as number of Monte Carlo cycles. It has been normalized with $\phi(0)$. The calculations have been performed for a 100×100 lattice at $k_BT/J = 2.4$ with a disordered state as starting point and at $k_BT/J = 1.5$ with an ordered state as starting point.

We notice that the time needed before $\phi(t)$ reaches zero is $t \sim 300$ for a temperature $k_BT/J = 2.4$. This time is close to the result we found in Fig. 10.10. Similarly, for $k_BT/J = 1.5$ the correlation function reaches zero quickly, in good agreement again with the results of Fig. 10.10. The time-scale, if we can define one, for which the correlation function falls off should in principle give us a measure of the correlation time τ of the simulation.

We can derive the correlation time by observing that our Metropolis algorithm is based on a random walk in the space of all possible spin configurations. We recall from chapter 9 that our probability distribution function $\hat{\mathbf{w}}(t)$ after a given number of time steps t could be written as

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^{\mathbf{t}}\hat{\mathbf{w}}(0),$$

with $\hat{\mathbf{w}}(0)$ the distribution at $t = 0$ and $\hat{\mathbf{W}}$ representing the transition probability matrix. We can always expand $\hat{\mathbf{w}}(0)$ in terms of the right eigenvectors of $\hat{\mathbf{v}}$ of $\hat{\mathbf{W}}$ as

$$\hat{\mathbf{w}}(0) = \sum_i \alpha_i \hat{\mathbf{v}}_i, \tag{10.89}$$

resulting in

$$\hat{\mathbf{w}}(t) = \hat{\mathbf{W}}^t\hat{\mathbf{w}}(0) = \hat{\mathbf{W}}^t\sum_i \alpha_i\hat{\mathbf{v}}_i = \sum_i \lambda_i^t\alpha_i\hat{\mathbf{v}}_i, \tag{10.90}$$

with λ_i the i^{th} eigenvalue corresponding to the eigenvector $\hat{\mathbf{v}}_i$. If we assume that λ_0 is the largest eigenvector we see that in the limit $t \to \infty$, $\hat{\mathbf{w}}(t)$ becomes proportional to the corresponding eigenvector $\hat{\mathbf{v}}_0$. This is our steady state or final distribution.

We can relate this property to an observable like the mean magnetization. With the probabilty $\hat{\mathbf{w}}(t)$ (which in our case is the Boltzmann distribution) we can write the mean magnetization as

$$\langle \mathcal{M}(t)\rangle = \sum_\mu \hat{\mathbf{w}}(t)_\mu \mathcal{M}_\mu, \tag{10.91}$$

or as the scalar of a vector product

$$\langle \mathcal{M}(t)\rangle = \hat{\mathbf{w}}(t)\mathbf{m}, \tag{10.92}$$

with $\mathbf{m}$ being the vector whose elements are the values of $\mathcal{M}_\mu$ in its various microstates μ. We rewrite this relation as

$$\langle \mathcal{M}(t)\rangle = \hat{\mathbf{w}}(t)\mathbf{m} = \sum_i \lambda_i^t\alpha_i\hat{\mathbf{v}}_i\mathbf{m}_i. \tag{10.93}$$

If we define $m_i = \hat{\mathbf{v}}_i\mathbf{m}_i$ as the expectation value of $\mathcal{M}$ in the i^{th} eigenstate we can rewrite the last equation as

$$\langle \mathcal{M}(t)\rangle = \sum_i \lambda_i^t\alpha_i m_i. \tag{10.94}$$

Since we have that in the limit $t \to \infty$ the mean magnetization is dominated by the the largest eigenvalue λ_0, we can rewrite the last equation as

$$\langle \mathcal{M}(t)\rangle = \langle \mathcal{M}(\infty)\rangle + \sum_{i\neq 0} \lambda_i^t\alpha_i m_i. \tag{10.95}$$

We define the quantity

$$\tau_i = -\frac{1}{log\lambda_i}, \tag{10.96}$$

and rewrite the last expectation value as

$$\langle \mathcal{M}(t)\rangle = \langle \mathcal{M}(\infty)\rangle + \sum_{i\neq 0} \alpha_i m_i e^{-t/\tau_i}. \tag{10.97}$$

The quantities τ_i are the correlation times for the system. They control also the auto-correlation function discussed above. The longest correlation time is obviously given by the second largest eigenvalue τ_1, which normally defines the correlation time discussed above. For large times, this is the only correlation time that survives. If higher eigenvalues of the transition matrix are well separated from λ_1 and we simulate long enough, τ_1 may well define the correlation time. In other cases we may not be able to extract a reliable result for τ_1. Coming back to the time correlation function $\phi(t)$ we can present a more general definition in terms of the mean magnetizations $\langle \mathcal{M}(t)\rangle$. Recalling that the mean value is equal to $\langle \mathcal{M}(\infty)\rangle$ we arrive at the expectation values

$$\phi(t) = \langle \mathcal{M}(0) - \mathcal{M}(\infty)\rangle\langle \mathcal{M}(t) - \mathcal{M}(\infty)\rangle, \tag{10.98}$$

and using Eq. (10.97) we arrive at

$$\phi(t) = \sum_{i,j\neq 0} m_i\alpha_i m_j\alpha_j e^{-t/\tau_i}, \tag{10.99}$$

which is appropriate for all times.

10.7 Exercises and projects

Project 10.1: Thermalization and the One-Dimensional Ising Model

In this project we will use the Metropolis algorithm to generate states according to the Boltzmann distribution. Each new configuration is given by the change of only one spin at the time, that is $s_k \to -s_k$. Use periodic boundary conditions and set the magnetic field $\mathcal{B} = 0$.

a) Write a program which simulates the one-dimensional Ising model. Choose $J > 0$, the number of spins $N = 20$, temperature $T = 3$ and the number of Monte Carlo samples $mcs = 100$. Let the initial configuration consist of all spins pointing up, i.e., $s_k = 1$. Compute the mean energy and magnetization for each cycle and find the number of cycles needed where the fluctuation of these variables is negligible. What kind of criterium would you use in order to determine when the fluctuations are negligible?

   Change thereafter the initial condition by letting the spins take random values, either -1 or 1. Compute again the mean energy and magnetization for each cycle and find the number of cycles needed where the fluctuation of these variables is negligible.

   Explain your results.

b) Let $mcs \geq 1000$ and compute $\langle E \rangle$, $\langle E^2 \rangle$ and C_V as functions of T for $0.1 \leq T \leq 5$. Plot the results and compare with the exact ones for periodic boundary conditions.

c) Using the Metropolis sampling method you should now find the number of accepted configurations as function of the total number of Monte Carlo samplings. How does the number of accepted configurations behave as function of temperature T? Explain the results.

d) Compute thereafter the probability $P(E)$ for a system with $N = 50$ at $T = 1$. Choose $mcs \geq 1000$ and plot $P(E)$ as function of E. Count the number of times a specific energy appears and build thereafter up a histogram. What does the histogram mean?

Project 10.2: simulation of the two-dimensional Ising model

a) Assume that the number of spins in the x and y directions are two, viz $L = 2$. Find the analytic expression for the partition function and the corresponding mean values for E, $\mathcal{M}$, the capacity C_V and the suceptibility χ as function of T using periodic boundary conditions.

b) Write your own code for the two-dimensional Ising model with periodic boundary conditions and zero external field $\mathcal{B}$. Set $L = 2$ and compare your numerical results with the analytic ones from the previous exercise. using $T = 0.5$ and $T = 2.5$. How many Monte Carlo cycles do you need before you reach the exact values with an unceertainty less than 1%? What are most likely starting configurations for the spins. Try both an ordered arrangement of the spins and a randomly assigned orientations for both temperature. Analyse the mean energy and magnetisation as functions of the number of Monte Carlo cycles and estimate how many thermalization cycles are needed.

c) We will now study the behavior of the Ising model in two dimensions close to the critical temperature as a function of the lattice size $L \times L$, with L the number of spins in the x and y directions. Calculate the expectation values for $\langle E \rangle$ and $\langle \mathcal{M} \rangle$, the specific heat C_V and the susceptibility χ as functions of T for $L = 10$, $L = 20$, $L = 40$ and $L = 80$ for $T \in [2.0, 2.4]$ with a step in temperature $\Delta T = 0.05$. Plot $\langle E \rangle$, $\langle \mathcal{M} \rangle$, C_V and χ as functions of T. Can you see an indication of a phase transition?

d) Use Eq. (10.77) and the exact result $\nu = 1$ in order to estimate T_C in the thermodynamic limit $L \to \infty$ using your simulations with $L = 10$, $L = 20$, $L = 40$ and $L = 80$.

e) In the remaining part we will use the exact result $kT_C/J = 2/ln(1+\sqrt{2}) \approx 2.269$ and $\nu = 1$. Determine the numerical values of C_V, χ and $\mathcal{M}$ at the exact value $T = T_C$ for $L = 10$, $L = 20$, $L = 40$ and $L = 80$. Plot $log_{10}\,\mathcal{M}$ and χ som funksjon av $log_{10}\,L$ and use the scaling relations of Eqs. (10.79) and (10.81) in order to determine the constants β and γ. Are your log-log plots close to straight lines? The exact values are $\beta = 1/8$ and $\gamma = 7/4$.

f) Make a log-log plot using the results for C_V as function of L for your computations at the exact critical temperature. The specific heat exhibits a logarithmic divergence with $\alpha = 0$, see Eqs. (10.65) and (10.67). Do your results agree with this behavior? Make also a plot of the specific heat computed at the critical temperature for the given lattice.

   The exact specific heats behaves as

$$C_V \approx -\frac{2}{\pi}\left(\frac{2J}{k_B T_C}\right)^2 ln\left|1 - \frac{T}{T_C}\right| + \text{const.}$$

   Comment your results.

Project 10.3: Potts Model

The Potts model has been, in addition to the Ising model, widely used in studies of phase transitions in statistical physics. The so-called two-dimensional q-state Potts model has an energy given by

$$E = -J\sum_{<kl>}^{N}\delta_{s_l,s_k},$$

where the spin s_k at lattice position k can take the values $1, 2, \ldots, q$. The Kroneckr delta function δ_{s_l,s_k} equals unity if the spins are equal and is zero otherwise. N is the total number of spins. For $q = 2$ the Potts model corresponds to the Ising model. To see that we can rewrite the last equation as

$$E = -\frac{J}{2}\sum_{<kl>}^{N} 2(\delta_{s_l,s_k} - \frac{1}{2}) - \sum_{<kl>}^{N}\frac{J}{2}.$$

Now, $2(\delta_{s_l,s_k} - \frac{1}{2})$ is +1 when $s_l = s_k$ and -1 when they are different. This model is thus equivalent to the Ising model except a trivial difference in the energy minimum given by a an additional constant and a factor $J \to J/2$. One of the many applications of the Potts model is to helium absorbed on the surface of graphite.

The Potts model exhibits a second order phase transition for low values of q and a first order transition for larger values of q. Using Eherenfest's definition of a phase transition, a second order phase transition has second derivatives of the free energy that are discontinuous or diverge (the heat capacity and susceptibility in our case) while a first order transition has first derivatives like the mean energy that are discontinuous or diverge. Since the calculations are done with a finite lattice it is always difficult to find the order of the phase transitions. In this project we will limit ourselves to find the temperature region where a phase transition occurs and see if the numerics allows us to extract enough information about the order of the transition.

a) Write a program which simulates the $q = 2$ Potts model for two-dimensional lattices with 10×10, 40×40 and 80×80 spins and compute the average energy and specific heat. Establish an appropriate temperature range for where you see a sudden change in the heat capacity and susceptibility. Make the analysis first for few Monte Carlo cycles and smaller lattices in order to narrow down the region of interest. To get appropriate statistics afterwards you should allow for at least 10^5 Monte Carlo cycles. In setting up this code you need to find an efficient way to simulate the energy differences between different microstates. In doing this you need also to find all possible values of ΔE.

b) Compare these results with those obtained with the two-dimensional Ising model. The exact critical temperature for the Ising model is $T_C = 2.269$. Here you can eventually use the abovementioned program from the lectures or write your own code for the Ising model. Tip when comparing results with the Ising model: remove the constant term. The first step is thus to check that your algorithm for the Potts model gives the same results as the ising model. Note that critical temperature for the $q = 2$ Potts model is half of that for the Ising model.

c) Extend the calculations to the Potts model with $q = 3, 6$ and $q = 10$. Make a table of the possible values of ΔE for each value of q. Establish first the location of the peak in the specific heat and study the behavior of the mean energy and magnetization as functions of q. Do you see a noteworthy change in behavior from the $q = 2$ case? For larger q values you may need lattices of at least 50×50 in size.

   For $q = 3$ and higher you can then proceed as follows:

   - Do a calculation with a small lattice first over a large temperature region. Use typical temperature steps of 0.1.
   - Establish a small region where you see the heat capacity and the susceptibility start to increase.
   - Decrease the temperature step in this region and perform calculations for larger lattices as well.

   For $q = 6$ and $q = 10$ we have a first order phase transition, the energy shows a discontinuity at the critical temperature.

To compute the magnetisation in this case can lead to some preliminary conceptual problems. For the $q = 2$ case we can always assign the values of -1 and $+1$ to the spins. We would then get the same magnetisation as we had with the two-dimensional Ising model. However, we could also assign the value of 0 and 1 to the spins. A simulation could then start with all spins equal 0 at low temperatures. This is then the ordered state. Increasing the temperature and crossing the region where we have the phase transition, both spins value should be equally possible. This means half of the spins take the value 0 and the other half take the value 1, yielding a final magnetisation per spin of $1/2$. The important point is that we see the change in magnetisation when we cross the critical temperature. For higher q values, for example $q = 3$ we could choose something similar to the Ising model. The spins could take the values $-1, 0, 1$. We would again start with an ordered state and let temperature increase. Above T_C all values are equally possible resulting again in a magnetisation equal zero. For the values $0, 1, 2$ the situation would be different. Above T_C, one third has value 0, another third takes the value 1 and the last third is 2, resulting in a net magnetisation per spin equal $0 \times 1/3 + 1 \times 1/3 + 2 \times 1/3 = 1$.

Chapter 11

Quantum Monte Carlo methods

If, in some cataclysm, all scientific knowledge were to be destroyed, and only one sentence passed on to the next generation of creatures, what statement would contain the most information in the fewest words? I believe it is the atomic hypothesis (or atomic fact, or whatever you wish to call it) that all things are made of atoms, little particles that move around in perpetual motion, attracting each other when they are a little distance apart, but repelling upon being squeezed into one another. In that one sentence you will see an enormous amount of information about the world, if just a little imagination and thinking are applied. *Richard Feynman, The Laws of Thermodynamics.*

11.1 Introduction

- Is physically intuitive.
- Allows one to study systems with many degrees of freedom. Diffusion Monte Carlo (DMC) and Green's function Monte Carlo (GFMC) yield in principle the exact solution to Schrödinger's equation.
- Variational Monte Carlo (VMC) is easy to implement but needs a reliable trial wave function, can be difficult to obtain. This is where we will use Hartree-Fock theory to construct an optimal basis.
- DMC/GFMC for fermions (spin with half-integer values, electrons, baryons, neutrinos, quarks) has a sign problem. Nature prefers an anti-symmetric wave function. PDF in this case given distribution of random walkers ($p \geq 0$).
- The solution has a statistical error, which can be large.
- There is a limit for how large systems one can study, DMC needs a huge number of random walkers in order to achieve stable results.
- Obtain only the lowest-lying states with a given symmetry. Difficult to get excited states.
- Quantum systems with many particles at finite temperature: Path Integral Monte Carlo with applications to dense matter and quantum liquids (phase transitions from normal fluid to superfluid). Strong correlations.

- Bose-Einstein condensation of dilute gases, method transition from non-linear PDE to Diffusion Monte Carlo as density increases.
- Light atoms, molecules, solids and nuclei.
- Lattice Quantum-Chromo Dynamics. Impossible to solve without MC calculations.
- Simulations of systems in solid state physics, from semiconductors to spin systems. Many electrons active and possibly strong correlations.

The aim of this chapter is to present examples of applications of Monte Carlo methods in studies of simple quantum mechanical systems. We study systems such as the harmonic oscillator, the hydrogen atom, the hydrogen molecule, the helium atom and more complicated atoms. Systems with man interacting fermions and bosons such as liquid ^{4}He and Bose Einstein condensation of atoms ae discussed in chapter 17. Most quantum mechanical problems of interest in for example atomic, molecular, nuclear and solid state physics consist of a large number of interacting electrons and ions or nucleons. The total number of particles N is usually sufficiently large that an exact solution cannot be found. In quantum mechanics we can express the expectation value for a given $\widehat{\mathbf{O}}$ operator for a system of N particles as

$$\langle \widehat{\mathbf{O}} \rangle = \frac{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) \widehat{\mathbf{o}}(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) \Psi(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N)}{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) \Psi(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N)}, \tag{11.1}$$

where $\Psi(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N)$ is the wave function describing a many-body system. Although we have omitted the time dependence in this equation, it is an in general intractable problem. As an example from the nuclear many-body problem, we can write Schrödinger's equation as a differential equation with the energy operator $\widehat{\mathbf{H}}$ (the so-called energy Hamiltonian) acting on the wave function as

$$\widehat{\mathbf{H}} \Psi(\mathbf{r}_1, .., \mathbf{r}_A, \alpha_1, .., \alpha_A) = E \Psi(\mathbf{r}_1, .., \mathbf{r}_A, \alpha_1, .., \alpha_A)$$

where

$$\mathbf{r}_1, .., \mathbf{r}_A,$$

are the coordinates and

$$\alpha_1, .., \alpha_A,$$

are sets of relevant quantum numbers such as spin and isospin for a system of A nucleons ($A = N + Z$, N being the number of neutrons and Z the number of protons). There are

$$2^A \times \left(\begin{array}{c} A \\ Z \end{array} \right)$$

coupled second-order differential equations in $3A$ dimensions. For a nucleus like ^{10}Be this number is 215040. This is a truely challenging many-body problem.

Eq. (11.1) is a multidimensional integral. As such, Monte Carlo methods are ideal for obtaining expectation values of quantum mechanical operators. Our problem is that we do not know the exact wavefunction $\Psi(\mathbf{r}_1, .., \mathbf{r}_A, \alpha_1, .., \alpha_N)$. We can circumvent this problem by introducing a function which depends on selected variational parameters. This function should capture essential features of the system under consideration. With such a trial wave function we can then attempt to perform a variational calculation of various observables, using Monte Carlo methods for solving Eq. (11.1).

The present chapter aims therefore at giving you an overview of the variational Monte Carlo approach to quantum mechanics. We limit the attention to the simple Metropolis algorithm, without the inclusion of importance sampling. Importance sampling and diffusion Monte Carlo methods are discussed in chapters 17 and 16.

However, before we proceed we need to recapitulate some of the postulates of quantum mechanics. This is done in the next section. The remaining sections deal with mathematical and computational aspects of the variational Monte Carlo methods, with applications from atomic and molecular physis.

11.2 Postulates of Quantum Mechanics

11.2.1 Mathematical Properties of the Wave Functions

Schrödinger's equation for a one-dimensional onebody problem reads

$$-\frac{\hbar^2}{2m}\nabla^2\Psi(x,t)+V(x,t)\Psi(x,t)=\imath\hbar\frac{\partial\Psi(x,t)}{\partial t}, \tag{11.2}$$

where $V(x,t)$ is a potential acting on the particle. The first term is the kinetic energy. The solution to this partial differential equation is the wave function $\Psi(x,t)$. The wave function itself is not an observable (or physical quantity) but it serves to define the quantum mechanical probability, which in turn can be used to compute expectation values of selected operators, such as the kinetic energy or the total energy itself. The quantum mechanical probability $P(x,t)dx$ is defined as[1]

$$P(x,t)dx=\Psi(x,t)^*\Psi(x,t)dx, \tag{11.3}$$

representing the probability of finding the system in a region between x and $x+dx$. It is, as opposed to the wave function, always real, which can be seen from the following definition of the wave function, which has real and imaginary parts,

$$\Psi(x,t)=R(x,t)+\imath I(x,t), \tag{11.4}$$

yielding

$$\Psi(x,t)^*\Psi(x,t)=(R-\imath I)(R+\imath I)=R^2+I^2. \tag{11.5}$$

The variational Monte Carlo approach uses actually this definition of the probability, allowing us thereby to deal with real quantities only. As a small digression, if we perform a rotation of time into the complex plane, using $\tau=it/\hbar$, the time-dependent Schrödinger equation becomes

$$\frac{\partial\Psi(x,\tau)}{\partial\tau}=\frac{\hbar^2}{2m}\frac{\partial^2\Psi(x,\tau)}{\partial x^2}-V(x,\tau)\Psi(x,\tau). \tag{11.6}$$

With $V=0$ we have a diffusion equation in complex time with diffusion constant

$$D=\frac{\hbar^2}{2m}.$$

This is the starting point for the Diffusion Monte Carlo method discussed in chapter 17. In that case it is the wave function itself, given by the distribution of random walkers, that defines the probability. The latter leads to conceptual problems when we have anti-symmetric wave functions, as is the case for particles with the spin being a multiplum of $1/2$. Examples of such particles are various leptons such as

[1] This is Max Born's postulate on how to interpret the wave function resulting from the solution of Schrödinger's equation. It is also the commonly accepted and operational interpretation.

electrons, muons and various neutrinos, baryons like protons and neutrons and quarks such as the up and down quarks.

The Born interpretation constrains the wave function to belong to the class of functions in L^2. Some of the selected conditions which Ψ has to satisfy are

1. Normalization
$$\int_{-\infty}^{\infty} P(x,t)dx = \int_{-\infty}^{\infty} \Psi(x,t)^*\Psi(x,t)dx = 1 \tag{11.7}$$
meaning that
$$\int_{-\infty}^{\infty} \Psi(x,t)^*\Psi(x,t)dx < \infty \tag{11.8}$$

2. $\Psi(x,t)$ and $\partial\Psi(x,t)/\partial x$ must be finite

3. $\Psi(x,t)$ and $\partial\Psi(x,t)/\partial x$ must be continuous.

4. $\Psi(x,t)$ and $\partial\Psi(x,t)/\partial x$ must be single valued

11.2.2 Important Postulates

We list here some of the postulates that we will use in our discussion.

Postulate I

Any physical quantity $A(\vec{r},\vec{p})$ which depends on position $\vec{r}$ and momentum $\vec{p}$ has a corresponding quantum mechanical operator by replacing $\vec{p}$ $-i\hbar\vec{\bigtriangledown}$, yielding the quantum mechanical operator

$$\widehat{\mathbf{A}} = A(\vec{r}, -i\hbar\vec{\bigtriangledown}).$$

Quantity	Classical definition	QM operator
Position	$\vec{r}$	$\widehat{\vec{\mathbf{r}}} = \vec{r}$
Momentum	$\vec{p}$	$\widehat{\vec{\mathbf{p}}} = -i\hbar\vec{\bigtriangledown}$
Orbital momentum	$\vec{L} = \vec{r}\times\vec{p}$	$\widehat{\vec{\mathbf{L}}} = \vec{r}\times(-i\hbar\vec{\bigtriangledown})$
Kinetic energy	$T = (\vec{p})^2/2m$	$\widehat{\mathbf{T}} = -(\hbar^2/2m)(\vec{\bigtriangledown})^2$
Total energy	$H = (p^2/2m) + V(\vec{r})$	$\widehat{\mathbf{H}} = -(\hbar^2/2m)(\vec{\bigtriangledown})^2 + V(\vec{r})$

Postulate II

The only possible outcome of an ideal measurement of the physical quantity A are the eigenvalues of the corresponding quantum mechanical operator $\widehat{\mathbf{A}}$.

$$\widehat{\mathbf{A}}\psi_\nu = a_\nu\psi_\nu,$$

resulting in the eigenvalues $a_1, a_2, a_3, \cdots$ as the only outcomes of a measurement. The corresponding eigenstates $\psi_1, \psi_2, \psi_3 \cdots$ contain all relevant information about the system.

Postulate III

Assume Φ is a linear combination of the eigenfunctions ψ_ν for $\widehat{\mathbf{A}}$,

$$\Phi = c_1\psi_1 + c_2\psi_2 + \cdots = \sum_\nu c_\nu\psi_\nu.$$

The eigenfunctions are orthogonal and we get

$$c_\nu = \int (\Phi)^*\psi_\nu d\tau.$$

From this we can formulate the third postulate:

When the eigenfunction is Φ, the probability of obtaining the value a_ν as the outcome of a measurement of the physical quantity A is given by $|c_\nu|^2$ and ψ_ν is an eigenfunction of $\widehat{\mathbf{A}}$ with eigenvalue a_ν.

As a consequence one can show that:
when a quantal system is in the state Φ, the mean value or expectation value of a physical quantity $A(\vec{r},\vec{p})$ is given by

$$\langle A\rangle = \int (\Phi)^*\widehat{\mathbf{A}}(\vec{r}, -i\hbar\vec{\bigtriangledown})\Phi d\tau.$$

We have assumed that Φ has been normalized, viz., $\int (\Phi)^*\Phi d\tau = 1$. Else

$$\langle A\rangle = \frac{\int (\Phi)^*\widehat{\mathbf{A}}\Phi d\tau}{\int (\Phi)^*\Phi d\tau}.$$

Postulate IV

The time development of a quantal system is given by

$$i\hbar\frac{\partial\Psi}{\partial t} = \widehat{\mathbf{H}}\Psi,$$

with $\widehat{\mathbf{H}}$ the quantal Hamiltonian operator for the system.

11.3 First Encounter with the Variational Monte Carlo Method

The required Monte Carlo techniques for variational Monte Carlo are conceptually simple, but the practical application may turn out to be rather tedious and complex, relying on a good starting point for the variational wave functions. These wave functions should include as much as possible of the inherent physics to the problem, since they form the starting point for a variational calculation of the expectation value of the hamiltonian H. Given a hamiltonian H and a trial wave function Ψ_T, the variational principle states that the expectation value of $\langle H\rangle$, defined through Postulate III

$$\langle H\rangle = \frac{\int d\mathbf{R}\Psi_T^*(\mathbf{R})H(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\Psi_T(\mathbf{R})}, \tag{11.9}$$

is an upper bound to the ground state energy E_0 of the hamiltonian H, that is

$$E_0 \le \langle H\rangle. \tag{11.10}$$

To show this, we note first that the trial wave function can be expanded in the eigenstates of the hamiltonian since they form a complete set, see again Postulate III,

$$\Psi_T(\mathbf{R}) = \sum_i a_i \Psi_i(\mathbf{R}), \tag{11.11}$$

and assuming the set of eigenfunctions to be normalized, insertion of the latter equation in Eq. (11.9) results in

$$\langle H \rangle = \frac{\sum_{mn} a_m^* a_n \int d\mathbf{R} \Psi_m^*(\mathbf{R}) H(\mathbf{R}) \Psi_n(\mathbf{R})}{\sum_{mn} a_m^* a_n \int d\mathbf{R} \Psi_m^*(\mathbf{R}) \Psi_n(\mathbf{R})} = \frac{\sum_{mn} a_m^* a_n \int d\mathbf{R} \Psi_m^*(\mathbf{R}) E_n(\mathbf{R}) \Psi_n(\mathbf{R})}{\sum_n a_n^2}, \tag{11.12}$$

which can be rewritten as

$$\frac{\sum_n a_n^2 E_n}{\sum_n a_n^2} \geq E_0. \tag{11.13}$$

In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. Traditional integration methods such as the Gauss-Legendre will not be adequate for say the computation of the energy of a many-body system. The fact that we need to sample over a multi-dimensional density and that the probability density is to be normalized by the division of the norm of the wave function, suggests that e.g., the Metropolis algorithm may be appropriate.

We could briefly summarize the above variational procedure in the following three steps.

1. Construct first a trial wave function $\psi_T(\mathbf{R};\alpha)$, for say a many-body system consisting of N particles located at positions $\mathbf{R} = (\mathbf{R_1}, \dots, \mathbf{R_N})$. The trial wave function depends on α variational parameters $\alpha = (\alpha_{\mathbf{1}}, \dots, \alpha_{\mathbf{N}})$.

2. Then we evaluate the expectation value of the hamiltonian H

$$\langle H \rangle = \frac{\int d\mathbf{R} \Psi_T^*(\mathbf{R};\alpha) H(\mathbf{R}) \Psi_T(\mathbf{R};\alpha)}{\int d\mathbf{R} \Psi_T^*(\mathbf{R};\alpha) \Psi_T(\mathbf{R};\alpha)}.$$

3. Thereafter we vary α according to some minimization algorithm and return to the first step.

The above loop stops when we reach the minimum of the energy according to some specified criterion. In most cases, a wave function has only small values in large parts of configuration space, and a straightforward procedure which uses homogenously distributed random points in configuration space will most likely lead to poor results. This may suggest that some kind of importance sampling combined with e.g., the Metropolis algorithm may be a more efficient way of obtaining the ground state energy. The hope is then that those regions of configurations space where the wave function assumes appreciable values are sampled more efficiently.

The tedious part in a variational Monte Carlo calculation is the search for the variational minimum. A good knowledge of the system is required in order to carry out reasonable variational Monte Carlo calculations. This is not always the case, and often variational Monte Carlo calculations serve rather as the starting point for so-called diffusion Monte Carlo calculations. Diffusion Monte Carlo is a way of solving exactly the many-body Schrödinger equation by means of a stochastic procedure. A good guess on the binding energy and its wave function is however necessary. A carefully performed variational Monte Carlo calculation can aid in this context. Diffusion Monte Carlo is discussed in depth in chapter 17.

11.4 Variational Monte Carlo for quantum mechanical systems

The variational quantum Monte Carlo has been widely applied to studies of quantal systems. Here we expose its philosophy and present applications and critical discussions.

The recipe, as discussed in chapter 8 as well, consists in choosing a trial wave function $\psi_T(\mathbf{R})$ which we assume to be as realistic as possible. The variable $\mathbf{R}$ stands for the spatial coordinates, in total $3N$ if we have N particles present. The trial wave function serves then, following closely the discussion on importance sampling in section 8.4, as a mean to define the quantal probability distribution

$$P(\mathbf{R};\alpha) = \frac{|\psi_T(\mathbf{R};\alpha)|^2}{\int |\psi_T(\mathbf{R};\alpha)|^2 \, d\mathbf{R};\alpha}. \tag{11.14}$$

This is our new probability distribution function (PDF).

The expectation value of the energy Hamiltonian is given by

$$\langle \widehat{\mathbf{H}} \rangle = \frac{\int d\mathbf{R}\Psi^*(\mathbf{R})H(\mathbf{R})\Psi(\mathbf{R})}{\int d\mathbf{R}\Psi^*(\mathbf{R})\Psi(\mathbf{R})}, \tag{11.15}$$

where Ψ is the exact eigenfunction. Using our trial wave function we define a new operator, the so-called local energy,

$$\widehat{\mathbf{E}}_L(\mathbf{R};\alpha) = \frac{1}{\psi_T(\mathbf{R};\alpha)} \widehat{\mathbf{H}} \psi_T(\mathbf{R};\alpha), \tag{11.16}$$

which, together with our trial PDF allows us to compute the expectation value of the local energy

$$\langle E_L(\alpha) \rangle = \int P(\mathbf{R};\alpha) \widehat{\mathbf{E}}_L(\mathbf{R};\alpha) d\mathbf{R}. \tag{11.17}$$

This equation expresses the variational Monte Carlo approach. We compute this integral for a set of values of α and possible trial wave functions and search for the minimum of the function $E_L(\alpha)$. If the trial wave function is close to the exact wave function, then $\langle E_L(\alpha) \rangle$ should approach $\langle \widehat{\mathbf{H}} \rangle$. Eq. (11.17) is solved using techniques from Monte Carlo integration, see the discussion below. For most hamiltonians, H is a sum of kinetic energy, involving a second derivative, and a momentum independent and spatial dependent potential. The contribution from the potential term is hence just the numerical value of the potential. A typical Hamiltonian reads thus

$$\widehat{\mathbf{H}} = -\frac{\hbar^2}{2m} \sum_{i=1}^{N} \nabla_i^2 + \sum_{i=1}^{N} V_{\mathrm{onebody}}(\mathbf{r}_i) + \sum_{i<j}^{N} V_{\mathrm{int}}(\mid \mathbf{r}_i - \mathbf{r}_j \mid). \tag{11.18}$$

where the sum runs over all particles N. We have included both a onebody potential $V_{\mathrm{onebody}}(\mathbf{r}_i)$ which acts on one particle at the time and a twobody interaction $V_{\mathrm{int}}(\mid \mathbf{r}_i - \mathbf{r}_j \mid)$ which acts between two particles at the time. We can obviously extend this to more complicated three-body and/or many-body forces as well. The main contributions to the energy of physical systems is largely dominated by one- and two-body forces. We will therefore limit our attention to such interactions only.

Our local energy operator becomes then

$$\widehat{\mathbf{E}}_L(\mathbf{R};\alpha) = \frac{1}{\psi_T(\mathbf{R};\alpha)} \left(-\frac{\hbar^2}{2m} \sum_{i=1}^{N} \nabla_i^2 + \sum_{i=1}^{N} V_{\mathrm{onebody}}(\mathbf{r}_i) + \sum_{i<j}^{N} V_{\mathrm{int}}(\mid \mathbf{r}_i - \mathbf{r}_j \mid) \right) \psi_T(\mathbf{R};\alpha), \tag{11.19}$$

resulting in

$$\widehat{\mathbf{E}}_L(\mathbf{R};\alpha) = \frac{1}{\psi_T(\mathbf{R};\alpha)}\left(-\frac{\hbar^2}{2m}\sum_{i=1}^{N}\nabla_i^2\right)\psi_T(\mathbf{R};\alpha) + \sum_{i=1}^{N}V_{\text{onebody}}(\mathbf{r}_i) + \sum_{i<j}^{N}V_{\text{int}}(\mid \mathbf{r}_i - \mathbf{r}_j \mid), \quad (11.20)$$

The numerically time-consuming part in the variational Monte Carlo calculation is the evaluation of the kinetic energy term. The potential energy, as long as it has a simple r-dependence adds only a simple term to the local energy operator.

In our discussion below, we base our numerical Monte Carlo solution on the Metropolis algorithm. The implementation is rather similar to the one discussed in connection with the Ising model, the main difference residing in the form of the PDF. The main test to be performed is a ratio of probabilities. Suppose we are attempting to move from position $\mathbf{R}$ to $\mathbf{R}'$. Then we perform the following two tests.

1. If
$$\frac{P(\mathbf{R}';\alpha)}{P(\mathbf{R};\alpha)} > 1,$$
where $\mathbf{R}'$ is the new position, the new step is accepted, or

2.
$$r \le \frac{P(\mathbf{R}';\alpha)}{P(\mathbf{R};\alpha)},$$
where r is random number generated with uniform PDF such that $r \in [0,1]$, the step is also accepted.

In the Ising model we were flipping one spin at the time. Here we change the position of say a given particle to a trial position $\mathbf{R}'$, and then evaluate the ratio between two probabilities. We note again that we do not need to evaluate the norm[2] $\int |\psi_T(\mathbf{R};\alpha)|^2 d\mathbf{R}$ (an in general impossible task), since we are only computing ratios.

When writing a variational Monte Carlo program, one should always prepare in advance the required formulae for the local energy E_L in Eq. (11.17) and the wave function needed in order to compute the ratios of probabilities in the Metropolis algorithm. These two functions are almost called as often as a random number generator, and care should therefore be exercised in order to prepare an efficient code.

If we now focus on the Metropolis algorithm and the Monte Carlo evaluation of Eq. (11.17), a more detailed algorithm is as follows

– Initialisation: Fix the number of Monte Carlo steps and thermalization steps. Choose an initial $\mathbf{R}$ and variational parameters α and calculate $|\psi_T(\mathbf{R};\alpha)|^2$. Define also the value of the stepsize to be used when moving from one value of $\mathbf{R}$ to a new one.

– Initialise the energy and the variance.

– Start the Monte Carlo calculation with a loop over a given number of Monte Carlo cycles

  1. Calculate a trial position $\mathbf{R}_p = \mathbf{R} + r * step$ where r is a random variable $r \in [0,1]$.
  2. Use then the Metropolis algorithm to accept or reject this move by calculating the ratio
  $$w = P(\mathbf{R}_p)/P(\mathbf{R}).$$
  If $w \ge s$, where s is a random number $s \in [0,1]$, the new position is accepted, else we stay at the same place.

[2]This corresponds to the partition function Z in statistical physics.

3. If the step is accepted, then we set $\mathbf{R} = \mathbf{R}_p$.
4. Update the local energy and the variance.

– When the Monte Carlo sampling is finished, we calculate the mean energy and the standard deviation. Finally, we may print our results to a specified file.

Note well that the way we choose the next step $\mathbf{R}_p = \mathbf{R} + r * step$ is not determined by the wave function. The wave function enters only the determination of the ratio of probabilities, similar to the way we simulated systems in statistical physics. This means in turn that our sampling of points may not be very efficient. We will return to an efficient sampling of integration points in our discussion of diffusion Monte Carlo in chapter 17. This leads to the concept of importance sampling. As such, we limit ourselves in this chapter to the simplest possible form of the Metropolis algorithm, and relegate both importance sampling and advanced optimization techniques to chapter 17.

The best way however to understand the above algorithm and a specific method is to study selected examples.

11.4.1 First illustration of variational Monte Carlo methods

The harmonic oscillator in one dimension lends itself nicely for illustrative purposes. The hamiltonian is

$$H = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{1}{2}kx^2, \tag{11.21}$$

where m is the mass of the particle and k is the force constant, e.g., the spring tension for a classical oscillator. In this example we will make life simple and choose $m = \hbar = k = 1$. We can rewrite the above equation as

$$H = -\frac{d^2}{dx^2} + x^2, \tag{11.22}$$

The energy of the ground state is then $E_0 = 1$. The exact wave function for the ground state is

$$\Psi_0(x) = \frac{1}{\pi^{1/4}}e^{-x^2/2}, \tag{11.23}$$

but since we wish to illustrate the use of Monte Carlo methods, we choose the trial function

$$\Psi_T(x) = \frac{\sqrt{\alpha}}{\pi^{1/4}}e^{-x^2\alpha^2/2}. \tag{11.24}$$

Inserting this function in the expression for the local energy in Eq. (11.16), we obtain the following expression for the local energy

$$E_L(x) = \alpha^2 + x^2(1-\alpha^4), \tag{11.25}$$

with the expectation value for the hamiltonian of Eq. (11.17) given by

$$\langle E_L \rangle = \int_{-\infty}^{\infty} |\psi_T(x)|^2 E_L(x)dx, \tag{11.26}$$

which reads with the above trial wave function

$$\langle E_L \rangle = \frac{\int_{-\infty}^{\infty} dx e^{-x^2\alpha^2}\alpha^2 + x^2(1-\alpha^4)}{\int_{-\infty}^{\infty} dx e^{-x^2\alpha^2}}. \tag{11.27}$$

Using the fact that

$$\int_{-\infty}^{\infty} dx e^{-x^2\alpha^2} = \sqrt{\frac{\pi}{\alpha^2}},$$

we obtain

$$\langle E_L \rangle = \frac{\alpha^2}{2} + \frac{1}{2\alpha^2}. \tag{11.28}$$

and the variance

$$\sigma^2 = \frac{(\alpha^4 - 1)^2}{2\alpha^4}. \tag{11.29}$$

In solving this problem we can choose whether we wish to use the Metropolis algorithm and sample over relevant configurations, or just use random numbers generated from a normal distribution, since the harmonic oscillator wave functions follow closely such a distribution. The latter approach is easily implemented in few lines, namely

```
...   initialisations , declarations of variables
...   mcs = number of  Monte Carlo samplings
//    loop over Monte Carlo samples
      for ( i=0; i < mcs; i++) {
//    generate random variables from gaussian distribution
          x = normal_random(&idum)/sqrt2/alpha;
          local_energy = alpha*alpha + x*x*(1-pow(alpha,4));
          energy += local_energy;
          energy2 += local_energy*local_energy;
//    end of sampling
      }
//    write out the mean energy and the standard deviation
      cout << energy/mcs <<  sqrt((energy2/mcs-(energy/mcs)**2)/mcs));
```

This variational Monte Carlo calculation is rather simple, we just generate a large number N of random numbers corresponding to the gaussian PDF $\sim |\Psi_T|^2$ and for each random number we compute the local energy according to the approximation

$$\langle \widehat{\mathbf{E}}_L \rangle = \int P(\mathbf{R})\widehat{\mathbf{E}}_L(\mathbf{R})d\mathbf{R} \approx \frac{1}{N}\sum_{i=1}^{N} E_L(x_i), \tag{11.30}$$

and the energy squared through

$$\langle \widehat{\mathbf{E}}_L^2 \rangle = \int P(\mathbf{R})\widehat{\mathbf{E}}_L^2(\mathbf{R})d\mathbf{R} \approx \frac{1}{N}\sum_{i=1}^{N} E_L^2(x_i). \tag{11.31}$$

In a certain sense, this is nothing but the importance Monte Carlo sampling discussed in chapter 8 Before we proceed however, there is an important aside which is worth keeping in mind when computing the local energy. We could think of splitting the computation of the expectation value of the local energy into a kinetic energy part and a potential energy part. If we are dealing with a three-dimensional system, the expectation value of the kinetic energy is

$$-\frac{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\nabla^2\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\Psi_T(\mathbf{R})}, \tag{11.32}$$

and we could be tempted to compute, if the wave function obeys spherical symmetry, just the second derivative with respect to one coordinate axis and then multiply by three. This will most likely increase the variance, and should be avoided, even if the final expectation values are similar. Recall that one of the subgoals of a Monte Carlo computation is to decrease the variance.

Another shortcut we could think of is to transform the numerator in the latter equation to

$$\int d\mathbf{R}\Psi_T^*(\mathbf{R})\nabla^2\Psi_T(\mathbf{R}) = -\int d\mathbf{R}(\nabla\Psi_T^*(\mathbf{R}))(\nabla\Psi_T(\mathbf{R})), \tag{11.33}$$

using integration by parts and the relation

$$\int d\mathbf{R}\nabla(\Psi_T^*(\mathbf{R})\nabla\Psi_T(\mathbf{R})) = 0, \tag{11.34}$$

where we have used the fact that the wave function is zero at $\mathbf{R} = \pm\infty$. This relation can in turn be rewritten through integration by parts to

$$\int d\mathbf{R}(\nabla\Psi_T^*(\mathbf{R}))(\nabla\Psi_T(\mathbf{R})) + \int d\mathbf{R}\Psi_T^*(\mathbf{R})\nabla^2\Psi_T(\mathbf{R})) = 0. \tag{11.35}$$

The rhs of Eq. (11.33) is easier and quicker to compute. However, in case the wave function is the exact one, or rather close to the exact one, the lhs yields just a constant times the wave function squared, implying zero variance. The rhs does not and may therefore increase the variance.

If we use integration by part for the harmonic oscillator case, the new local energy is

$$E_L(x) = x^2(1+\alpha^4), \tag{11.36}$$

and the variance

$$\sigma^2 = \frac{(\alpha^4+1)^2}{2\alpha^4}, \tag{11.37}$$

which is larger than the variance of Eq. (11.29).

11.5 Variational Monte Carlo for atoms

The Hamiltonian for an N-electron atomic system consists of two terms

$$\hat{H}(\mathbf{x}) = \hat{T}(\mathbf{x}) + \hat{V}(\mathbf{x}); \tag{11.38}$$

the kinetic and the potential energy operator. Here $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots \mathbf{x}_N\}$ is the spatial and spin degrees of freedom associated with the different particles. The classical kinetic energy

$$T = \frac{\mathbf{P^2}}{2m} + \sum_{j=1}^{N}\frac{\mathbf{p}_j^2}{2m}$$

is transformed to the quantum mechanical kinetic energy operator by operator substitution of the momentum ($p_k \rightarrow -i\hbar\partial/\partial x_k$)

$$\hat{T}(\mathbf{x}) = -\frac{\hbar^2}{2M}\nabla_0^2 - \sum_{i=1}^{N}\frac{\hbar^2}{2m}\nabla_i^2. \tag{11.39}$$

Here the first term is the kinetic energy operator of the nucleus, the second term is the kinetic energy operator of the electrons, M is the mass of the nucleus and m is the electron mass. The potential energy operator is given by

$$\hat{V}(\mathbf{x}) = -\sum_{i=1}^{N}\frac{Ze^2}{(4\pi\epsilon_0)r_i} + \sum_{i=1,i<j}^{N}\frac{e^2}{(4\pi\epsilon_0)r_{ij}}, \tag{11.40}$$

where the r_i's are the electron-nucleus distances and the r_{ij}'s are the inter-electronic distances.

We seek to find controlled and well understood approximations in order to reduce the complexity of the above equations. The *Born-Oppenheimer approximation* is a commonly used approximation, in which the motion of the nucleus is disregarded.

11.5.1 The Born-Oppenheimer Approximation

In a system of interacting electrons and a nucleus there will usually be little momentum transfer between the two types of particles due to their differing masses. The forces between the particles are of similar magnitude due to their similar charge. If one assumes that the momenta of the particles are also similar, the nucleus must have a much smaller velocity than the electrons due to its far greater mass. On the time-scale of nuclear motion, one can therefore consider the electrons to relax to a ground-state given by the Hamiltonian of Eqs. (11.38), (11.39) and (11.40) with the nucleus at a fixed location. This separation of the electronic and nuclear degrees of freedom is known as the Born-Oppenheimer approximation.

In the center of mass system the kinetic energy operator reads

$$\hat{T}(\mathbf{x}) = -\frac{\hbar^2}{2(M+Nm)}\nabla^2_{CM} - \frac{\hbar^2}{2\mu}\sum_{i=1}^{N}\nabla^2_i - \frac{\hbar^2}{M}\sum_{i>j}^{N}\nabla_i\cdot\nabla_j, \tag{11.41}$$

while the potential energy operator remains unchanged. Note that the Laplace operators ∇^2_i now are in the center of mass reference system.

The first term of Eq. (11.41) represents the kinetic energy operator of the center of mass. The second term represents the sum of the kinetic energy operators of the N electrons, each of them having their mass m replaced by the reduced mass $\mu = mM/(m+M)$ because of the motion of the nucleus. The nuclear motion is also responsible for the third term, or the *mass polarization* term.

The nucleus consists of protons and neutrons. The proton-electron mass ratio is about $1/1836$ and the neutron-electron mass ratio is about $1/1839$, so regarding the nucleus as stationary is a natural approximation. Taking the limit $M \to \infty$ in Eq. (11.41), the kinetic energy operator reduces to

$$\hat{T} = -\sum_{i=1}^{N}\frac{\hbar^2}{2m}\nabla^2_i \tag{11.42}$$

The Born-Oppenheimer approximation thus disregards both the kinetic energy of the center of mass as well as the mass polarization term. The effects of the Born-Oppenheimer approximation are quite small and they are also well accounted for. However, this simplified electronic Hamiltonian remains very difficult to solve, and analytical solutions do not exist for general systems with more than one electron. We use the Born-Oppenheimer approximation in our discussion of atomic and molecular systems.

The first term of Eq. (11.40) is the nucleus-electron potential and the second term is the electron-electron potential. The inter-electronic potentials are the main problem in atomic physics. Because of these terms, the Hamiltonian cannot be separated into one-particle parts, and the problem must be solved as a whole. A common approximation is to regard the effects of the electron-electron interactions either as averaged over the domain or by means of introducing a density functional, such as by Hartree-Fock (HF) or Density Functional Theory (DFT). These approaches are actually very efficient, and about 99% or more of the electronic energies are obtained for most HF calculations. Other observables are usually obtained to an accuracy of about $90-95\%$ (ref. [64]).

11.5.2 The hydrogen Atom

The spatial Schrödinger equation for the three-dimensional hydrogen atom can be solved analytically, see for example Ref. [65] for details. To achieve this, we rewrite the equation in terms of spherical coordinates using

$$x = rsin\theta cos\phi, \tag{11.43}$$

$$y = rsin\theta sin\phi, \tag{11.44}$$

and

$$z = rcos\theta. \tag{11.45}$$

The reason we introduce spherical coordinates is the spherical symmetry of the Coulomb potential

$$\frac{e^2}{4\pi\epsilon_0 r} = \frac{e^2}{4\pi\epsilon_0\sqrt{x^2+y^2+z^2}}, \tag{11.46}$$

where we have used $r = \sqrt{x^2+y^2+z^2}$. It is not possible to find a separable solution of the type

$$\psi(x,y,z) = \psi(x)\psi(y)\psi(z). \tag{11.47}$$

as we can with the harmonic oscillator in three dimensions. However, with spherical coordinates we can find a solution of the form

$$\psi(r,\theta,\phi) = R(r)P(\theta)F(\phi) = RPF. \tag{11.48}$$

These three coordinates yield in turn three quantum numbers which determine the enegy of the systems. We obtain three sets of ordinary second-order differential equations which can be solved analytically, resulting in

$$\frac{1}{F}\frac{\partial^2 F}{\partial\phi^2} = -C_\phi^2, \tag{11.49}$$

$$C_r sin^2(\theta)P + sin(\theta)\frac{\partial}{\partial\theta}(sin(\theta)\frac{\partial P}{\partial\theta}) = C_\phi^2 P, \tag{11.50}$$

and

$$\frac{1}{R}\frac{\partial}{\partial r}(r^2\frac{\partial R}{\partial r}) + \frac{2mrke^2}{\hbar^2} + \frac{2mr^2}{\hbar^2}E = C_r, \tag{11.51}$$

where C_r and C_ϕ are constants. The angle-dependent differential equations result in the spherical harmonic functions as solutions, with quantum numbers l and m_l. These functions are given by

$$Y_{lm_l}(\theta,\phi) = P(\theta)F(\phi) = \sqrt{\frac{(2l+1)(l-m_l)!}{4\pi(l+m_l)!}}P_l^{m_l}(cos(\theta))\exp{(im_l\phi)}, \tag{11.52}$$

with $P_l^{m_l}$ being the associated Legendre polynomials They can be rewritten as

$$Y_{lm_l}(\theta,\phi) = sin^{|m_l|}(\theta)\times(\text{polynom}(cos\theta))\exp{(im_l\phi)}, \tag{11.53}$$

with the following selected examples

$$Y_{00} = \sqrt{\frac{1}{4\pi}}, \tag{11.54}$$

for $l = m_l = 0$,

$$Y_{10} = \sqrt{\frac{3}{4\pi}}cos(\theta), \tag{11.55}$$

for $l = 1$ og $m_l = 0$,

$$Y_{1\pm1} = \sqrt{\frac{3}{8\pi}} sin(\theta) exp(\pm i\phi), \tag{11.56}$$

for $l = 1$ og $m_l = \pm 1$, and

$$Y_{20} = \sqrt{\frac{5}{16\pi}} (3cos^2(\theta) - 1) \tag{11.57}$$

for $l = 2$ og $m_l = 0$. The quantum numbers l and m_l represent the orbital momentum and projection of the orbital momentum, respectively and take the values

1. $$l \geq 0$$

2. $$l = 0, 1, 2, \ldots$$

3. $$m_l = -l, -l+1, \ldots, l-1, l$$

The spherical harmonics for $l \leq 3$ are listed in Table 11.1.

Spherical Harmonics

$m_l \backslash l$	0	1	2	3
+3				$-\frac{1}{8}(\frac{35}{\pi})^{1/2} sin^3\theta e^{+3i\phi}$
+2			$\frac{1}{4}(\frac{15}{2\pi})^{1/2} sin^2\theta e^{+2i\phi}$	$\frac{1}{4}(\frac{105}{2\pi})^{1/2} cos\theta sin^2\theta e^{+2i\phi}$
+1		$-\frac{1}{2}(\frac{3}{2\pi})^{1/2} sin\theta e^{+i\phi}$	$-\frac{1}{2}(\frac{15}{2\pi})^{1/2} cos\theta sin\theta e^{+i\phi}$	$-\frac{1}{8}(\frac{21}{2\pi})^{1/2}(5cos^2\theta - 1) sin\theta e^{+i\phi}$
0	$\frac{1}{2\pi^{1/2}}$	$\frac{1}{2}(\frac{3}{\pi})^{1/2} cos\theta$	$\frac{1}{4}(\frac{5}{\pi})^{1/2}(3cos^2\theta - 1)$	$\frac{1}{4}(\frac{7}{\pi})^{1/2}(2 - 5sin^2\theta) cos\theta$
-1		$+\frac{1}{2}(\frac{3}{2\pi})^{1/2} sin\theta e^{-i\phi}$	$+\frac{1}{2}(\frac{15}{2\pi})^{1/2} cos\theta sin\theta e^{-i\phi}$	$+\frac{1}{8}(\frac{21}{2\pi})^{1/2}(5cos^2\theta - 1) sin\theta e^{-i\phi}$
-2			$\frac{1}{4}(\frac{15}{2\pi})^{1/2} sin^2\theta e^{-2i\phi}$	$\frac{1}{4}(\frac{105}{2\pi})^{1/2} cos\theta sin^2\theta e^{-2i\phi}$
-3				$+\frac{1}{8}(\frac{35}{\pi})^{1/2} sin^3\theta e^{-3i\phi}$

Table 11.1: Spherical harmonics Y_{lm_l} for the lowest l and m_l values.

We concentrate on the radial equation, which can be rewritten as

$$-\frac{\hbar^2 r^2}{2m}\left(\frac{\partial}{\partial r}(r^2\frac{\partial R(r)}{\partial r})\right) - \frac{ke^2}{r}R(r) + \frac{\hbar^2 l(l+1)}{2mr^2}R(r) = ER(r). \tag{11.58}$$

Introducing the function $u(r) = rR(r)$, we can rewrite the last equation as

The radial Schrödinger equation for the hydrogen atom can be written as

$$-\frac{\hbar^2}{2m}\frac{\partial^2 u(r)}{\partial r^2} - \left(\frac{ke^2}{r} - \frac{\hbar^2 l(l+1)}{2mr^2}\right)u(r) = Eu(r), \tag{11.59}$$

where m is the mass of the electron, l its orbital momentum taking values $l = 0, 1, 2, \dots$, and the term ke^2/r is the Coulomb potential. The first terms is the kinetic energy. The full wave function will also depend on the other variables θ and ϕ as well. The energy, with no external magnetic field is however determined by the above equation . We can then think of the radial Schrödinger equation to be equivalent to a one-dimensional movement conditioned by an effective potential

$$V_{\text{eff}}(r) = -\frac{ke^2}{r} + \frac{\hbar^2 l(l+1)}{2mr^2}. \tag{11.60}$$

The radial equation can also be solved analytically resulting in the quantum numbers n in addition to lm_l. The solution R_{nl} to the radial equation is given by the Laguerre polynomials. The analytic solutions are given by

$$\psi_{nlm_l}(r, \theta, \phi) = \psi_{nlm_l} = R_{nl}(r) Y_{lm_l}(\theta, \phi) = R_{nl} Y_{lm_l} \tag{11.61}$$

The ground state is defined by $n = 1$ og $l = m_l = 0$ and reads

$$\psi_{100} = \frac{1}{a_0^{3/2}\sqrt{\pi}} e^{-r/a_0}, \tag{11.62}$$

where we have defined the Bohr radius $a_0 = 0.05$ nm

$$a_0 = \frac{\hbar^2}{mke^2}. \tag{11.63}$$

The first excited state with $l = 0$ is

$$\psi_{200} = \frac{1}{4a_0^{3/2}\sqrt{2\pi}} \left(2 - \frac{r}{a_0}\right) e^{-r/2a_0}. \tag{11.64}$$

For states with with $l = 1$ and $n = 2$, we can have the following combinations with $m_l = 0$

$$\psi_{210} = \frac{1}{4a_0^{3/2}\sqrt{2\pi}} \left(\frac{r}{a_0}\right) e^{-r/2a_0} cos(\theta), \tag{11.65}$$

and $m_l = \pm 1$

$$\psi_{21\pm 1} = \frac{1}{8a_0^{3/2}\sqrt{\pi}} \left(\frac{r}{a_0}\right) e^{-r/2a_0} sin(\theta) e^{\pm i\phi}. \tag{11.66}$$

The exact energy is independent of l and m_l, since the potential is spherically symmetric.

The first few non-normalized radial solutions of equation are listed in Table 11.2. A problem with the spherical harmonics of table 11.1 is that they are complex. The introduction of *solid harmonics* allows the use of real orbital wave-functions for a wide range of applications. The complex solid harmonics $\mathcal{Y}_{lm_l}(\mathbf{r})$ are related to the spherical harmonics $Y_{lm_L}(\mathbf{r})$ through

$$\mathcal{Y}_{lm_l}(\mathbf{r}) = r^l Y_{lm_l}(\mathbf{r}).$$

By factoring out the leading r-dependency of the radial-function

$$\mathcal{R}_{nl}(\mathbf{r}) = r^{-l} R_{nl}(\mathbf{r}),$$

we obtain

$$\Psi_{nlm_l}(r, \theta, \phi) = \mathcal{R}_{nl}(\mathbf{r}) \cdot \mathcal{Y}_{lm_l}(\mathbf{r}).$$

Hydrogen-Like Atomic Radial Functions

$l\backslash n$	1	2	3
0	e^{-Zr}	$(2-r)e^{-Zr/2}$	$(27-18r+2r^2)e^{-Zr/3}$
1		$re^{-Zr/2}$	$r(6-r)e^{-Zr/3}$
2			$r^2e^{-Zr/3}$

Table 11.2: The first few radial functions of the hydrogen-like atoms.

For the theoretical development of the *real solid harmonics* see Ref. [66]. Here Helgaker *et al* first express the complex solid harmonics, C_{lm_l}, by (complex) Cartesian coordinates, and arrive at the real solid harmonics, S_{lm_l}, through the unitary transformation

$$\begin{pmatrix} S_{lm_l} \\ S_{l,-m_l} \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} (-1)_l^m & 1 \\ -(-1)_l^m i & i \end{pmatrix} \begin{pmatrix} C_{lm_l} \\ C_{l,-m_l} \end{pmatrix}.$$

This transformation will not alter any physical quantities that are degenerate in the subspace consisting of opposite magnetic quantum numbers (the angular momentum l is equal for both these cases). This means for example that the above transformation does not alter the energies, unless an external magnetic field is applied to the system. Henceforth, we will use the solid harmonics, and note that changing the spherical potential beyond the Coulomb potential will not alter the solid harmonics. The lowest-order real solid harmonics are listed in table 11.3.

Real Solid Harmonics

$m_l\backslash l$	0	1	2	3
+3				$\frac{1}{2}\sqrt{\frac{5}{2}}(x^2-3y^2)x$
+2			$\frac{1}{2}\sqrt{3}(x^2-y^2)$	$\frac{1}{2}\sqrt{15}(x^2-y^2)z$
+1		x	$\sqrt{3}xz$	$\frac{1}{2}\sqrt{\frac{3}{2}}(5z^2-r^2)x$
0	1	y	$\frac{1}{2}(3z^2-r^2)$	$\frac{1}{2}(5z^2-3r^2)x$
-1		z	$\sqrt{3}yz$	$\frac{1}{2}\sqrt{\frac{3}{2}}(5z^2-r^2)y$
-2			$\sqrt{3}xy$	$\sqrt{15}xyz$
-3				$\frac{1}{2}\sqrt{\frac{5}{2}}(3x^2-y^2)y$

Table 11.3: The first-order real solid harmonics $\mathcal{Y}_{lm_l}$.

When solving equations numerically, it is often convenient to rewrite the equation in terms of dimensionless variables. One reason is the fact that several of the constants may differ largely in value, and hence result in potential losses of numerical precision. The other main reason for doing this is that the equation in dimensionless form is easier to code, sparing one for eventual typographic errors. In order to do so, we introduce first the dimensionless variable $\rho = r/\beta$, where β is a constant we can choose. Schrödinger's equation is then rewritten as

$$-\frac{1}{2}\frac{\partial^2 u(\rho)}{\partial \rho^2} - \frac{mke^2\beta}{\hbar^2\rho}u(\rho) + \frac{l(l+1)}{2\rho^2}u(\rho) = \frac{m\beta^2}{\hbar^2}Eu(\rho). \tag{11.67}$$

We can determine β by simply requiring[3]

$$\frac{mke^2\beta}{\hbar^2} = 1 \tag{11.68}$$

With this choice, the constant β becomes the famous Bohr radius $a_0 = 0.05$ nm $a_0 = \beta = \hbar^2/mke^2$.

As a petit digression, we list here the standard units used in atomic physics and molecular physics calculations. It is common to scale atomic units by setting $m = e = \hbar = 4\pi\epsilon_0 = 1$, see table 11.4.

Atomic Units

Quantity	**SI**	**Atomic unit**
Electron mass, m	$9.109 \cdot 10^{-31}$ kg	1
Charge, e	$1.602 \cdot 10^{-19}$ C	1
Planck's reduced constant, $\hbar$	$1.055 \cdot 10^{-34}$ Js	1
Permittivity, $4\pi\epsilon_0$	$1.113 \cdot 10^{-10}$ C^2 J^{-1} m^{-1}	1
Energy, $\frac{e^2}{4\pi\epsilon_0 a_0}$	27.211 eV	1
Length, $a_0 = \frac{4\pi\epsilon_0\hbar^2}{me^2}$	$0.529 \cdot 10^{-10}$ m	1

Table 11.4: Scaling from SI to atomic units

We introduce thereafter the variable λ

$$\lambda = \frac{m\beta^2}{\hbar^2}E, \tag{11.69}$$

and inserting β and the exact energy $E = E_0/n^2$, with $E_0 = 13.6$ eV, we have that

$$\lambda = -\frac{1}{2n^2}, \tag{11.70}$$

n being the principal quantum number. The equation we are then going to solve numerically is now

$$-\frac{1}{2}\frac{\partial^2 u(\rho)}{\partial \rho^2} - \frac{u(\rho)}{\rho} + \frac{l(l+1)}{2\rho^2}u(\rho) - \lambda u(\rho) = 0, \tag{11.71}$$

with the hamiltonian

$$H = -\frac{1}{2}\frac{\partial^2}{\partial \rho^2} - \frac{1}{\rho} + \frac{l(l+1)}{2\rho^2}. \tag{11.72}$$

[3]Remember that we are free to choose β.

The ground state of the hydrogen atom has the energy $\lambda = -1/2$, or $E = -13.6$ eV. The exact wave function obtained from Eq. (11.71) is

$$u(\rho) = \rho e^{-\rho}, \tag{11.73}$$

which yields the energy $\lambda = -1/2$. Sticking to our variational philosophy, we could now introduce a variational parameter α resulting in a trial wave function

$$u_T^{\alpha}(\rho) = \alpha\rho e^{-\alpha\rho}. \tag{11.74}$$

Inserting this wave function into the expression for the local energy E_L of Eq. (11.16) yields (check it!)

$$E_L(\rho) = -\frac{1}{\rho} - \frac{\alpha}{2}\left(\alpha - \frac{2}{\rho}\right). \tag{11.75}$$

For the hydrogen atom, we could perform the variational calculation along the same lines as we did for the harmonic oscillator. The only difference is that Eq. (11.17) now reads

$$\langle H \rangle = \int P(\mathbf{R})E_L(\mathbf{R})d\mathbf{R} = \int_0^{\infty} \alpha^2\rho^2 e^{-2\alpha\rho}E_L(\rho)\rho^2 d\rho, \tag{11.76}$$

since $\rho \in [0,\infty]$. In this case we would use the exponential distribution instead of the normal distrubution, and our code would contain the following elements

```
...   initialisations , declarations of variables
...   mcs = number of  Monte Carlo samplings

//    loop over Monte Carlo samples
      for ( i=0; i < mcs; i++) {

//    generate random variables from the exponential
//    distribution using ran1 and transforming to
//    to an exponential mapping y = -ln(1-x)
        x=ran1(&idum);
        y=-log(1.-x);
//    in our case y = rho*alpha*2
        rho = y/alpha/2;
        local_energy = -1/rho  -0.5*alpha*(alpha-2/rho);
        energy += (local_energy);
        energy2 += local_energy*local_energy;
//    end of sampling
      }
//    write out the mean energy and the standard deviation
      cout << energy/mcs << sqrt((energy2/mcs-(energy/mcs)**2)/mcs);
```

As for the harmonic oscillator case we just need to generate a large number N of random numbers corresponding to the exponential PDF $\alpha^2\rho^2 e^{-2\alpha\rho}$ and for each random number we compute the local energy and variance.

11.5.3 Metropolis sampling for the hydrogen atom and the harmonic oscillator

We present in this subsection results for the ground states of the hydrogen atom and harmonic oscillator using a variational Monte Carlo procedure. For the hydrogen atom, the trial wave function

$$u_T^{\alpha}(\rho) = \alpha\rho e^{-\alpha\rho},$$

depends only on the dimensionless radius ρ. It is the solution of a one-dimensional differential equation, as is the case for the harmonic oscillator as well. The latter has the trial wave function

$$\Psi_T(x) = \frac{\sqrt{\alpha}}{\pi^{1/4}} e^{-x^2\alpha^2/2}.$$

However, for the hydrogen atom we have $\rho \in [0,\infty]$, while for the harmonic oscillator we have $x \in [-\infty,\infty]$.

This has important consequences for the way we generate random positions. For the hydrogen atom we have a random position given by e.g.,

```
r_old = step_length*(ran1(&idum))/alpha;
```

which ensures that $\rho \geq 0$, while for the harmonic oscillator we have

```
r_old = step_length*(ran1(&idum)-0.5)/alpha;
```

in order to have $x \in [-\infty,\infty]$. This is however not implemented in the program below. There, importance sampling is not included. We simulate points in the x, y and z directions using random numbers generated by the uniform distribution and multiplied by the step length. Note that we have to define a step length in our calculations. Here one has to play around with different values for the step and as a rule of thumb (one of the golden Monte Carlo rules), the step length should be chosen so that roughly 50% of all new moves are accepted. In the program at the end of this section we have also scaled the random position with the variational parameter α. The reason for this particular choice is that we have an external loop over the variational parameter. Different variational parameters will obviously yield different acceptance rates if we use the same step length. An alternative to the code below is to perform the Monte Carlo sampling with just one variational parameter, and play around with different step lengths in order to achieve a reasonable acceptance ratio. Another possibility is to include a more advanced test which restarts the Monte Carlo sampling with a new step length if the specific variational parameter and chosen step length lead to a too low acceptance ratio.

In Figs. 11.1 and 11.2 we plot the ground state energies for the one-dimensional harmonic oscillator and the hydrogen atom, respectively, as functions of the variational parameter α. These results are also displayed in Tables 11.5 and 11.6. In these tables we list the variance and the standard deviation as well. We note that at α we obtain the exact result, and the variance is zero, as it should. The reason is that we then have the exact wave function, and the action of the hamiltionan on the wave function

$$H\psi = \mathrm{constant} \times \psi,$$

yields just a constant. The integral which defines various expectation values involving moments of the hamiltonian becomes then

$$\langle H^n \rangle = \frac{\int d\mathbf{R}\Psi_T^*(\mathbf{R})H^n(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\Psi_T(\mathbf{R})} = \mathrm{constant} \times \frac{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\Psi_T(\mathbf{R})} = \mathrm{constant}. \tag{11.77}$$

This explains why the variance is zero for $\alpha = 1$. However, the hydrogen atom and the harmonic oscillator are some of the few cases where we can use a trial wave function proportional to the exact one. These two systems are also some of the few examples of cases where we can find an exact solution to the problem. In most cases of interest, we do not know a priori the exact wave function, or how to make a good trial wave function. In essentially all real problems a large amount of CPU time and numerical experimenting is needed in order to ascertain the validity of a Monte Carlo estimate. The next examples deal with such problems.

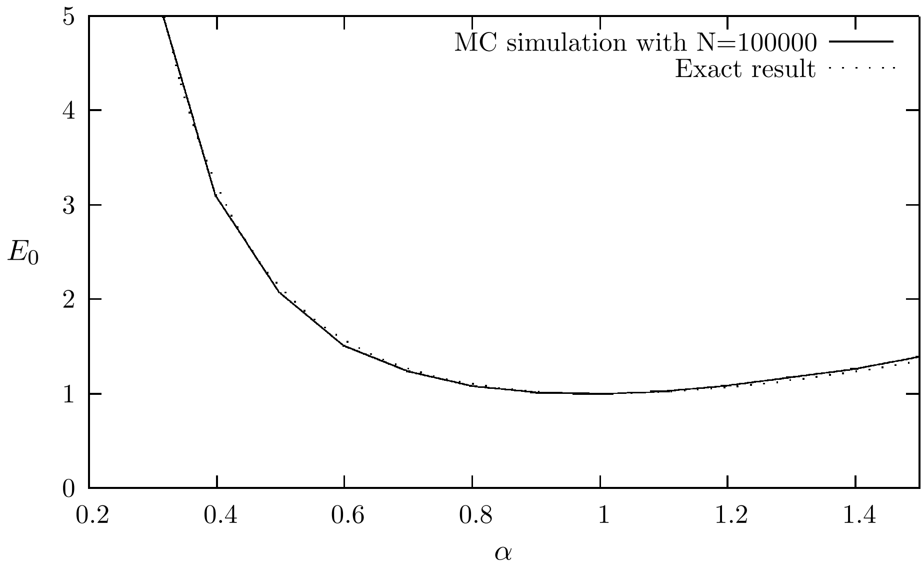


Figure 11.1: Result for ground state energy of the harmonic oscillator as function of the variational parameter α. The exact result is for $\alpha = 1$ with an energy $E = 1$. See text for further details

Table 11.5: Result for ground state energy of the harmonic oscillator as function of the variational parameter α. The exact result is for $\alpha = 1$ with an energy $E = 1$. The energy variance σ^2 and the standard deviation $\sigma/\sqrt{N}$ are also listed. The variable N is the number of Monte Carlo samples. In this calculation we set $N = 100000$ and a step length of 2 was used in order to obtain an acceptance of $\approx 50\%$.

α	$\langle H \rangle$	σ^2	$\sigma/\sqrt{N}$
5.00000E-01	2.06479E+00	5.78739E+00	7.60749E-03
6.00000E-01	1.50495E+00	2.32782E+00	4.82475E-03
7.00000E-01	1.23264E+00	9.82479E-01	3.13445E-03
8.00000E-01	1.08007E+00	3.44857E-01	1.85703E-03
9.00000E-01	1.01111E+00	7.24827E-02	8.51368E-04
1.00000E-00	1.00000E+00	0.00000E+00	0.00000E+00
1.10000E+00	1.02621E+00	5.95716E-02	7.71826E-04
1.20000E+00	1.08667E+00	2.23389E-01	1.49462E-03
1.30000E+00	1.17168E+00	4.78446E-01	2.18734E-03
1.40000E+00	1.26374E+00	8.55524E-01	2.92493E-03
1.50000E+00	1.38897E+00	1.30720E+00	3.61553E-03

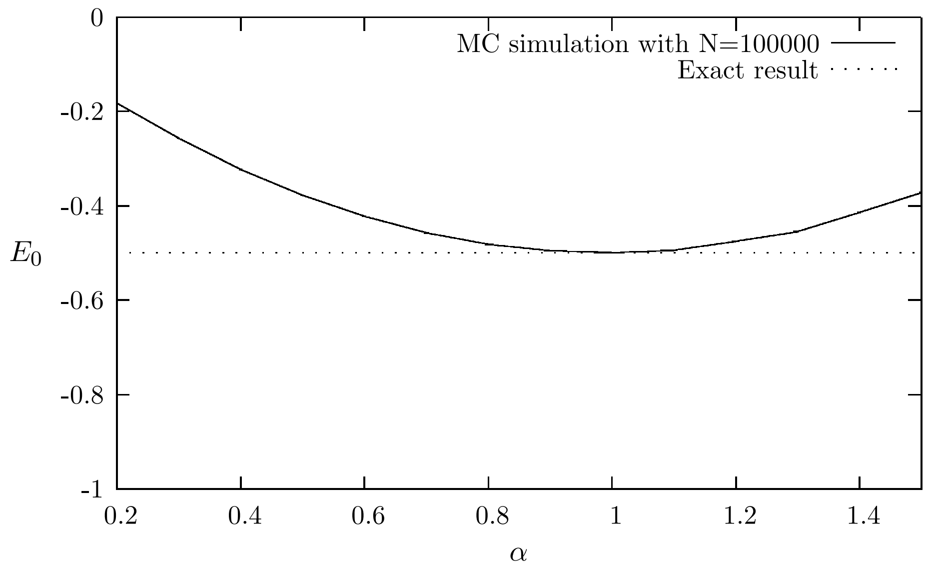


Figure 11.2: Result for ground state energy of the hydrogen atom as function of the variational parameter α. The exact result is for $\alpha = 1$ with an energy $E = -1/2$. See text for further details

Table 11.6: Result for ground state energy of the hydrogen atom as function of the variational parameter α. The exact result is for $\alpha = 1$ with an energy $E = -1/2$. The energy variance σ^2 and the standard deviation $\sigma/\sqrt{N}$ are also listed. The variable N is the number of Monte Carlo samples. In this calculation we fixed $N = 100000$ and a step length of 4 Bohr radii was used in order to obtain an acceptance of $\approx 50\%$.

α	$\langle H \rangle$	σ^2	$\sigma/\sqrt{N}$
5.00000E-01	-3.76740E-01	6.10503E-02	7.81347E-04
6.00000E-01	-4.21744E-01	5.22322E-02	7.22718E-04
7.00000E-01	-4.57759E-01	4.51201E-02	6.71715E-04
8.00000E-01	-4.81461E-01	3.05736E-02	5.52934E-04
9.00000E-01	-4.95899E-01	8.20497E-03	2.86443E-04
1.00000E-00	-5.00000E-01	0.00000E+00	0.00000E+00
1.10000E+00	-4.93738E-01	1.16989E-02	3.42036E-04
1.20000E+00	-4.75563E-01	8.85899E-02	9.41222E-04
1.30000E+00	-4.54341E-01	1.45171E-01	1.20487E-03
1.40000E+00	-4.13220E-01	3.14113E-01	1.77232E-03
1.50000E+00	-3.72241E-01	5.45568E-01	2.33574E-03

11.5.4 The helium atom

Most physical problems of interest in atomic, molecular and solid state physics consist of a number of interacting electrons and ions. The total number of particles N is usually sufficiently large that an exact solution cannot be found. Typically, the expectation value for a chosen hamiltonian for a system of N particles is

$$\langle H \rangle = \frac{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) H(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) \Psi(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N)}{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N) \Psi(\mathbf{R_1}, \mathbf{R}_2, \dots, \mathbf{R}_N)}, \quad (11.78)$$

an in general intractable problem. Controlled and well understood approximations are sought to reduce the complexity to a tractable level. Once the equations are solved, a large number of properties may be calculated from the wave function. Errors or approximations made in obtaining the wave function will be manifest in any property derived from the wave function. Where high accuracy is required, considerable attention must be paid to the derivation of the wave function and any approximations made.

The helium atom consists of two electrons and a nucleus with charge $Z = 2$. In setting up the hamiltonian of this system, we need to account for the repulsion between the two electrons as well.

A common and very reasonable approximation used in the solution of equation of the Schrödinger equation for systems of interacting electrons and ions is the Born-Oppenheimer approximation discussed above. In a system of interacting electrons and nuclei there will usually be little momentum transfer between the two types of particles due to their greatly differing masses. The forces between the particles are of similar magnitude due to their similar charge. If one then assumes that the momenta of the particles are also similar, then the nuclei must have much smaller velocities than the electrons due to their far greater mass. On the time-scale of nuclear motion, one can therefore consider the electrons to relax to a ground-state with the nuclei at fixed locations. This separation of the electronic and nuclear degrees of freedom is the the Born-Oppenheimer approximation we discussed previously in this chapter. But even this simplified electronic Hamiltonian remains very difficult to solve. No analytic solutions exist for general systems with more than one electron.

If we label the distance between electron 1 and the nucleus as r_1. Similarly we have r_2 for electron 2. The contribution to the potential energy due to the attraction from the nucleus is

$$-\frac{2ke^2}{r_1} - \frac{2ke^2}{r_2}, \quad (11.79)$$

and if we add the repulsion arising from the two interacting electrons, we obtain the potential energy

$$V(r_1, r_2) = -\frac{2ke^2}{r_1} - \frac{2ke^2}{r_2} + \frac{ke^2}{r_{12}}, \quad (11.80)$$

with the electrons separated at a distance $r_{12} = |\mathbf{r}_1 - \mathbf{r}_2|$. The hamiltonian becomes then

$$\widehat{\mathbf{H}} = -\frac{\hbar^2 \nabla_1^2}{2m} - \frac{\hbar^2 \nabla_2^2}{2m} - \frac{2ke^2}{r_1} - \frac{2ke^2}{r_2} + \frac{ke^2}{r_{12}}, \quad (11.81)$$

and Schrödingers equation reads

$$\widehat{\mathbf{H}}\psi = E\psi. \quad (11.82)$$

Note that this equation has been written in atomic units $a.u.$ which are more convenient for quantum mechanical problems. This means that the final energy has to be multiplied by a $2 \times E_0$, where $E_0 = 13.6$ eV, the binding energy of the hydrogen atom.

A very simple first approximation to this system is to omit the repulsion between the two electrons. The potential energy becomes then

$$V(r_1, r_2) \approx -\frac{Zke^2}{r_1} - \frac{Zke^2}{r_2}. \tag{11.83}$$

The advantage of this approximation is that each electron can be treated as being independent of each other, implying that each electron sees just a centrally symmetric potential, or central field.

To see whether this gives a meaningful result, we set $Z = 2$ and neglect totally the repulsion between the two electrons. Electron 1 has the following hamiltonian

$$\widehat{\mathbf{h}}_1 = -\frac{\hbar^2\nabla_1^2}{2m} - \frac{2ke^2}{r_1}, \tag{11.84}$$

with pertinent wave function and eigenvalue

$$\widehat{\mathbf{h}}_1\psi_a = E_a\psi_a, \tag{11.85}$$

where $a = \{n_a l_a m_{l_a}\}$, are its quantum numbers. The energy E_a is

$$E_a = -\frac{Z^2E_0}{n_a^2}, \tag{11.86}$$

med $E_0 = 13.6$ eV, being the ground state energy of the hydrogen atom. In a similar way, we obatin for electron 2

$$\widehat{\mathbf{h}}_2 = -\frac{\hbar^2\nabla_2^2}{2m} - \frac{2ke^2}{r_2}, \tag{11.87}$$

with wave function

$$\widehat{\mathbf{h}}_2\psi_b = E_b\psi_b, \tag{11.88}$$

and $b = \{n_b l_b m_{l_b}\}$, and energy

$$E_b = \frac{Z^2E_0}{n_b^2}. \tag{11.89}$$

Since the electrons do not interact, we can assume that the ground state wave function of the helium atom is given by

$$\psi = \psi_a\psi_b, \tag{11.90}$$

resulting in the following approximation to Schrödinger's equation

$$\left(\widehat{\mathbf{h}}_1 + \widehat{\mathbf{h}}_2\right)\psi = \left(\widehat{\mathbf{h}}_1 + \widehat{\mathbf{h}}_2\right)\psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2) = E_{ab}\psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2). \tag{11.91}$$

The energy becomes then

$$\left(\widehat{\mathbf{h}}_1\psi_a(\mathbf{r}_1)\right)\psi_b(\mathbf{r}_2) + \left(\widehat{\mathbf{h}}_2\psi_b(\mathbf{r}_2)\right)\psi_a(\mathbf{r}_1) = (E_a + E_b)\,\psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2), \tag{11.92}$$

yielding

$$E_{ab} = Z^2E_0\left(\frac{1}{n_a^2} + \frac{1}{n_b^2}\right). \tag{11.93}$$

If we insert $Z = 2$ and assume that the ground state is determined by two electrons in the lowest-lying hydrogen orbit with $n_a = n_b = 1$, the energy becomes

$$E_{ab} = 8E_0 = -108.8 \text{ eV}, \tag{11.94}$$

while the experimental value is -78.8 eV. Clearly, this discrepancy is essentially due to our omission of the repulsion arising from the interaction of two electrons.

Choice of trial wave function

The choice of trial wave function is critical in variational Monte Carlo calculations. How to choose it is however a highly non-trivial task. All observables are evaluated with respect to the probability distribution

$$P(\mathbf{R}) = \frac{|\psi_T(\mathbf{R})|^2}{\int |\psi_T(\mathbf{R})|^2 \, d\mathbf{R}}. \tag{11.95}$$

generated by the trial wave function. The trial wave function must approximate an exact eigenstate in order that accurate results are to be obtained. Improved trial wave functions also improve the importance sampling, reducing the cost of obtaining a certain statistical accuracy.

Quantum Monte Carlo methods are able to exploit trial wave functions of arbitrary forms. Any wave function that is physical and for which the value, gradient and laplacian of the wave function may be efficiently computed can be used. The power of Quantum Monte Carlo methods lies in the flexibility of the form of the trial wave function.

It is important that the trial wave function satisfies as many known properties of the exact wave function as possible. A good trial wave function should exhibit much of the same features as does the exact wave function. Especially, it should be well-defined at the origin, that is $\Psi(|\mathbf{R}| = 0) \neq 0$, and its derivative at the origin should also be well-defined . One possible guideline in choosing the trial wave function is the use of constraints about the behavior of the wave function when the distance between one electron and the nucleus or two electrons approaches zero. These constraints are the so-called “cusp conditions” and are related to the derivatives of the wave function.

To see this, let us single out one of the electrons in the helium atom and assume that this electron is close to the nucleus, i.e., $r_1 \to 0$. We assume also that the two electrons are far from each other and that $r_2 \neq 0$. The local energy can then be written as

$$E_L(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})} H\psi_T(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})}\left(-\frac{1}{2}\nabla_1^2 - \frac{Z}{r_1}\right)\psi_T(\mathbf{R}) + \text{finite terms}. \tag{11.96}$$

Writing out the kinetic energy term in the spherical coordinates of electron 1, we arrive at the following expression for the local energy

$$E_L(R) = \frac{1}{\mathcal{R}_T(r_1)}\left(-\frac{1}{2}\frac{d^2}{dr_1^2} - \frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1}\right)\mathcal{R}_T(r_1) + \text{finite terms}, \tag{11.97}$$

where $\mathcal{R}_T(r_1)$ is the radial part of the wave function for electron 1. We have also used that the orbital momentum of electron 1 is $l = 0$. For small values of r_1, the terms which dominate are

$$\lim_{r_1 \to 0} E_L(R) = \frac{1}{\mathcal{R}_T(r_1)}\left(-\frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1}\right)\mathcal{R}_T(r_1), \tag{11.98}$$

since the second derivative does not diverge due to the finiteness of Ψ at the origin. The latter implies that in order for the kinetic energy term to balance the divergence in the potential term, we must have

$$\frac{1}{\mathcal{R}_T(r_1)}\frac{d\mathcal{R}_T(r_1)}{dr_1} = -Z, \tag{11.99}$$

implying that

$$\mathcal{R}_T(r_1) \propto e^{-Zr_1}. \tag{11.100}$$

A similar condition applies to electron 2 as well. For orbital momenta $l > 0$ we have (show this!)

$$\frac{1}{\mathcal{R}_T(r)}\frac{d\mathcal{R}_T(r)}{dr} = -\frac{Z}{l+1}. \tag{11.101}$$

Another constraint on the wave function is found for two electrons approaching each other. In this case it is the dependence on the separation r_{12} between the two electrons which has to reflect the correct behavior in the limit $r_{12} \to 0$. The resulting radial equation for the r_{12} dependence is the same for the electron-nucleus case, except that the attractive Coulomb interaction between the nucleus and the electron is replaced by a repulsive interaction and the kinetic energy term is twice as large. We obtain then

$$\lim_{r_{12}\to 0} E_L(R) = \frac{1}{\mathcal{R}_T(r_{12})}\left(-\frac{4}{r_{12}}\frac{d}{dr_{12}} + \frac{2}{r_{12}}\right)\mathcal{R}_T(r_{12}), \tag{11.102}$$

with still $l = 0$. This yields the so-called 'cusp'-condition

$$\frac{1}{\mathcal{R}_T(r_{12})}\frac{d\mathcal{R}_T(r_{12})}{dr_{12}} = \frac{1}{2}, \tag{11.103}$$

while for $l > 0$ we have

$$\frac{1}{\mathcal{R}_T(r_{12})}\frac{d\mathcal{R}_T(r_{12})}{dr_{12}} = \frac{1}{2(l+1)}. \tag{11.104}$$

For general systems containing more than two electrons, we have this condition for each electron pair ij.

Based on these consideration, a possible trial wave function which ignores the 'cusp'-condition between the two electrons is

$$\psi_T(\mathbf{R}) = e^{-\alpha(r_1+r_2)}, \tag{11.105}$$

where $r_{1,2}$ are dimensionless radii and α is a variational parameter which is to be interpreted as an effective charge.

A possible trial wave function which also reflects the 'cusp'-condition between the two electrons is

$$\psi_T(\mathbf{R}) = e^{-\alpha(r_1+r_2)}e^{r_{12}/2}. \tag{11.106}$$

The last equation can be generalized to

$$\psi_T(\mathbf{R}) = \phi(\mathbf{r}_1)\phi(\mathbf{r}_2)\dots\phi(\mathbf{r}_N)\prod_{i<j} f(r_{ij}), \tag{11.107}$$

for a system with N electrons or particles. The wave function $\phi(\mathbf{r}_i)$ is the single-particle wave function for particle i, while $f(r_{ij})$ account for more complicated two-body correlations. For the helium atom, we placed both electrons in the hydrogenic orbit $1s$. We know that the ground state for the helium atom has a symmetric spatial part, while the spin wave function is anti-symmetric in order to obey the Pauli principle. In the present case we need not to deal with spin degrees of freedom, since we are mainly trying to reproduce the ground state of the system. However, adopting such a single-particle representation for the individual electrons means that for atoms beyond helium, we cannot continue to place electrons in the lowest hydrogenic orbit. This is a consenquence of the Pauli principle, which states that the total wave function for a system of identical particles such as fermions, has to be anti-symmetric. The program we include below can use either Eq. (11.105) or Eq. (11.106) for the trial wave function. One or two electrons can be placed in the lowest hydrogen orbit, implying that the program can only be used for studies of the ground state of hydrogen or helium.

11.5.5 Program example for atomic systems

The variational Monte Carlo algorithm consists of two distinct phases. In the first a walker, a single electron in our case, consisting of an initially random set of electron positions is propagated according to the Metropolis algorithm, in order to equilibrate it and begin sampling . In the second phase, the walker continues to be moved, but energies and other observables are also accumulated for later averaging and statistical analysis. In the program below, the electrons are moved individually and not as a whole configuration. This improves the efficiency of the algorithm in larger systems, where configuration moves require increasingly small steps to maintain the acceptance ratio.

The main part of the code contains calls to various functions, setup and declarations of arrays etc. Note that we have defined a fixed step length h for the numerical computation of the second derivative of the kinetic energy. Furthermore, we perform the Metropolis test when we have moved all electrons. This should be compared to the case where we move one electron at the time and perform the Metropolis test. The latter is similar to the algorithm for the Ising model discussed in the previous chapter. A more detailed discussion and better statistical treatments and analyses are discussed in chapters 17 and 16.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter11/cpp/program1.cpp

```
// Variational Monte Carlo for atoms with up to two electrons
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace  std;
// output file as global variable
ofstream ofile;
// the step length and its squared inverse for the second derivative
#define h 0.001
#define h2 1000000

// declaraton of functions

// Function to read in data from screen, note call by reference
void initialise(int&, int&, int&, int&, int&, int&, double&) ;

// The Mc sampling for the variational Monte Carlo
void  mc_sampling(int, int, int, int, int, int, double, double *, double *);

// The variational wave function
double  wave_function(double **, double, int, int);

// The local energy
double  local_energy(double **, double, double, int, int, int);

// prints to screen the results of the calculations
void  output(int, int, int, double *, double *);


// Begin of main program

//int main()
int main(int argc, char* argv[])
```

```
{
  char *outfilename;
  int number_cycles, max_variations, thermalization, charge;
  int dimension, number_particles;
  double step_length;
  double *cumulative_e, *cumulative_e2;

  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  //   Read in data
  initialise(dimension, number_particles, charge,
             max_variations, number_cycles,
             thermalization, step_length) ;
  cumulative_e = new double[max_variations+1];
  cumulative_e2 = new double[max_variations+1];

  //  Do the mc sampling
  mc_sampling(dimension, number_particles, charge,
              max_variations, thermalization,
              number_cycles, step_length, cumulative_e, cumulative_e2);
  // Print out results
  output(max_variations, number_cycles, charge, cumulative_e, cumulative_e2)
     ;
  delete [] cumulative_e; delete [] cumulative_e;
  ofile.close();  // close output file
  return 0;
}
```

The implementation of the brute force Metropolis algorithm is shown in the next function. Here we have a loop over the variational variables α. It calls two functions, one to compute the wave function and one to update the local energy.

```
// Monte Carlo sampling with the Metropolis algorithm

void mc_sampling(int dimension, int number_particles, int charge,
                 int max_variations,
                 int thermalization, int number_cycles, double step_length,
                 double *cumulative_e, double *cumulative_e2)
{
  int cycles, variate, accept, dim, i, j;
  long idum;
  double wfnew, wfold, alpha, energy, energy2, delta_e;
  double **r_old, **r_new;
  alpha = 0.5*charge;
  idum=-1;
```

```
// allocate matrices which contain the position of the particles
r_old = (double **) matrix( number_particles, dimension, sizeof(double));
r_new = (double **) matrix( number_particles, dimension, sizeof(double));
for (i = 0; i < number_particles; i++) {
  for ( j=0; j < dimension; j++) {
    r_old[i][j] = r_new[i][j] = 0;
  }
}
// loop over variational parameters
for (variate=1; variate <= max_variations; variate++){
  // initialisations of variational parameters and energies
  alpha += 0.1;
  energy = energy2 = 0; accept =0; delta_e=0;
  //  initial trial position, note calling with alpha
  //  and in three dimensions
  for (i = 0; i < number_particles; i++) {
    for ( j=0; j < dimension; j++) {
      r_old[i][j] = step_length*(ran1(&idum)-0.5);
    }
  }
  wfold = wave_function(r_old, alpha, dimension, number_particles);
  // loop over monte carlo cycles
  for (cycles = 1; cycles <= number_cycles+thermalization; cycles++){
    // new position
    for (i = 0; i < number_particles; i++) {
      for ( j=0; j < dimension; j++) {
        r_new[i][j] = r_old[i][j]+step_length*(ran1(&idum)-0.5);
      }
    }
    wfnew = wave_function(r_new, alpha, dimension, number_particles);
    // Metropolis test
    if(ran1(&idum) <= wfnew*wfnew/wfold/wfold ) {
      for (i = 0; i < number_particles; i++) {
        for ( j=0; j < dimension; j++) {
          r_old[i][j]=r_new[i][j];
        }
      }
      wfold = wfnew;
      accept = accept+1;
    }
    // compute local energy
    if ( cycles > thermalization ) {
      delta_e = local_energy(r_old, alpha, wfold, dimension,
                             number_particles, charge);
      // update energies
      energy += delta_e;
      energy2 += delta_e*delta_e;
    }
  }   // end of loop over MC trials
  cout << "variational parameter= " << alpha
       << " accepted steps= " << accept << endl;
  // update the energy average and its squared
  cumulative_e[variate] = energy/number_cycles;
```

```
    cumulative_e2[variate] = energy2/number_cycles;

  }    // end of loop over variational  steps
  free_matrix((void **) r_old); // free memory
  free_matrix((void **) r_new); // free memory
}   // end mc_sampling function
```

The wave function is in turn defined in the next function. Here we limit ourselves to a function which consists only of the product of single-particle wave functions.

```
// Function to compute the squared wave function, simplest form

double  wave_function(double **r, double alpha,int dimension, int
    number_particles)
{
  int i, j, k;
  double wf, argument, r_single_particle, r_12;

  argument = wf = 0;
  for (i = 0; i < number_particles; i++) {
    r_single_particle = 0;
    for (j = 0; j < dimension; j++) {
      r_single_particle  += r[i][j]*r[i][j];
    }
    argument += sqrt(r_single_particle);
  }
  wf = exp(-argument*alpha) ;
  return wf;
}
```

Finally, the local energy is computed using a numerical derivation for the kinetic energy. We use the familiar expression derived in Eq. (3.4), that is

$$f_0'' = \frac{f_h - 2f_0 + f_{-h}}{h^2},$$

in order to compute

$$-\frac{1}{2\psi_T(\mathbf{R})}\nabla^2\psi_T(\mathbf{R}). \tag{11.108}$$

The variable h is a chosen step length. For helium, since it is rather easy to evaluate the local energy, the above is an unnecessary complication. However, for many-electron or other many-particle systems, the derivation of an analytic expression for the kinetic energy can be quite involved, and the numerical evaluation of the kinetic energy using Eq. (3.4) may result in a simpler code and/or even a faster one.

```
// Function to calculate the local energy with num derivative

double  local_energy(double **r, double alpha, double wfold, int dimension,
                        int number_particles, int charge)
{
  int i, j , k;
  double e_local, wfminus, wfplus, e_kinetic, e_potential, r_12,
    r_single_particle;
  double **r_plus, **r_minus;
```

```
  // allocate matrices which contain the position of the particles
  // the function matrix is defined in the progam library
  r_plus = (double **) matrix( number_particles, dimension, sizeof(double));
  r_minus = (double **) matrix( number_particles, dimension, sizeof(double))
      ;
  for (i = 0; i < number_particles; i++) {
    for ( j=0; j < dimension; j++) {
      r_plus[i][j] = r_minus[i][j] = r[i][j];
    }
  }
  // compute the kinetic energy
  e_kinetic = 0;
  for (i = 0; i < number_particles; i++) {
    for (j = 0; j < dimension; j++) {
      r_plus[i][j] = r[i][j]+h;
      r_minus[i][j] = r[i][j]-h;
      wfminus = wave_function(r_minus, alpha, dimension, number_particles);
      wfplus  = wave_function(r_plus, alpha, dimension, number_particles);
      e_kinetic -= (wfminus+wfplus-2*wfold);
      r_plus[i][j] = r[i][j];
      r_minus[i][j] = r[i][j];
    }
  }
  // include electron mass and hbar squared and divide by wave function
  e_kinetic = 0.5*h2*e_kinetic/wfold;
  // compute the potential energy
  e_potential = 0;
  // contribution from electron-proton potential
  for (i = 0; i < number_particles; i++) {
    r_single_particle = 0;
    for (j = 0; j < dimension; j++) {
      r_single_particle += r[i][j]*r[i][j];
    }
    e_potential -= charge/sqrt(r_single_particle);
  }
  // contribution from electron-electron potential
  for (i = 0; i < number_particles-1; i++) {
    for (j = i+1; j < number_particles; j++) {
      r_12 = 0;
      for (k = 0; k < dimension; k++) {
        r_12 += (r[i][k]-r[j][k])*(r[i][k]-r[j][k]);
      }
      e_potential += 1/sqrt(r_12);
    }
  }
  free_matrix((void **) r_plus); // free memory
  free_matrix((void **) r_minus);
  e_local = e_potential+e_kinetic;
  return e_local;
}
```

The remaining part of the program consists of the output and initialize functions and is not listed here.

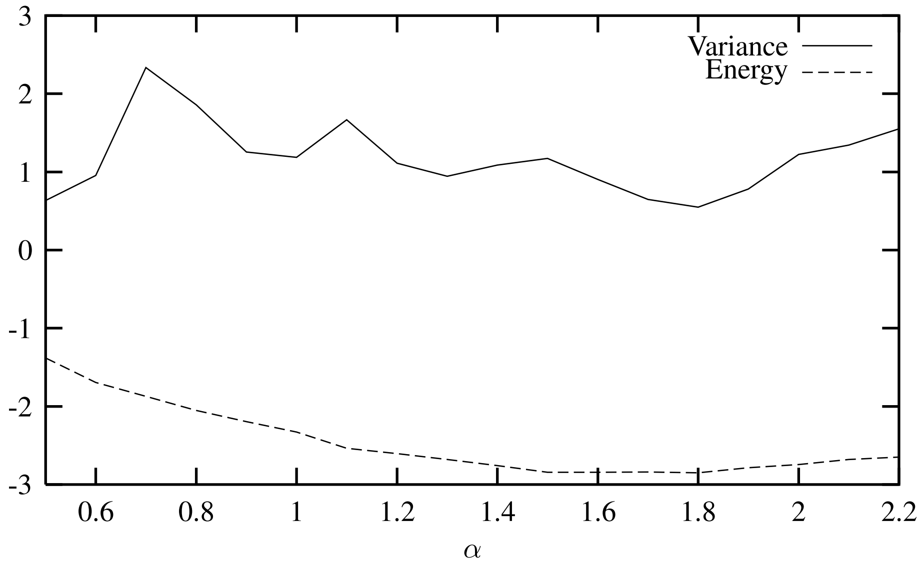


Figure 11.3: Result for ground state energy of the helium atom using Eq. (11.105) for the trial wave function. The variance is also plotted. A total of 100000 Monte Carlo moves were used with a step length of 2 Bohr radii.

The way we have rewritten Schrödinger's equation results in energies given in atomic units. If we wish to convert these energies into more familiar units like electronvolt (eV), we have to multiply our reults with $2E_0$ where $E_0 = 13.6$ eV, the binding energy of the hydrogen atom. Using Eq. (11.105) for the trial wave function, we obtain an energy minimum at $\alpha \approx 1.75$. The ground state is $E = -2.85$ in atomic units or $E = -77.5$ eV. The experimental value is -78.8 eV. Obviously, improvements to the wave function such as including the 'cusp'-condition for the two electrons as well, see Eq. (11.106), could improve our agreement with experiment. Such an implementation is the topic for the next project.

We note that the effective charge is less than the charge of the nucleus. We can interpret this reduction as an effective way of incorporating the repulsive electron-electron interaction. Finally, since we do not have the exact wave function, we see from Fig. 11.3 that the variance is not zero at the energy minimum. Techniques such as importance sampling, to be contrasted to the brute force Metropolis sampling used here, and various optimization techniques of the variance and the energy, will be discussed under advanced topics, see chapter 17.

11.5.6 Helium and beyond

We need to establish some rules regarding the construction of physically reliable wave-functions for systems with more than one electron. The *Pauli principle*, after Wolfgang Pauli states that *The total wave function must be antisymmetric under the interchange of any pair of identical fermions and symmetric under the interchange of any pair of identical bosons.*

A result of the Pauli principle is the so-called Pauli exclusion principle which that *no two electrons can occupy the same state.*

Overall wave functions that satisfy the Pauli principle are often written as *Slater Determinants*.

The Slater Determinant

For the helium atom we assumed that the two electrons were both in the $1s$ state. This fulfills the Pauli exclusion principle as the two electrons in the ground state have different intrinsic spin. However, the wave function we discussed above was not antisymmetric with respect to an interchange of the different electrons. This is not totally true as we only included the spatial part of the wave function. For the helium ground state the spatial part of the wave function is symmetric and the spin part is antisymmetric. The product is therefore antisymmetric as well. The Slater-determinant consists of single-particle *spin-orbitals*; joint spin-space states of the electrons

$$\Psi_{1s}^{\uparrow}(1) = \Psi_{1s}(1) \uparrow (1),$$

and similarly

$$\Psi_{1s}^{\downarrow}(2) = \Psi_{1s}(2) \downarrow (2).$$

Here the two spin functions are given by

$$\uparrow (I) = \begin{cases} 1 & \text{if } m_s(I) = \frac{1}{2} \\ 0 & \text{if } m_s(I) = -\frac{1}{2} \end{cases},$$

and

$$\downarrow (I) = \begin{cases} 0 & \text{if } m_s(I) = \frac{1}{2} \\ 1 & \text{if } m_s(I) = -\frac{1}{2} \end{cases}, \tag{11.109}$$

with $I = 1, 2$. The ground state can then be expressed by the following determinant

$$\Psi(1,2) = \frac{1}{\sqrt(2)} \begin{vmatrix} \Psi_{1s}(1) \uparrow (1) & \Psi_{1s}(2) \uparrow (2) \\ \Psi_{1s}(1) \downarrow (1) & \Psi_{1s}(2) \downarrow (2) \end{vmatrix}.$$

This is an example of a *Slater determinant*. This determinant is antisymmetric since particle interchange is identical to an interchange of the two columns. For the ground state the spatial wave-function is symmetric. Therefore we simply get

$$\Psi(1,2) = \Psi_{1s}(1)\Psi_{1s}(2) \left[\uparrow (1) \downarrow (2) - \uparrow (2) \downarrow (1)\right].$$

The spin part of the wave-function is here antisymmetric. This has no effect when calculating physical observables because the sign of the wave function is squared in all expectation values.

The general form of a Slater determinant composed of n orthonormal orbitals $\{\phi_i\}$ is

$$\Psi = \frac{1}{\sqrt{N!}} \begin{vmatrix} \phi_1(1) & \phi_1(2) & \dots & \phi_1(N) \\ \phi_2(1) & \phi_2(2) & \dots & \phi_2(N) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_N(1) & \phi_N(2) & \dots & \phi_N(N) \end{vmatrix}. \tag{11.110}$$

The introduction of the Slater determinant is very important for the treatment of many-body systems, and is the principal building block for various variational wave functions. As long as we express the wave-function in terms of either one Slater determinant or a linear combination of several Slater determinants, the Pauli principle is satisfied. When constructing many-electron wave functions this picture provides an easy way to include many of the physical features. One problem with the Slater matrix is that it is computationally demanding. Limiting the number of calculations will be one of the most important issues concerning the implementation of the Slater determinant. This will be discussed in detail in chapter 17. Chapter 17 is dedicated to the discussion of advanced many-body methods for solving Schrödinger's equation.

11.6 The H_2^+ molecule

The H_2^+ molecule consists of two protons and one electron, with binding energy $E_B = -2.8$ eV and an equilibrium position $r_0 = 0.106$ nm between the two protons.

We define our system through the following variables. The electron is at a distance $\mathbf{r}$ from a chosen origo, one of the protons is at the distance $-\mathbf{R}/2$ while the other one is placed at $\mathbf{R}/2$ from origo, resulting in a distance to the electron of $\mathbf{r} - \mathbf{R}/2$ and $\mathbf{r} + \mathbf{R}/2$, respectively.

In our solution of Schrödinger's equation for this system we are going to neglect the kinetic energies of the protons, since they are 2000 times heavier than the electron. We assume thus that their velocities are negligible compared to the velocity of the electron. In addition we omit contributions from nuclear forces, since they act at distances of several orders of magnitude smaller than the equilibrium position.

We can then write Schrödinger's equation as follows

$$\left\{-\frac{\hbar^2\nabla_r^2}{2m_e} - \frac{ke^2}{|\mathbf{r}-\mathbf{R}/2|} - \frac{ke^2}{|\mathbf{r}+\mathbf{R}/2|} + \frac{ke^2}{R}\right\}\psi(\mathbf{r},\mathbf{R}) = E\psi(\mathbf{r},\mathbf{R}), \tag{11.111}$$

where the first term is the kinetic energy of the electron, the second term is the potential energy the electron feels from the proton at $-\mathbf{R}/2$ while the third term arises from the potential energy contribution from the proton at $\mathbf{R}/2$. The last term arises due to the repulsion between the two protons. In Fig. 11.4 we show a plot of the potential energy

$$V(\mathbf{r},\mathbf{R}) = -\frac{ke^2}{|\mathbf{r}-\mathbf{R}/2|} - \frac{ke^2}{|\mathbf{r}+\mathbf{R}/2|} + \frac{ke^2}{R}. \tag{11.112}$$

Here we have fixed $|\mathbf{R}| = 2a_0$ og $|\mathbf{R}| = 8a_0$, being 2 and 8 Bohr radii, respectively. Note that in the region between $|\mathbf{r}| = -|\mathbf{R}|/2$ (units are r/a_0 in this figure, with $a_0 = 0.0529$) and $|\mathbf{r}| = |\mathbf{R}|/2$ the electron can tunnel through the potential barrier. Recall that $-\mathbf{R}/2$ og $\mathbf{R}/2$ correspond to the positions of the two protons. We note also that if R is increased, the potential becomes less attractive. This has consequences for the binding energy of the molecule. The binding energy decreases as the distance $\mathbf{R}$ increases. Since the potential is symmetric with respect to the interchange of $\mathbf{R} \to -\mathbf{R}$ and $\mathbf{r} \to -\mathbf{r}$ it means that the probability for the electron to move from one proton to the other must be equal in both directions. We can say that the electron shares it's time between both protons.

With this caveat, we can now construct a model for simulating this molecule. Since we have only one elctron, we could assume that in the limit $R \to \infty$, i.e., when the distance between the two protons is large, the electron is essentially bound to only one of the protons. This should correspond to a hydrogen atom. As a trial wave function, we could therefore use the electronic wave function for the ground state of hydrogen, namely

$$\psi_{100}(r) = \left(\frac{1}{\pi a_0^3}\right)^{1/2} e^{-r/a_0}. \tag{11.113}$$

Since we do not know exactly where the electron is, we have to allow for the possibility that the electron can be coupled to one of the two protons. This form includes the 'cusp'-condition discussed in the previous section. We define thence two hydrogen wave functions

$$\psi_1(\mathbf{r},\mathbf{R}) = \left(\frac{1}{\pi a_0^3}\right)^{1/2} e^{-|\mathbf{r}-\mathbf{R}/2|/a_0}, \tag{11.114}$$

and

$$\psi_2(\mathbf{r},\mathbf{R}) = \left(\frac{1}{\pi a_0^3}\right)^{1/2} e^{-|\mathbf{r}+\mathbf{R}/2|/a_0}. \tag{11.115}$$

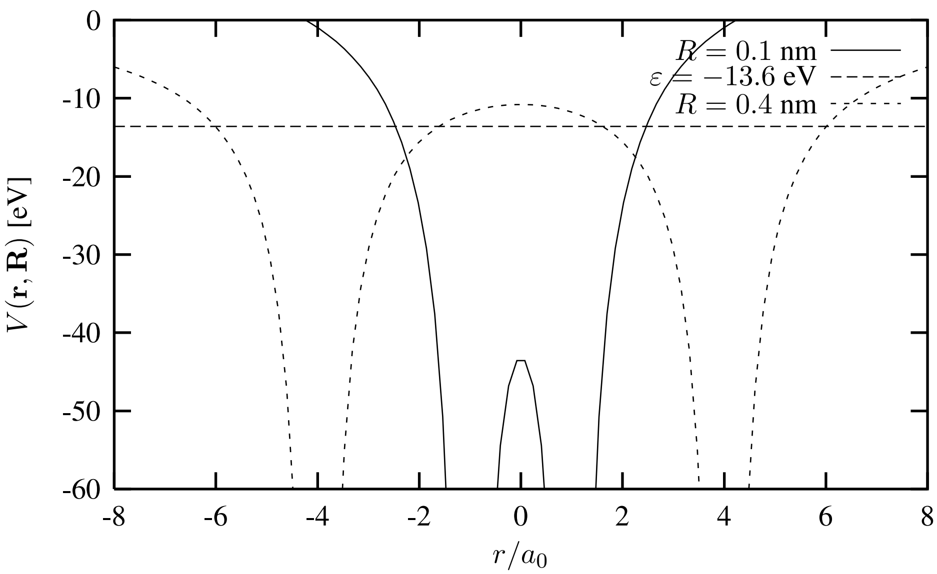


Figure 11.4: Plot of $V(r, R)$ for $|\mathbf{R}|$=0.1 and 0.4 nm. Units along the x-axis are r/a_0 . The straight line is the binding energy of the hydrogen atom, $\varepsilon = -13.6$ eV.

Based on these two wave functions, which represent where the electron can be, we attempt at the following linear combination

$$\psi_{\pm}(\mathbf{r}, \mathbf{R}) = C_{\pm}\left(\psi_1(\mathbf{r}, \mathbf{R}) \pm \psi_2(\mathbf{r}, \mathbf{R})\right), \tag{11.116}$$

with $C_{\pm}$ a constant. Based on this discussion, we add a second electron in order to simulate the H_2 molecule. That is the topic for project 11.3.

11.7 Exercises and projects

Project 11.1: Studies of light Atoms

The aim of this project is to test the variational Monte Carlo apppled to light atoms. We will test different trial wave function Ψ_T. The systems we study are atoms consisting of two electrons only, such as the helium atom, Li_{II} and Be_{III}. The atom Li_{II} has two electrons and $Z = 3$ while Be_{III} has $Z = 4$ but still two electrons only. A general ansatz for the trial wave function is

$$\psi_T(\mathbf{R}) = \phi(\mathbf{r}_1)\phi(\mathbf{r}_2)f(r_{12}). \tag{11.117}$$

For all systems we assume that the one-electron wave functions $\phi(\mathbf{r}_i)$ are described by the an elecron in the lowest hydrogen orbital $1s$.

The specific trial functions we study are

$$\psi_{T1}(\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_{12}}) = \exp\left(-\alpha(r_1 + r_2)\right), \tag{11.118}$$

where α is the variational parameter,

$$\psi_{T2}(\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_{12}}) = \exp\left(-\alpha(r_1 + r_2)\right)(1 + \beta r_{12}), \tag{11.119}$$

with β as a new variational parameter and

$$\psi_{T3}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}}) = \exp\left(-\alpha(r_1+r_2)\right)\exp\left(\frac{r_{12}}{2(1+\beta r_{12})}\right). \tag{11.120}$$

a) Find the analytic expressions for the local energy for the above trial wave function for the helium atom. Study the behavior of the local energy with these functions in the limits $r_1 \to 0$, $r_2 \to 0$ and $r_{12} \to 0$.

b) Compute

$$\langle \widehat{\mathbf{H}} \rangle = \frac{\int d\mathbf{R}\Psi^*_T(\mathbf{R})\widehat{\mathbf{H}}(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi^*_T(\mathbf{R})\Psi_T(\mathbf{R})}, \tag{11.121}$$

for the helium atom using the variational Monte Carlo method employing the Metropolis algorithm to sample the different states using the trial wave function $\psi_{T1}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})$. Compare your results with the analytic expression

$$\langle \widehat{\mathbf{H}} \rangle = \frac{\hbar^2}{m_e}\alpha^2 - \frac{27}{32}\frac{e^2}{\pi\epsilon_0}\alpha. \tag{11.122}$$

c) Use the optimal value of α from the previous point to compute the ground state of the helium atom using the other two trial wave functions $\psi_{T2}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})$ and $\psi_{T3}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})$. In this case you have to vary both α and β. Explain briefly which function $\psi_{T1}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})$, $\psi_{T2}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})$ and $\psi_{T3}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})$ is the best.

d) Use the optimal value for all parameters and all wave functions to compute the expectation value of the mean distance $\langle r_{12}\rangle$ between the two electrons. Comment your results.

e) We will now repeat point 1c), but we replace the helium atom with the ions Li_{II} and Be_{III}. Perform first a variational calculation using the first ansatz for the trial wave function $\psi_{T1}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})$ in order to find an optimal value for α. Use then this value to start the variational calculation of the energy for the wave functions $\psi_{T2}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})$ and $\psi_{T3}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})$. Comment your results.

Project 11.2: Ground state of He, Be and Ne

The task here is to perform a variational Monte Carlo calculation of the ground state energy of the atoms He, Be and Ne.

a) Here we limit the attention to He and employ the following trial wave function

$$\psi_{T}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}}) = \exp\left(-\alpha(r_1+r_2)\right)\exp\left(\frac{r_{12}}{2(1+\beta r_{12})}\right), \tag{11.123}$$

with α and β as variational parameters. The interaction is

$$V(r_1,r_2) = -\frac{2}{r_1} - \frac{2}{r_2} + \frac{1}{r_{12}}, \tag{11.124}$$

yielding the following hamiltonian for the helium atom

$$\hat{H} = -\frac{\nabla_1^2}{2} - \frac{\nabla_2^2}{2} - \frac{2}{r_1} - \frac{2}{r_2} + \frac{1}{r_{12}}. \tag{11.125}$$

Your task is to perform a Variational Monte Carlo calculation using the Metropolis algorithm to compute the integral

$$\langle H \rangle = \frac{\int d\mathbf{R}\psi^*_T(\mathbf{R})H(\mathbf{R})\psi_T(\mathbf{R})}{\int d\mathbf{R}\psi^*_T(\mathbf{R})\psi_T(\mathbf{R})}. \tag{11.126}$$

b) We turn the attention to the ground state energy for the Be atom. In this case the trial wave function is given by

$$\psi_T(\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_3}, \mathbf{r_4}) = Det\left(\phi_1(\mathbf{r_1}), \phi_2(\mathbf{r_2}), \phi_3(\mathbf{r_3}), \phi_4(\mathbf{r_4})\right) \prod_{i<j}^{4} \exp\left(\frac{r_{ij}}{2(1+\beta r_{ij})}\right), \tag{11.127}$$

where the Det is a Slater determinant and the single-particle wave functions are the hydrogen wave functions for the $1s$ and $2s$ orbitals. Their form within the variational ansatz is given by

$$\phi_{1s}(\mathbf{r_i}) = e^{-\alpha r_i}, \tag{11.128}$$

and

$$\phi_{2s}(\mathbf{r_i}) = (2 - \alpha r_i)\, e^{-\alpha r_i/2}. \tag{11.129}$$

Set up the expression for the Slater determinant and perform a variational calculation with α and β as variational parameters.

c) Now we compute the ground state energy for the Neon atom following the same steps as in a) and b) but with the trial wave function

$$\psi_T(\mathbf{r_1}, \mathbf{r_2}, \dots, \mathbf{r_{10}}) = Det\left(\phi_1(\mathbf{r_1}), \phi_2(\mathbf{r_2}), \dots, \phi_{10}(\mathbf{r_{10}})\right) \prod_{i<j}^{10} \exp\left(\frac{r_{ij}}{2(1+\beta r_{ij})}\right), \tag{11.130}$$

Set up the expression for the Slater determinant and repeat steps a) and b) including the Slater determinant. The variational parameters are still α and β only. In this case you need to include the $2p$ wave function as well. It is given as

$$\phi_{2p}(\mathbf{r_i}) = \alpha \mathbf{r_i} e^{-\alpha r_i/2}. \tag{11.131}$$

Observe that $r_i = \sqrt{r_{i_x}^2 + r_{i_y}^2 + r_{i_z}^2}$.

Project 11.3: the H_2 molecule

The H_2 molecule consists of two protons and two electrons with a ground state energy $E = -1.17460$ a.u. and equilibrium distance between the two hydrogen atoms of $r_0 = 1.40$ Bohr radii. We define our systems using the following variables. Origo is chosen to be halfway between the two protons. The distance from proton 1 is defined as $-\mathbf{R}/2$ whereas proton 2 has a distance $\mathbf{R}/2$. Calculations are performed for fixed distances $\mathbf{R}$ between the two protons.

Electron 1 has a distance r_1 from the chose origo, while electron 2 has a distance r_2. The kinetic energy operator becomes then

$$-\frac{\nabla_1^2}{2} - \frac{\nabla_2^2}{2}. \tag{11.132}$$

The distance between the two electrons is $r_{12} = |\mathbf{r}_1 - \mathbf{r}_2|$. The repulsion between the two electrons results in a potential energy term given by

$$+\frac{1}{r_{12}}. \tag{11.133}$$

In a similar way we obtain a repulsive contribution from the interaction between the two protons given by

$$+\frac{1}{|\mathbf{R}|}, \tag{11.134}$$

where $\mathbf{R}$ is the distance between the two protons. To obtain the final potential energy we need to include the attraction the electrons feel from the protons. To model this, we need to define the distance between the electrons and the two protons. If we model this along a chosen z-akse with electron 1 placed at a distance $\mathbf{r}_1$ from a chose origo, one proton at $-\mathbf{R}/2$ and the other at $\mathbf{R}/2$, the distance from proton 1 to electron 1 becomes

$$\mathbf{r}_{1p1} = \mathbf{r}_1 + \mathbf{R}/2, \tag{11.135}$$

and

$$\mathbf{r}_{1p2} = \mathbf{r}_1 - \mathbf{R}/2, \tag{11.136}$$

from proton 2. Similarly, for electron 2 we obtain

$$\mathbf{r}_{2p1} = \mathbf{r}_2 + \mathbf{R}/2, \tag{11.137}$$

and

$$\mathbf{r}_{2p2} = \mathbf{r}_2 - \mathbf{R}/2. \tag{11.138}$$

These four distances define the attractive contributions to the potential energy

$$-\frac{1}{r_{1p1}} - \frac{1}{r_{1p2}} - \frac{1}{r_{2p1}} - \frac{1}{r_{2p2}}. \tag{11.139}$$

We can then write the total Hamiltonian as

$$\widehat{\mathbf{H}} = -\frac{\nabla_1^2}{2} - \frac{\nabla_2^2}{2} - \frac{1}{r_{1p1}} - \frac{1}{r_{1p2}} - \frac{1}{r_{2p1}} - \frac{1}{r_{2p2}} + \frac{1}{r_{12}} + \frac{1}{|\mathbf{R}|}, \tag{11.140}$$

and if we choose $\mathbf{R} = 0$ we obtain the helium atom.

In this project we will use a trial wave function of the form

$$\psi_T(\mathbf{r_1}, \mathbf{r_2}, \mathbf{R}) = \psi(\mathbf{r}_1, \mathbf{R})\psi(\mathbf{r}_2, \mathbf{R}) \exp\left(\frac{r_{12}}{2(1+\beta r_{12})}\right), \tag{11.141}$$

with the following trial wave function

$$\psi(\mathbf{r}_1, \mathbf{R}) = \left(\exp\left(-\alpha r_{1p1}\right) + \exp\left(-\alpha r_{1p2}\right)\right), \tag{11.142}$$

for electron 1 and

$$\psi(\mathbf{r}_2, \mathbf{R}) = \left(\exp\left(-\alpha r_{2p1}\right) + \exp\left(-\alpha r_{2p2}\right)\right). \tag{11.143}$$

The variational parameters are α and β.

One can show that in the limit where all distances approach zero that

$$\alpha = 1 + \exp\left(-R/\alpha\right), \tag{11.144}$$

resulting in β kas the only variational parameter. The last equation is a non-linear equation which we can solve with for example Newton's method discussed in chapter 5.

a) Find the local energy as function of R.

b) Set up and algorithm and write a program which computes the expectation value of $\langle \widehat{\mathbf{H}} \rangle$ using the variational Monte Carlo method with a brute force Metropolis sampling. For each inter-proton distance R you must find the parameter β which minimizes the energy. Plot the corresponding energy as function of the distance R between the protons.

c) Use thereafter the optimal parameter sets to compute the average distance $\langle r_{12} \rangle$ between the electrons where the energy as function of R exhibits its minimum. Comment your results.

d) We modify now the approximation for the wave functions of electrons 1 and 2 by subtracting the two terms instead of adding up, viz

$$\psi(\mathbf{r}_1, \mathbf{R}) = \left(\exp\left(-\alpha r_{1p1}\right) - \exp\left(-\alpha r_{1p2}\right)\right), \tag{11.145}$$

for electron 1

$$\psi(\mathbf{r}_2, \mathbf{R}) = \left(\exp\left(-\alpha r_{2p1}\right) - \exp\left(-\alpha r_{2p2}\right)\right), \tag{11.146}$$

for electron 2. Mathematically, this approach is equally viable as the previous one. Repeat your calculations from point b) and see if you can obtain an energy minimum as function of R. Comment your results.

Chapter 12

Eigensystems

12.1 Introduction

Together with linear equations and least squares, the third major problem in matrix computations deals with the algebraic eigenvalue problem. Here we limit our attention to the symmetric case. We focus in particular on two similarity transformations, the Jacobi method, the famous QR algoritm with Householder's method for obtaining a triangular matrix and Francis' algorithm for the final eigenvalues. Our presentation follows closely that of Golub and Van Loan, see Ref. [27].

12.2 Eigenvalue problems

Let us consider the matrix $\mathbf{A}$ of dimension n. The eigenvalues of $\mathbf{A}$ is defined through the matrix equation

$$\mathbf{A}\mathbf{x}^{(\nu)} = \lambda^{(\nu)}\mathbf{x}^{(\nu)}, \tag{12.1}$$

where $\lambda^{(\nu)}$ are the eigenvalues and $\mathbf{x}^{(\nu)}$ the corresponding eigenvectors. Unless otherwise stated, when we use the wording eigenvector we mean the right eigenvector. The left eigenvector is defined as

$$\mathbf{x}^{(\nu)}{}_L\mathbf{A} = \lambda^{(\nu)}\mathbf{x}^{(\nu)}{}_L$$

The above right eigenvector problem is equivalent to a set of n equations with n unknowns x_i

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = &\quad \lambda x_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = &\quad \lambda x_2 \\
\dots &\quad \dots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = &\quad \lambda x_n.
\end{aligned}$$

We can rewrite Eq. (12.1) as

$$\left(\mathbf{A} - \lambda^{(\nu)}I\right)\mathbf{x}^{(\nu)} = 0,$$

with I being the unity matrix. This equation provides a solution to the problem if and only if the determinant is zero, namely

$$\left|\mathbf{A} - \lambda^{(\nu)}\mathbf{I}\right| = 0,$$

which in turn means that the determinant is a polynomial of degree n in λ and in general we will have n distinct zeros. The eigenvalues of a matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ are thus the n roots of its characteristic polynomial

$$P(\lambda) = det(\lambda\mathbf{I} - \mathbf{A}), \tag{12.2}$$

or

$$P(\lambda) = \prod_{i=1}^{n} (\lambda_i - \lambda). \tag{12.3}$$

The set of these roots is called the spectrum and is denoted as $\lambda(\mathbf{A})$. If $\lambda(\mathbf{A}) = \{\lambda_1, \lambda_2, \dots, \lambda_n\}$ then we have

$$det(\mathbf{A}) = \lambda_1\lambda_2 \dots \lambda_n,$$

and if we define the trace of $\mathbf{A}$ as

$$Tr(\mathbf{A}) = \sum_{i=1}^{n} a_{ii}$$

then $Tr(\mathbf{A}) = \lambda_1 + \lambda_2 + \dots + \lambda_n$.

Procedures based on these ideas can be used if only a small fraction of all eigenvalues and eigenvectors are required or if the matrix is on a tridiagonal form, but the standard approach to solve Eq. (12.1) is to perform a given number of similarity transformations so as to render the original matrix $\mathbf{A}$ in either a diagonal form or as a tridiagonal matrix which then can be be diagonalized by computational very effective procedures.

The first method leads us to Jacobi's method whereas the second one is given by Householder's algorithm for tridiagonal transformations. We will discuss both methods below.

12.3 Similarity transformations

In the present discussion we assume that our matrix is real and symmetric, that is $\mathbf{A} \in \mathbb{R}^{n\times n}$. The matrix $\mathbf{A}$ has n eigenvalues $\lambda_1 \dots \lambda_n$ (distinct or not). Let $\mathbf{D}$ be the diagonal matrix with the eigenvalues on the diagonal

$$\mathbf{D} = \begin{pmatrix} \lambda_1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \lambda_{n-1} & \\ 0 & \dots & \dots & \dots & \dots & 0 & \lambda_n \end{pmatrix}.$$

If $\mathbf{A}$ is real and symmetric then there exists a real orthogonal matrix $\mathbf{S}$ such that

$$\mathbf{S}^T\mathbf{A}\mathbf{S} = \mathrm{diag}(\lambda_1, \lambda_2, \dots, \lambda_n),$$

and for $j = 1 : n$ we have $\mathbf{A}\mathbf{S}(:, j) = \lambda_j\mathbf{S}(:, j)$. See chapter 8 of Ref. [27] for proof.

To obtain the eigenvalues of $\mathbf{A} \in \mathbb{R}^{n\times n}$, the strategy is to perform a series of similarity transformations on the original matrix $\mathbf{A}$, in order to reduce it either into a diagonal form as above or into a tridiagonal form.

We say that a matrix $\mathbf{B}$ is a similarity transform of $\mathbf{A}$ if

$$\mathbf{B} = \mathbf{S}^T\mathbf{A}\mathbf{S}, \qquad \text{where} \qquad \mathbf{S}^T\mathbf{S} = \mathbf{S}^{-1}\mathbf{S} = \mathbf{I}.$$

The importance of a similarity transformation lies in the fact that the resulting matrix has the same eigenvalues, but the eigenvectors are in general different. To prove this we start with the eigenvalue problem and a similarity transformed matrix $\mathbf{B}$.

$$\mathbf{Ax} = \lambda\mathbf{x} \qquad \text{and} \qquad \mathbf{B} = \mathbf{S}^T\mathbf{AS}.$$

We multiply the first equation on the left by $\mathbf{S}^T$ and insert $\mathbf{S}^T\mathbf{S} = \mathbf{I}$ between $\mathbf{A}$ and $\mathbf{x}$. Then we get

$$(\mathbf{S^TAS})(\mathbf{S^Tx}) = \lambda\mathbf{S^Tx}, \tag{12.4}$$

which is the same as

$$\mathbf{B}\left(\mathbf{S^Tx}\right) = \lambda\left(\mathbf{S^Tx}\right).$$

The variable λ is an eigenvalue of $\mathbf{B}$ as well, but with eigenvector $\mathbf{S^Tx}$.

The basic philosophy is to

- either apply subsequent similarity transformations so that

$$\mathbf{S_N^T} \dots \mathbf{S_1^T A S_1} \dots \mathbf{S_N} = \mathbf{D}, \tag{12.5}$$

- or apply subsequent similarity transformations so that $\mathbf{A}$ becomes tridiagonal. Thereafter, techniques for obtaining eigenvalues from tridiagonal matrices can be used.

Let us look at the first method, better known as Jacobi's method or Given's rotations.

12.4 Jacobi's method

Consider an $(n \times n)$ orthogonal transformation matrix

$$\mathbf{S} = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\ 0 & 0 & \dots & cos\theta & 0 & \dots & 0 & sin\theta \\ 0 & 0 & \dots & 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\ 0 & 0 & \dots & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & -sin\theta & \dots & \dots & 0 & cos\theta \end{pmatrix}$$

with property $\mathbf{S^T} = \mathbf{S^{-1}}$. It performs a plane rotation around an angle θ in the Euclidean $n-$dimensional space. It means that its matrix elements that differ from zero are given by

$$s_{kk} = s_{ll} = cos\theta, s_{kl} = -s_{lk} = -sin\theta, s_{ii} = -s_{ii} = 1 \quad i \neq k \quad i \neq l,$$

A similarity transformation

$$\mathbf{B} = \mathbf{S}^T\mathbf{AS},$$

results in

$$\begin{aligned} b_{ik} &= a_{ik}cos\theta - a_{il}sin\theta, i \neq k, i \neq l \\ b_{il} &= a_{il}cos\theta + a_{ik}sin\theta, i \neq k, i \neq l \\ b_{kk} &= a_{kk}cos^2\theta - 2a_{kl}cos\theta sin\theta + a_{ll}sin^2\theta \\ b_{ll} &= a_{ll}cos^2\theta + 2a_{kl}cos\theta sin\theta + a_{kk}sin^2\theta \\ b_{kl} &= (a_{kk} - a_{ll})cos\theta sin\theta + a_{kl}(cos^2\theta - sin^2\theta) \end{aligned}$$

The angle θ is arbitrary. The recipe is to choose θ so that all non-diagonal matrix elements b_{kl} become zero.

The algorithm is then quite simple. We perform a number of iterations untill the sum over the squared non-diagonal matrix elements are less than a prefixed test (ideally equal zero). The algorithm is more or less foolproof for all real symmetric matrices, but becomes much slower than methods based on tridiagonalization for large matrices.

The main idea is thus to reduce systematically the norm of the off-diagonal matrix elements of a matrix $\mathbf{A}$

$$\mathrm{off}(\mathbf{A}) = \sqrt{\sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n} a_{ij}^2}.$$

To demonstrate the algorithm, we consider the simple 2×2 similarity transformation of the full matrix. The matrix is symmetric, we single out $1 \leq k < l \leq n$ and use the abbreviations $c = \cos\theta$ and $s = \sin\theta$ to obtain

$$\left(\begin{array}{cc} b_{kk} & 0 \\ 0 & b_{ll} \end{array}\right) = \left(\begin{array}{cc} c & -s \\ s & c \end{array}\right)\left(\begin{array}{cc} a_{kk} & a_{kl} \\ a_{lk} & a_{ll} \end{array}\right)\left(\begin{array}{cc} c & s \\ -s & c \end{array}\right).$$

We require that the non-diagonal matrix elements $b_{kl} = b_{lk} = 0$, implying that

$$a_{kl}(c^2 - s^2) + (a_{kk} - a_{ll})cs = b_{kl} = 0.$$

If $a_{kl} = 0$ one sees immediately that $\cos\theta = 1$ and $\sin\theta = 0$.

The Frobenius norm of an orthogonal transformation is always preserved. The Frobenius norm is defined as

$$||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}|a_{ij}|^2}.$$

This means that for our 2×2 case we have

$$2a_{kl}^2 + a_{kk}^2 + a_{ll}^2 = b_{kk}^2 + b_{ll}^2,$$

which leads to

$$\mathrm{off}(\mathbf{B})^2 = ||\mathbf{B}||_F^2 - \sum_{i=1}^{n} b_{ii}^2 = \mathrm{off}(\mathbf{A})^2 - 2a_{kl}^2,$$

since

$$||\mathbf{B}||_F^2 - \sum_{i=1}^{n} b_{ii}^2 = ||\mathbf{A}||_F^2 - \sum_{i=1}^{n} a_{ii}^2 + (a_{kk}^2 + a_{ll}^2 - b_{kk}^2 - b_{ll}^2).$$

This results means that the matrix $\mathbf{A}$ moves closer to diagonal form for each transformation.

Defining the quantities $\tan\theta = t = s/c$ and

$$\tau = \frac{a_{kk} - a_{ll}}{2a_{kl}},$$

we obtain the quadratic equation

$$t^2 + 2\tau t - 1 = 0,$$

resulting in

$$t = -\tau \pm \sqrt{1 + \tau^2},$$

and c and s are easily obtained via

$$c = \frac{1}{\sqrt{1+t^2}},$$

and $s = tc$. Choosing t to be the smaller of the roots ensures that $|\theta| \leq \pi/4$ and has the effect of minimizing the difference between the matrices $\mathbf{B}$ and $\mathbf{A}$ since

$$||\mathbf{B}-\mathbf{A}||_F^2 = 4(1-c)\sum_{i=1,i\neq k,l}^{n}(a_{ik}^2+a_{il}^2)+\frac{2a_{kk}^2}{c^2}.$$

To implement the Jacobi algorithm we can proceed as follows

- Choose a tolerance ϵ, making it a small number, typically 10^{-8} or smaller.
- Setup a **while**-test where one compares the norm of the newly computed off-diagonal matrix elements

$$\mathrm{off}(\mathbf{A}) = \sqrt{\sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n} a_{ij}^2} > \epsilon.$$

- Now choose the matrix elements a_{kl} so that we have those with largest value, that is $|a_{kl}| = \max_{i\neq j}|a_{ij}|$.
- Compute thereafter $\tau = (a_{ll}-a_{kk})/2a_{kl}$, $\tan\theta$, $\cos\theta$ and $\sin\theta$.
- Compute thereafter the similarity transformation for this set of values (k,l), obtaining the new matrix $\mathbf{B} = \mathbf{S}(k,l,\theta)^T\mathbf{A}\mathbf{S}(k,l,\theta)$.
- Compute the new norm of the off-diagonal matrix elements and continue till you have satisfied $\mathrm{off}(\mathbf{B}) \leq \epsilon$

The convergence rate of the Jacobi method is however poor, one needs typically $3n^2 - 5n^2$ rotations and each rotation requires $4n$ operations, resulting in a total of $12n^3 - 20n^3$ operations in order to zero out non-diagonal matrix elements. Although the classical Jacobi algorithm performs badly compared with methods based on tridiagonalization, it is easy to parallelize. We discuss how to parallelize this method in the next subsection.

12.4.1 Parallel Jacobi algorithm

In preparation for end fall 2008.

12.5 Diagonalization through the Householder's method for tridiagonalization

In this case the diagonalization is performed in two steps: First, the matrix is transformed into tridiagonal form by the Householder similarity transformation. Secondly, the tridiagonal matrix is then diagonalized. The reason for this two-step process is that diagonalising a tridiagonal matrix is computational much faster then the corresponding diagonalization of a general symmetric matrix. Let us discuss the two steps in more detail.

12.5.1 The Householder's method for tridiagonalization

The first step consists in finding an orthogonal matrix $\mathbf{S}$ which is the product of $(n-2)$ orthogonal matrices

$$\mathbf{S} = \mathbf{S}_1\mathbf{S}_2 \dots \mathbf{S}_{n-2},$$

each of which successively transforms one row and one column of $\mathbf{A}$ into the required tridiagonal form. Only $n-2$ transformations are required, since the last two elements are already in tridiagonal form. In order to determine each $\mathbf{S_i}$ let us see what happens after the first multiplication, namely,

$$\mathbf{S}_1^T\mathbf{A}\mathbf{S}_1 = \begin{pmatrix} a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & a'_{22} & a'_{23} & \dots & \dots & \dots & a'_{2n} \\ 0 & a'_{32} & a'_{33} & \dots & \dots & \dots & a'_{3n} \\ 0 & \dots & \dots & \dots & \dots & \dots & \\ 0 & a'_{n2} & a'_{n3} & \dots & \dots & \dots & a'_{nn} \end{pmatrix}$$

where the primed quantities represent a matrix $\mathbf{A}'$ of dimension $n-1$ which will subsequentely be transformed by $\mathbf{S_2}$. The factor e_1 is a possibly non-vanishing element. The next transformation produced by $\mathbf{S_2}$ has the same effect as $\mathbf{S_1}$ but now on the submatirx $\mathbf{A}'$ only

$$(\mathbf{S}_1\mathbf{S}_2)^T\,\mathbf{A}\mathbf{S}_1\mathbf{S}_2 = \begin{pmatrix} a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & a'_{22} & e_2 & 0 & \dots & \dots & 0 \\ 0 & e_2 & a''_{33} & \dots & \dots & \dots & a''_{3n} \\ 0 & \dots & \dots & \dots & \dots & \dots & \\ 0 & 0 & a''_{n3} & \dots & \dots & \dots & a''_{nn} \end{pmatrix}$$

Note that the effective size of the matrix on which we apply the transformation reduces for every new step. In the previous Jacobi method each similarity transformation is in principle performed on the full size of the original matrix.

After a series of such transformations, we end with a set of diagonal matrix elements

$$a_{11}, a'_{22}, a''_{33} \dots a_{nn}^{n-1},$$

and off-diagonal matrix elements

$$e_1, e_2, e_3, \dots, e_{n-1}.$$

The resulting matrix reads

$$\mathbf{S}^T\mathbf{A}\mathbf{S} = \begin{pmatrix} a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & a'_{22} & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & a''_{33} & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & a_{n-2}^{(n-1)} & e_{n-1} \\ 0 & \dots & \dots & \dots & \dots & e_{n-1} & a_{n-1}^{(n-1)} \end{pmatrix}.$$

It remains to find a recipe for determining the transformation $\mathbf{S}_n$. We illustrate the method for $\mathbf{S}_1$ which we assume takes the form

$$\mathbf{S_1} = \begin{pmatrix} 1 & \mathbf{0^T} \\ \mathbf{0} & \mathbf{P} \end{pmatrix},$$

with $\mathbf{0^T}$ being a zero row vector, $\mathbf{0^T} = \{0, 0, \cdots\}$ of dimension $(n-1)$. The matrix $\mathbf{P}$ is symmetric with dimension $((n-1) \times (n-1))$ satisfying $\mathbf{P}^2 = \mathbf{I}$ and $\mathbf{P}^T = \mathbf{P}$. A possible choice which fullfils the latter two requirements is

$$\mathbf{P} = \mathbf{I} - 2\mathbf{u}\mathbf{u}^T,$$

where $\mathbf{I}$ is the $(n-1)$ unity matrix and $\mathbf{u}$ is an $n-1$ column vector with norm $\mathbf{u}^T\mathbf{u}$(inner product. Note that $\mathbf{u}\mathbf{u}^T$ is an outer product giving a dimension $((n-1) \times (n-1))$. Each matrix element of $\mathbf{P}$ then reads

$$P_{ij} = \delta_{ij} - 2u_i u_j,$$

where i and j range from 1 to $n-1$. Applying the transformation $\mathbf{S}_1$ results in

$$\mathbf{S}_1^T \mathbf{A} \mathbf{S}_1 = \begin{pmatrix} a_{11} & (\mathbf{Pv})^T \\ \mathbf{Pv} & \mathbf{A}' \end{pmatrix},$$

where $\mathbf{v^T} = \{a_{21}, a_{31}, \cdots, a_{n1}\}$ and $\mathbf{P}$ must satisfy $(\mathbf{Pv})^T = \{k, 0, 0, \cdots\}$. Then

$$\mathbf{Pv} = \mathbf{v} - 2\mathbf{u}(\mathbf{u}^T\mathbf{v}) = k\mathbf{e}, \tag{12.6}$$

with $\mathbf{e^T} = \{1, 0, 0, \ldots 0\}$. Solving the latter equation gives us $\mathbf{u}$ and thus the needed transformation $\mathbf{P}$. We do first however need to compute the scalar k by taking the scalar product of the last equation with its transpose and using the fact that $\mathbf{P}^2 = \mathbf{I}$. We get then

$$(\mathbf{Pv})^T\mathbf{Pv} = k^2 = \mathbf{v}^T\mathbf{v} = |v|^2 = \sum_{i=2}^{n} a_{i1}^2,$$

which determines the constant $k = \pm v$. Now we can rewrite Eq. (12.6) as

$$\mathbf{v} - k\mathbf{e} = 2\mathbf{u}(\mathbf{u}^T\mathbf{v}),$$

and taking the scalar product of this equation with itself and obtain

$$2(\mathbf{u}^T\mathbf{v})^2 = (v^2 \pm a_{21}v), \tag{12.7}$$

which finally determines

$$\mathbf{u} = \frac{\mathbf{v} - k\mathbf{e}}{2(\mathbf{u}^T\mathbf{v})}.$$

In solving Eq. (12.7) great care has to be exercised so as to choose those values which make the right-hand largest in order to avoid loss of numerical precision. The above steps are then repeated for every transformations till we have a tridiagonal matrix suitable for obtaining the eigenvalues.

12.5.2 Diagonalization of a tridiagonal matrix

The matrix is now transformed into tridiagonal form and the last step is to transform it into a diagonal matrix giving the eigenvalues on the diagonal.

Before we discuss the algorithms, we note that the eigenvalues of a tridiagonal matrix can be obtained using the characteristic polynomial

$$P(\lambda) = det(\lambda\mathbf{I} - \mathbf{A}) = \prod_{i=1}^{n} (\lambda_i - \lambda),$$

which rewritten in matrix form reads

$$P(\lambda) = \begin{pmatrix} d_1 - \lambda & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & d_2 - \lambda & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & d_3 - \lambda & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & d_{N_{\rm step}-2} - \lambda & e_{N_{\rm step}-1} \\ 0 & \dots & \dots & \dots & \dots & e_{N_{\rm step}-1} & d_{N_{\rm step}-1} - \lambda \end{pmatrix}$$

We can solve this equation in a recursive manner. We let $P_k(\lambda)$ be the value of k subdeterminant of the above matrix of dimension $n \times n$. The polynomial $P_k(\lambda)$ is clearly a polynomial of degree k. Starting with $P_1(\lambda)$ we have $P_1(\lambda) = d_1 - \lambda$. The next polynomial reads $P_2(\lambda) = (d_2 - \lambda)P_1(\lambda) - e_1^2$. By expanding the determinant for $P_k(\lambda)$ in terms of the minors of the nth column we arrive at the recursion relation

$$P_k(\lambda) = (d_k - \lambda)P_{k-1}(\lambda) - e_{k-1}^2 P_{k-2}(\lambda).$$

Together with the starting values $P_1(\lambda)$ and $P_2(\lambda)$ and good root searching methods we arrive at an efficient computational scheme for finding the roots of $P_n(\lambda)$. However, for large matrices this algorithm is rather inefficient and time-consuming.

The programs which performs these transformations are matrix $\mathbf{A} \longrightarrow$ tridiagonal matrix $\longrightarrow$ diagonal matrix

C:	void trd2(double $**$a, int n, double d[], double e[])
	void tqli(double d[], double[], int n, double $**$z)
Fortran:	CALL tred2(a, n, d, e)
	CALL tqli(d, e, n, z)

The last step through the function *tqli()* involves several technical details. Let us describe the basic idea in terms of a four-dimensional example. For more details, see Ref. [27], in particular chapters seven and eight.

The current tridiagonal matrix takes the form

$$\mathbf{A} = \begin{pmatrix} d_1 & e_1 & 0 & 0 \\ e_1 & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{pmatrix}.$$

As a first observation, if any of the elements e_i are zero the matrix can be separated into smaller pieces before diagonalization. Specifically, if $e_1 = 0$ then d_1 is an eigenvalue. Thus, let us introduce a transformation $\mathbf{S_1}$

$$\mathbf{S_1} = \begin{pmatrix} \cos\theta & 0 & 0 & \sin\theta \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -\sin\theta & 0 & 0 & \cos\theta \end{pmatrix}$$

Then the similarity transformation

$$\mathbf{S_1^T A S_1} = \mathbf{A'} = \begin{pmatrix} d'_1 & e'_1 & 0 & 0 \\ e'_1 & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e'3 \\ 0 & 0 & e'_3 & d'_4 \end{pmatrix}$$

produces a matrix where the primed elements in $\mathbf{A}'$ have been changed by the transformation whereas the unprimed elements are unchanged. If we now choose θ to give the element $a_{21}^{'} = e^{'} = 0$ then we have the first eigenvalue $= a_{11}^{'} = d_1^{'}$.

This procedure can be continued on the remaining three-dimensional submatrix for the next eigenvalue. Thus after four transformations we have the wanted diagonal form.

12.6 The QR algorithm for finding eigenvalues

In preparation

12.7 Schrödinger's equation through diagonalization

Instead of solving the Schrödinger equation as a differential equation, we will solve it through diagonalization of a large matrix. However, in both cases we need to deal with a problem with boundary conditions, viz., the wave function goes to zero at the endpoints.

To solve the Schrödinger equation as a matrix diagonalization problem, let us study the radial part of the Schrödinger equation. The radial part of the wave function, $R(r)$, is a solution to

$$-\frac{\hbar^2}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr} - \frac{l(l+1)}{r^2}\right)R(r) + V(r)R(r) = ER(r).$$

Then we substitute $R(r) = (1/r)u(r)$ and obtain

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r) + \left(V(r) + \frac{l(l+1)}{r^2}\frac{\hbar^2}{2m}\right)u(r) = Eu(r).$$

We introduce a dimensionless variable $\rho = (1/\alpha)r$ where α is a constant with dimension length and get

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(r) + \left(V(\rho) + \frac{l(l+1)}{\rho^2}\frac{\hbar^2}{2m\alpha^2}\right)u(\rho) = Eu(\rho).$$

In the example below, we will replace the latter equation with that for the one-dimensional harmonic oscillator. Note however that the procedure which we give below applies equally well to the case of e.g., the hydrogen atom. We replace ρ with x, take away the centrifugal barrier term and set the potential equal to

$$V(x) = \frac{1}{2}kx^2,$$

with k being a constant. In our solution we will use units so that $k = \hbar = m = \alpha = 1$ and the Schrödinger equation for the one-dimensional harmonic oscillator becomes

$$-\frac{d^2}{dx^2}u(x) + x^2u(x) = 2Eu(x).$$

Let us now see how we can rewrite this equation as a matrix eigenvalue problem. First we need to compute the second derivative. We use here the following expression for the second derivative of a function f

$$f'' = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2), \tag{12.8}$$

where h is our step. Next we define minimum and maximum values for the variable x, $R_{\rm min}$ and $R_{\rm max}$, respectively. With a given number of steps, $N_{\rm step}$, we then define the step h as

$$h = \frac{R_{\rm max} - R_{\rm min}}{N_{\rm step}}.$$

If we now define an arbitrary value of x as

$$x_i = R_{\rm min} + ih \qquad i = 1, 2, \dots, N_{\rm step} - 1$$

we can rewrite the Schrödinger equation for x_i as

$$-\frac{u(x_k + h) - 2u(x_k) + u(x_k - h)}{h^2} + x_k^2 u(x_k) = 2Eu(x_k),$$

or in a more compact way

$$-\frac{u_{k+1} - 2u_k + u_{k-1}}{h^2} + x_k^2 u_k = -\frac{u_{k+1} - 2u_k + u_{k-1}}{h^2} + V_k u_k = 2Eu_k,$$

where $u_k = u(x_k)$, $u_{k\pm1} = u(x_k \pm h)$ and $V_k = x_k^2$, the given potential. Let us see how this recipe may lead to a matrix reformulation of the Schrödinger equation. Define first the diagonal matrix element

$$d_k = \frac{2}{h^2} + V_k,$$

and the non-diagonal matrix element

$$e_k = -\frac{1}{h^2}.$$

In this case the non-diagonal matrix elements are given by a mere constant. *All non-diagonal matrix elements are equal.* With these definitions the Schrödinger equation takes the following form

$$d_k u_k + e_{k-1} u_{k-1} + e_{k+1} u_{k+1} = 2Eu_k,$$

where u_k is unknown. Since we have $N_{\rm step} - 1$ values of k we can write the latter equation as a matrix eigenvalue problem

$$\left(\begin{array}{ccccccc} d_1 & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & d_2 & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & d_3 & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & d_{N_{\rm step}-2} & e_{N_{\rm step}-1} \\ 0 & \dots & \dots & \dots & \dots & e_{N_{\rm step}-1} & d_{N_{\rm step}-1} \end{array}\right) \left(\begin{array}{c} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{N_{\rm step}-1} \end{array}\right) = 2E \left(\begin{array}{c} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{N_{\rm step}-1} \end{array}\right) \tag{12.9}$$

or if we wish to be more detailed, we can write the tridiagonal matrix as

$$\left(\begin{array}{ccccccc} \frac{2}{h^2} + V_1 & -\frac{1}{h^2} & 0 & 0 & \dots & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2} + V_2 & -\frac{1}{h^2} & 0 & \dots & 0 & 0 \\ 0 & -\frac{1}{h^2} & \frac{2}{h^2} + V_3 & -\frac{1}{h^2} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \frac{2}{h^2} + V_{N_{\rm step}-2} & -\frac{1}{h^2} \\ 0 & \dots & \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2} + V_{N_{\rm step}-1} \end{array}\right) \tag{12.10}$$

This is a matrix problem with a tridiagonal matrix of dimension $N_{\rm step}-1 \times N_{\rm step}-1$ and will thus yield $N_{\rm step}-1$ eigenvalues. It is important to notice that we do not set up a matrix of dimension $N_{\rm step} \times N_{\rm step}$ since we can fix the value of the wave function at $k = N_{\rm step}$. Similarly, we know the wave function at the other end point, that is for x_0.

The above equation represents an alternative to the numerical solution of the differential equation for the Schrödinger equation discussed in chapter 14.

The eigenvalues of the harmonic oscillator in one dimension are well known. In our case, with all constants set equal to 1, we have

$$E_n = n + \frac{1}{2},$$

with the ground state being $E_0 = 1/2$. Note however that we have rewritten the Schrödinger equation so that a constant 2 stands in front of the energy. Our program will then yield twice the value, that is we will obtain the eigenvalues $1, 3, 5, 7.. \dots$.

In the next subsection we will try to delineate how to solve the above equation. A program listing is also included.

12.7.1 Numerical solution of the Schrödinger equation by diagonalization

The algorithm for solving Eq. (12.9) may take the following form

- Define values for $N_{\rm step}$, $R_{\rm min}$ and $R_{\rm max}$. These values define in turn the step size h. Typical values for $R_{\rm max}$ and $R_{\rm min}$ could be 10 and -10 respectively for the lowest-lying states. The number of mesh points $N_{\rm step}$ could be in the range 100 to some thousands. You can check the stability of the results as functions of $N_{\rm step}-1$ and $R_{\rm max}$ and $R_{\rm min}$ against the exact solutions.

- Construct then two one-dimensional arrays which contain all values of x_k and the potential V_k. For the latter it can be convenient to write a small function which sets up the potential as function of x_k. For the three-dimensional case you may also need to include the centrifugal potential. The dimension of these two arrays should go from 0 up to $N_{\rm step}$.

- Construct thereafter the one-dimensional vectors d and e, where d stands for the diagonal matrix elements and e the non-diagonal ones. Note that the dimension of these two arrays runs from 1 up to $N_{\rm step}-1$, since we know the wave function u at both ends of the chosen grid.

- We are now ready to obtain the eigenvalues by calling the function *tqli* which can be found on the web page of the course. Calling *tqli*, you have to transfer the matrices d and e, their dimension $n = N_{\rm step}-1$ and a matrix z of dimension $N_{\rm step}-1 \times N_{\rm step}-1$ which returns the eigenfunctions. On return, the array d contains the eigenvalues. If z is given as the unity matrix on input, it returns the eigenvectors. For a given eigenvalue k, the eigenvector is given by the column k in z, that is z[][k] in C, or z(:,k) in Fortran.

- TQLI does however not return an ordered sequence of eigenvalues. You may then need to sort them as e.g., an ascending series of numbers. The program we provide includes a sorting function as well.

- Finally, you may perhaps need to plot the eigenfunctions as well, or calculate some other expectation values. Or, you would like to compare the eigenfunctions with the analytical answers for the harmonic oscillator or the hydrogen atom. We provide a function *plot* which has as input one eigenvalue chosen from the output of *tqli*. This function gives you a normalized wave function u where the norm is calculated as

$$\int_{R_{\mathrm{min}}}^{R_{\mathrm{max}}} |u(x)|^2\, dx \to h \sum_{i=0}^{N_{\mathrm{step}}} u_i^2 = 1,$$

and we have used the trapezoidal rule for integration discussed in chapter 7.

12.7.2 Program example and results for the one-dimensional harmonic oscillator

We present here a program example which encodes the above algorithm.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter12/cpp/program1.cpp

```
/*
  Solves the one-particle Schrodinger equation
  for a potential specified in function
  potential(). This example is for the harmonic oscillator
*/
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace  std;
// output file as global variable
ofstream ofile;

// function declarations

void initialise(double&, double&, int&, int&) ;
double potential(double);
int comp(const double *, const double *);
void output(double, double, int, double *);

int main(int argc, char* argv[])
{
  int        i, j, max_step, orb_l;
  double     r_min, r_max, step, const_1, const_2, orb_factor,
             *e, *d, *w, *r, **z;
  char *outfilename;
  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  //   Read in data
  initialise(r_min, r_max, orb_l, max_step);
```

```
  // initialise constants
  step      = (r_max - r_min) / max_step;
  const_2  = -1.0 / (step * step);
  const_1  =  - 2.0 * const_2;
  orb_factor = orb_l * (orb_l + 1);

  // local memory for r and the potential w[r]
  r = new double[max_step + 1];
  w = new double[max_step + 1];
  for(i = 0; i <= max_step; i++) {
    r[i] = r_min + i * step;
    w[i] = potential(r[i]) + orb_factor / (r[i] * r[i]);
  }
  // local memory for the diagonalization process
  d = new double[max_step];     // diagonal elements
  e = new double[max_step];     // tridiagonal off-diagonal elements
  z = (double **) matrix(max_step, max_step, sizeof(double));
  for(i = 0; i < max_step; i++) {
    d[i]    = const_1 + w[i + 1];
    e[i]    = const_2;
    z[i][i] = 1.0;
    for(j = i + 1; j < max_step; j++)  {
      z[i][j] = 0.0;
    }
  }
  // diagonalize and obtain eigenvalues
  tqli(d, e, max_step - 1, z);
  // Sort eigenvalues as an ascending series
  qsort(d,(UL) max_step - 1,sizeof(double),
        (int(*)(const void *,const void *))comp);
  // send results to ouput file
  output(r_min , r_max, max_step, d);
  delete [] r; delete [] w; delete [] e; delete [] d;
  free_matrix((void **) z); // free memory
  ofile.close();  // close output file
  return 0;
} // End: function main()

/*
  The function potential()
  calculates and return the value of the
  potential for a given argument x.
  The potential here is for the 1-dim harmonic oscillator
*/

double potential(double x)
{
   return x*x;

} // End: function potential()

/*
  The function   int comp()
```

```
  is a utility function for the library function qsort()
  to sort double numbers after increasing values.
*/

int comp(const double *val_1, const double *val_2)
{
  if((*val_1) <= (*val_2))       return -1;
  else  if((*val_1) > (*val_2))  return +1;
  else                     return  0;
} // End: function comp()

// read in min  and max radius, number of mesh points and l
void initialise(double& r_min, double& r_max, int& orb_l, int& max_step)
{
  cout << "Min vakues of R = ";
  cin >> r_min;
  cout << "Max value of R = ";
  cin >> r_max;
  cout << "Orbital momentum = ";
  cin >> orb_l;
  cout << "Number of steps = ";
  cin >> max_step;
}  // end of function initialise
// output of results
void output(double r_min , double r_max, int max_step, double *d)
{
  int i;
  ofile << "RESULTS:" << endl;
  ofile << setiosflags(ios::showpoint | ios::uppercase);
  ofile <<"R_min = " << setw(15) << setprecision(8) << r_min << endl;
  ofile <<"R_max = " << setw(15) << setprecision(8) << r_max << endl;
  ofile <<"Number of steps = " << setw(15) << max_step << endl;
  ofile << "Five lowest eigenvalues:" << endl;
  for(i = 0; i < 5; i++) {
    ofile << setw(15) << setprecision(8) << d[i] << endl;
  }
}  // end of function output
```

There are several features to be noted in this program.

The main program calls the function *initialise*, which reads in the minimum and maximum values of r, the number of steps and the orbital angular momentum l. Thereafter we allocate place for the vectors containing r and the potential, given by the variables $r[i]$ and $w[i]$, respectively. We also set up the vectors $d[i]$ and $e[i]$ containing the diagonal and non-diagonal matrix elements. Calling the function $tqli$ we obtain in turn the unsorted eigenvalues. The latter are sorted by the intrinsic C-function $qsort$.

The calculaton of the wave function for the lowest eigenvalue is done in the function $plot$, while all output of the calculations is directed to the fuction $output$.

The included table exhibits the precision achieved as function of the number of mesh points N. The exact values are $1, 3, 5, 7, 9$.

The agreement with the exact solution improves with increasing numbers of mesh points. However, the agreement for the excited states is by no means impressive. Moreover, as the dimensionality increases, the time consumption increases dramatically. Matrix diagonalization scales typically as $\approx N^3$.

Table 12.1: Five lowest eigenvalues as functions of the number of mesh points N with $r_{\min} = -10$ and $r_{\max} = 10$.

N	E_0	E_1	E_2	E_3	E_4
50	9.898985E-01	2.949052E+00	4.866223E+00	6.739916E+00	8.568442E+00
100	9.974893E-01	2.987442E+00	4.967277E+00	6.936913E+00	8.896282E+00
200	9.993715E-01	2.996864E+00	4.991877E+00	6.984335E+00	8.974301E+00
400	9.998464E-01	2.999219E+00	4.997976E+00	6.996094E+00	8.993599E+00
1000	1.000053E+00	2.999917E+00	4.999723E+00	6.999353E+00	8.999016E+00

In addition, there is a maximum size of a matrix which can be stored in RAM.

The obvious question which then arises is whether this scheme is nothing but a mere example of matrix diagonalization, with few practical applications of interest. In chapter 6, where we deal with interpolation and extrapolation, we discussed also a called Richardson's deferred extrapolation scheme. Applied to this particualr case, the philosophy of this scheme would be to diagonalize the above matrix for a set of values of N and thereby the step length h. Thereafter, an extrapolation is made to $h \to 0$. The obtained eigenvalues agree then with a remarkable precision with the exact solution. The algorithm is then as follows

- Perform a series of diagonalizations of the matrix in Eq. (12.10) for different values of the step size h. We obtain then a series of eigenvalues $E(h/2^k)$ with $k = 0, 1, 2, \dots$. That will give us an array of 'x-values' $h, h/2, h/4, \dots$ and an array of 'y-values' $E(h), E(h/2), E(h/4), \dots$. Note that you will have such a set for each eigenvalue.
- Use these values to perform an extrapolation calling e.g., the function POLINT with the point where we wish to extrapolate to given by $h = 0$.
- End the iteration over k when the error returned by POLINT is smaller than a fixed test.

The results for the 10 lowest-lying eigenstates for the one-dimensional harmonic oscillator are listed below after just 3 iterations, i.e., the step size has been reduced to $h/8$ only. The exact results are $1, 3, 5, \dots, 19$ and we see that the agreement is just excellent for the extrapolated results. The results after diagonalization differ already at the fourth-fifth digit.

Parts of a Fortran program which includes Richardson's extrapolation scheme is included here. It performs five diagonalizations and establishes results for various step lengths and interpolates using the function POLINT.

```
!   start loop over interpolations, here we set max interpolations to 5
      DO interpol=1, 5
         IF ( interpol == 1) THEN
            max_step=start_step
         ELSE
            max_step=(interpol -1)*2*start_step
         ENDIF
```

Table 12.2: Result for numerically calculated eigenvalues of the one-dimensional harmonic oscillator after three iterations starting with a matrix of size 100×100 and ending with a matrix of dimension 800×800. These four values are then used to extrapolate the 10 lowest-lying eigenvalues to $h = 0$.. The values of x span from -10 to 10, that means that the starting step was $h = 20/100 = 0.2$. We list here only the results after three iterations. The error test was set equal 10^{-6}.

Extrapolation	Diagonalization	Error
0.100000D+01	0.999931D+00	0.206825D-10
0.300000D+01	0.299965D+01	0.312617D-09
0.500000D+01	0.499910D+01	0.174602D-08
0.700000D+01	0.699826D+01	0.605671D-08
0.900000D+01	0.899715D+01	0.159170D-07
0.110000D+02	0.109958D+02	0.349902D-07
0.130000D+02	0.129941D+02	0.679884D-07
0.150000D+02	0.149921D+02	0.120735D-06
0.170000D+02	0.169899D+02	0.200229D-06
0.190000D+02	0.189874D+02	0.314718D-06

```
          n=max_step-1
          ALLOCATE ( e(n) , d(n) )
          ALLOCATE ( w(0:max_step), r(0:max_step))
          d=0. ; e =0.
!   define the step size
          step=(rmax-rmin)/FLOAT(max_step)
          hh(interpol)=step*step
!   define constants for the matrix to be diagonalized
          const1=2./(step*step)
          const2=-1./(step*step)
!      set up r, the distance from the nucleus and the function w for energy
   =0
!      w corresponds then to the potential
!      values at
          DO i=0, max_step
             r(i) = rmin+i*step
             w(i) = potential(r(i))
          ENDDO
!      setup the diagonal d and the non-diagonal part e  of
!      the tridiagonal matrix matrix to be diagonalized
          d(1:n)=const1+w(1:n)  ;  e(1:n)=const2
!   allocate space for eigenvector info
          ALLOCATE ( z(n,n) )
!   obtain the eigenvalues
          CALL tqli(d,e,n,z)
!   sort eigenvalues as an ascending series
          CALL eigenvalue_sort(d,n)
          DEALLOCATE (z)
          err1=0.
!   the interpolation part starts here
```

```
      DO l=1,20
         err2=0.
         value(interpol,l)=d(l)
         inp=d(l)
         IF ( interpol > 1 ) THEN
            CALL polint(hh,value(:,l),interpol,0.d0 ,inp,err2)
            err1=MAX(err1,err2)
            WRITE(6,'(D12.6,2X,D12.6,2X,D12.6)') inp, d(l), err1
         ELSE
            WRITE(6,'(D12.6,2X,D12.6,2X,D12.6)') d(l), d(l), err1
         ENDIF
      ENDDO
      DEALLOCATE ( w, r, d, e)
   ENDDO
```

12.8 Discussion of BLAS and LAPACK functionalities

In preparation for fall 2008.

12.9 Exercises and projects

Project 12.1: Bound states in momentum space

In this problem we will solve the Schrödinger equation in momentum space for the deuteron. The deuteron has only one bound state at an energy of -2.223 MeV. The ground state is given by the quantum numbers $l = 0$, $S = 1$ and $J = 1$, with l, S, and J the relative orbital momentum, the total spin and the total angular momentum, respectively. These quantum numbers are the sum of the single-particle quantum numbers. The deuteron consists of a proton and neutron, with mass (average) of 938 MeV. The electron is not included in the solution of the Schrödinger equation since its mass is much smaller than those of the proton and the neutron. We can neglect it here. This means that e.g., the total spin S is the sum of the spin of the neutron and the proton. The above three quantum numbers can be summarized in the spectroscopic notation $^{2S+1}l_J =^3 S_1$, where S represents $l = 0$ here. It is a spin triplet state. The spin wave function is thus symmetric. This also applies to the spatial part, since $l = 0$. To obtain a totally anti-symmetric wave function we need to introduce another quantum number, namely isospin. The deuteron has isospin $T = 0$, which gives a final wave function which is anti-symmetric.

We are going to use a simplified model for the interaction between the neutron and the proton. We will assume that it goes like

$$V(r) = V_0 \frac{\exp(-\mu r)}{r}, \tag{12.11}$$

where μ has units m^{-1} and serves to screen the potential for large values of r. The variable r is the distance between the proton and the neutron. It is the relative coordinate, the centre of mass is not needed in this problem. The nucleon-nucleon interaction has a finite and small range, typically of some few fm[1]. We will in this exercise set $\mu = 0.7$ fm^{-1}. It is then proportional to the mass of the pion. The pion is the lightest meson, and sets therefore the range of the nucleon-nucleon interaction. For low-energy problems

[1] 1 fm = 10^{-15} m.

we can describe the nucleon-nucleon interaction through meson-exchange models, and the pion is the lightest known meson, with mass of approximately 138 MeV.

Since we are going to solve the Schrödinger equation in momentum, we need the Fourier transform of $V(r)$. In a partial wave basis for $l = 0$ it becomes

$$V(k', k) = \frac{V_0}{4k'k} ln\left(\frac{(k'+k)^2 + \mu^2}{(k'-k)^2 + \mu^2}\right), \tag{12.12}$$

where k' and k are the relative momenta for the proton and neutron system.

For relative coordinates, the Schrödinger equation in momentum space becomes

$$\frac{k^2}{m}\psi(k) + \frac{2}{\pi}\int_0^\infty dpp^2 V(k,p)\psi(p) = E\psi(k). \tag{12.13}$$

Here we have used units $\hbar = c = 1$. This means that k has dimension energy. This is the equation we are going to solve, with eigenvalue E and eigenfunction $\psi(k)$. The approach to solve this equations goes then as follows.

First we need to evaluate the integral over p using e.g., gaussian quadrature. This means that we rewrite an integral like

$$\int_a^b f(x)dx \approx \sum_{i=1}^N \omega_i f(x_i),$$

where we have fixed N lattice points through the corresponding weights ω_i and points x_i. The integral in Eq. (12.13) is rewritten as

$$\frac{2}{\pi}\int_0^\infty dpp^2 V(k,p)\psi(p) \approx \frac{2}{\pi}\sum_{i=1}^N \omega_i p_i^2 V(k,p_i)\psi(p_i). \tag{12.14}$$

We can then rewrite the Schrödinger equation as

$$\frac{k^2}{m}\psi(k) + \frac{2}{\pi}\sum_{j=1}^N \omega_j p_j^2 V(k,p_j)\psi(p_j) = E\psi(k). \tag{12.15}$$

Using the same mesh points for k as we did for p in the integral evaluation, we get

$$\frac{p_i^2}{m}\psi(p_i) + \frac{2}{\pi}\sum_{j=1}^N \omega_j p_j^2 V(p_i,p_j)\psi(p_j) = E\psi(p_i), \tag{12.16}$$

with $i, j = 1, 2, \ldots, N$. This is a matrix eigenvalue equation and if we define an $N \times N$ matrix $\mathbf{H}$ to be

$$H_{ij} = \frac{p_i^2}{m}\delta_{ij} + \frac{2}{\pi}\omega_j p_j^2 V(p_i,p_j), \tag{12.17}$$

where δ_{ij} is the Kronecker delta, and an $N \times 1$ vector

$$\Psi = \begin{pmatrix} \psi(p_1) \\ \psi(p_2) \\ \ldots \\ \ldots \\ \psi(p_N) \end{pmatrix}, \tag{12.18}$$

we have the eigenvalue problem

$$\mathbf{H}\Psi = E\Psi. \tag{12.19}$$

The algorithm for solving the last equation may take the following form

- Fix the number of mesh points N.
- Use the function $gauleg$ in the program library to set up the weights ω_i and the points p_i. Before you go on you need to recall that $gauleg$ uses the Legendre polynomials to fix the mesh points and weights. This means that the integral is for the interval [-1,1]. Your integral is for the interval [0,∞]. You will need to map the weights from $gauleg$ to your interval. To do this, call first $gauleg$, with $a = -1$, $b = 1$. It returns the mesh points and weights. You then map these points over to the limits in your integral. You can then use the following mapping

$$p_i = \text{const} \times tan\left\{\frac{\pi}{4}(1 + x_i)\right\},$$

  and

$$\omega_i = \text{const}\frac{\pi}{4}\frac{w_i}{cos^2\left(\frac{\pi}{4}(1 + x_i)\right)}.$$

  $const$ is a constant which we discuss below.
- Construct thereafter the matrix $\mathbf{H}$ with

$$V(p_i, p_j) = \frac{V_0}{4p_ip_j}ln\left(\frac{(p_j + p_i)^2 + \mu^2}{(p_j - p_i)^2 + \mu^2}\right).$$

- We are now ready to obtain the eigenvalues. We need first to rewrite the matrix $\mathbf{H}$ in tridiagonal form. Do this by calling the library function *tred2*. This function returns the vector d with the diagonal matrix elements of the tridiagonal matrix while e are the non-diagonal ones. To obtain the eigenvalues we call the function $tqli$. On return, the array d contains the eigenvalues. If z is given as the unity matrix on input, it returns the eigenvectors. For a given eigenvalue k, the eigenvector is given by the column k in z, that is z[][k] in C, or z(:,k) in Fortran.

The problem to solve

1. Before you write the main program for the above algorithm make a dimensional analysis of Eq. (12.13)! You can choose units so that p_i and ω_i are in fm-1. This is the standard unit for the wave vector. Recall then to insert $\hbar c$ in the appropriate places. For this case you can set the value of $const = 1$. You could also choose units so that the units of p_i and ω_i are in MeV. (we have previously used so-called natural units $\hbar = c = 1$). You will then need to multiply μ with $\hbar c = 197$ MeVfm to obtain the same units in the expression for the potential. Why? Show that $V(p_i, p_j)$ must have units MeV^{-2}. What is the unit of V_0? If you choose these units you should also multiply the mesh points and the weights with $\hbar c = 197$. That means, set the constant $const = 197$.
2. Write your own program so that you can solve the Schrödinger equation in momentum space.
3. Adjust the value of V_0 so that you get close to the experimental value of the binding energy of the deuteron, -2.223 MeV. Which sign should V_0 have?
4. Try increasing the number of mesh points in steps of 8, for example 16, 24, etc and see how the energy changes. Your program returns equally many eigenvalues as mesh points N. Only the true ground state will be at negative energy.

Chapter 13

Differential equations

> If God has made the world a perfect mechanism, he has at least conceded so much to our imperfect intellect that in order to predict little parts of it, we need not solve innumerable differential equations, but can use dice with fair success. *Max Born, quoted in H. R. Pagels, The Cosmic Code [67]*

13.1 Introduction

We may trace the origin of differential equations back to Newton in 1687[1] and his treatise on the gravitational force and what is known to us as Newton's second law in dynamics.

Needless to say, differential equations pervade the sciences and are to us the tools by which we attempt to express in a concise mathematical language the laws of motion of nature. We uncover these laws via the dialectics between theories, simulations and experiments, and we use them on a daily basis which spans from applications in engineering or financial engineering to basic research in for example biology, chemistry, mechanics, physics, ecological models or medicine.

We have already met the differential equation for radioactive decay in nuclear physics. Other famous differential equations are Newton's law of cooling in thermodynamics. the wave equation, Maxwell's equations in electromagnetism, the heat equation in thermodynamic, Laplace's equation and Poisson's equation, Einstein's field equation in general relativity, Schrödinger equation in quantum mechanics, the Navier-Stokes equations in fluid dynamics, the Lotka-Volterra equation in population dynamics, the Cauchy-Riemann equations in complex analysis and the Black-Scholes equation in finance, just to mention a few. Excellent texts on differential equations and computations are the texts of Eriksson, Estep, Hansbo and Johnson [68], Butcher [69] and Hairer, Nørsett and Wanner [70].

There are five main types of differential equations,

- ordinary differential equations (ODEs), discussed in this chapter for initial value problems only. They contain functions of one independent variable, and derivatives in that variable. The next chapter deals with ODEs and boundary value problems.
- Partial differential equations with functions of multiple independent variables and their partial derivatives, covered in chapter 15.

[1] Newton had most of the relations for his laws ready 22 years earlier, when according to legend he was contemplating falling apples. However, it took more than two decades before he published his theories, chiefly because he was lacking an essential mathematical tool, differential calculus.

- So-called delay differential equations that involve functions of one dependent variable, derivatives in that variable, and depend on previous states of the dependent variables.
- Stochastic differential equations (SDEs) are differential equations in which one or more of the terms is a stochastic process, thus resulting in a solution which is itself a stochastic process.
- Finally we have so-called differential algebraic equations (DAEs). These are differential equation comprising differential and algebraic terms, given in implicit form.

In this chapter we restrict the attention to ordinary differential equations. We focus on initial value problems and present some of the more commonly used methods for solving such problems numerically. The physical systems which are discussed range from the classical pendulum with non-linear terms to the physics of a neutron star or a white dwarf.

13.2 Ordinary differential equations

In this section we will mainly deal with ordinary differential equations and numerical methods suitable for dealing with them. However, before we proceed, a brief remainder on differential equations may be appropriate.

- The order of the ODE refers to the order of the derivative on the left-hand side in the equation

  $$\frac{dy}{dt} = f(t, y). \tag{13.1}$$

  This equation is of first order and f is an arbitrary function. A second-order equation goes typically like

  $$\frac{d^2y}{dt^2} = f(t, \frac{dy}{dt}, y). \tag{13.2}$$

  A well-known second-order equation is Newton's second law

  $$m\frac{d^2x}{dt^2} = -kx, \tag{13.3}$$

  where k is the force constant. ODE depend only on one variable, whereas
- partial differential equations like the time-dependent Schrödinger equation

  $$i\hbar\frac{\partial\psi(\mathbf{x}, t)}{\partial t} = \frac{\hbar^2}{2m}\left(\frac{\partial^2\psi(\mathbf{r}, t)}{\partial x^2} + \frac{\partial^2\psi(\mathbf{r}, t)}{\partial y^2} + \frac{\partial^2\psi(\mathbf{r}, t)}{\partial z^2}\right) + V(\mathbf{x})\psi(\mathbf{x}, t), \tag{13.4}$$

  may depend on several variables. In certain cases, like the above equation, the wave function can be factorized in functions of the separate variables, so that the Schrödinger equation can be rewritten in terms of sets of ordinary differential equations.
- We distinguish also between linear and non-linear differential equation where e.g.,

  $$\frac{dy}{dt} = g^3(t)y(t), \tag{13.5}$$

  is an example of a linear equation, while

  $$\frac{dy}{dt} = g^3(t)y(t) - g(t)y^2(t), \tag{13.6}$$

is a non-linear ODE. Another concept which dictates the numerical method chosen for solving an ODE, is that of initial and boundary conditions. To give an example, in our study of neutron stars below, we will need to solve two coupled first-order differential equations, one for the total mass m and one for the pressure P as functions of ρ

$$\frac{dm}{dr} = 4\pi r^2 \rho(r)/c^2,$$

and

$$\frac{dP}{dr} = -\frac{Gm(r)}{r^2}\rho(r)/c^2.$$

where ρ is the mass-energy density. The initial conditions are dictated by the mass being zero at the center of the star, i.e., when $r = 0$, yielding $m(r = 0) = 0$. The other condition is that the pressure vanishes at the surface of the star. This means that at the point where we have $P = 0$ in the solution of the integral equations, we have the total radius R of the star and the total mass $m(r = R)$. These two conditions dictate the solution of the equations. Since the differential equations are solved by stepping the radius from $r = 0$ to $r = R$, so-called one-step methods (see the next section) or Runge-Kutta methods may yield stable solutions.

In the solution of the Schrödinger equation for a particle in a potential, we may need to apply boundary conditions as well, such as demanding continuity of the wave function and its derivative.

– In many cases it is possible to rewrite a second-order differential equation in terms of two first-order differential equations. Consider again the case of Newton's second law in Eq. (13.3). If we define the position $x(t) = y^{(1)}(t)$ and the velocity $v(t) = y^{(2)}(t)$ as its derivative

$$\frac{dy^{(1)}(t)}{dt} = \frac{dx(t)}{dt} = y^{(2)}(t), \tag{13.7}$$

we can rewrite Newton's second law as two coupled first-order differential equations

$$m\frac{dy^{(2)}(t)}{dt} = -kx(t) = -ky^{(1)}(t), \tag{13.8}$$

and

$$\frac{dy^{(1)}(t)}{dt} = y^{(2)}(t). \tag{13.9}$$

13.3 Finite difference methods

These methods fall under the general class of one-step methods. The algoritm is rather simple. Suppose we have an initial value for the function $y(t)$ given by

$$y_0 = y(t = t_0). \tag{13.10}$$

We are interested in solving a differential equation in a region in space [a,b]. We define a step h by splitting the interval in N sub intervals, so that we have

$$h = \frac{b-a}{N}. \tag{13.11}$$

With this step and the derivative of y we can construct the next value of the function y at

$$y_1 = y(t_1 = t_0 + h), \tag{13.12}$$

and so forth. If the function is rather well-behaved in the domain [a,b], we can use a fixed step size. If not, adaptive steps may be needed. Here we concentrate on fixed-step methods only. Let us try to generalize the above procedure by writing the step y_{i+1} in terms of the previous step y_i

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h\Delta(t_i, y_i(t_i)) + O(h^{p+1}), \tag{13.13}$$

where $O(h^{p+1})$ represents the truncation error. To determine Δ, we Taylor expand our function y

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}), \tag{13.14}$$

where we will associate the derivatives in the parenthesis with

$$\Delta(t_i, y_i(t_i)) = (y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}). \tag{13.15}$$

We define

$$y'(t_i) = f(t_i, y_i) \tag{13.16}$$

and if we truncate Δ at the first derivative, we have

$$y_{i+1} = y(t_i) + hf(t_i, y_i) + O(h^2), \tag{13.17}$$

which when complemented with $t_{i+1} = t_i + h$ forms the algorithm for the well-known Euler method. Note that at every step we make an approximation error of the order of $O(h^2)$, however the total error is the sum over all steps $N = (b-a)/h$, yielding thus a global error which goes like $NO(h^2) \approx O(h)$. To make Euler's method more precise we can obviously decrease h (increase N). However, if we are computing the derivative f numerically by e.g., the two-steps formula

$$f'_{2c}(x) = \frac{f(x+h) - f(x)}{h} + O(h),$$

we can enter into roundoff error problems when we subtract two almost equal numbers $f(x+h) - f(x) \approx 0$. Euler's method is not recommended for precision calculation, although it is handy to use in order to get a first view on how a solution may look like. As an example, consider Newton's equation rewritten in Eqs. (13.8) and (13.9). We define $y_0 = y^{(1)}(t = 0)$ an $v_0 = y^{(2)}(t = 0)$. The first steps in Newton's equations are then

$$y_1^{(1)} = y_0 + hv_0 + O(h^2) \tag{13.18}$$

and

$$y_1^{(2)} = v_0 - hy_0k/m + O(h^2). \tag{13.19}$$

The Euler method is asymmetric in time, since it uses information about the derivative at the beginning of the time interval. This means that we evaluate the position at $y_1^{(1)}$ using the velocity at $y_0^{(2)} = v_0$. A simple variation is to determine $y_{n+1}^{(1)}$ using the velocity at $y_{n+1}^{(2)}$, that is (in a slightly more generalized form)

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1}^{(2)} + O(h^2) \tag{13.20}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2). \tag{13.21}$$

The acceleration a_n is a function of $a_n(y_n^{(1)}, y_n^{(2)}, t)$ and needs to be evaluated as well. This is the Euler-Cromer method.

Let us then include the second derivative in our Taylor expansion. We have then

$$\Delta(t_i, y_i(t_i)) = f(t_i) + \frac{h}{2}\frac{df(t_i, y_i)}{dt} + O(h^3). \tag{13.22}$$

The second derivative can be rewritten as

$$y'' = f' = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}f \tag{13.23}$$

and we can rewrite Eq. (13.14) as

$$y_{i+1} = y(t = t_i + h) = y(t_i) + hf(t_i) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}f\right) + O(h^3), \tag{13.24}$$

which has a local approximation error $O(h^3)$ and a global error $O(h^2)$. These approximations can be generalized by using the derivative f to arbitrary order so that we have

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(f(t_i, y_i) + \dots f^{(p-1)}(t_i, y_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}). \tag{13.25}$$

These methods, based on higher-order derivatives, are in general not used in numerical computation, since they rely on evaluating derivatives several times. Unless one has analytical expressions for these, the risk of roundoff errors is large.

13.3.1 Improvements to Euler's algorithm, higher-order methods

The most obvious improvements to Euler's and Euler-Cromer's algorithms, avoiding in addition the need for computing a second derivative, is the so-called midpoint method. We have then

$$y_{n+1}^{(1)} = y_n^{(1)} + \frac{h}{2}\left(y_{n+1}^{(2)} + y_n^{(2)}\right) + O(h^2) \tag{13.26}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2), \tag{13.27}$$

yielding

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_n^{(2)} + \frac{h^2}{2}a_n + O(h^3) \tag{13.28}$$

implying that the local truncation error in the position is now $O(h^3)$, whereas Euler's or Euler-Cromer's methods have a local error of $O(h^2)$. Thus, the midpoint method yields a global error with second-order accuracy for the position and first-order accuracy for the velocity. However, although these methods yield exact results for constant accelerations, the error increases in general with each time step.

One method that avoids this is the so-called half-step method. Here we define

$$y_{n+1/2}^{(2)} = y_{n-1/2}^{(2)} + ha_n + O(h^2), \tag{13.29}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{13.30}$$

Note that this method needs the calculation of $y_{1/2}^{(2)}$. This is done using e.g., Euler's method

$$y_{1/2}^{(2)} = y_0^{(2)} + \frac{h}{2}a_0 + O(h^2). \tag{13.31}$$

As this method is numerically stable, it is often used instead of Euler's method. Another method which one may encounter is the Euler-Richardson method with

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_{n+1/2} + O(h^2), \tag{13.32}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{13.33}$$

13.3.2 Predictor-Corrector methods

Consider again the first-order differential equation

$$\frac{dy}{dt} = f(t, y),$$

which solved with Euler's algorithm results in the following algorithm

$$y_{i+1} \approx y(t_i) + hf(t_i, y_i)$$

with $t_{i+1} = t_i + h$. This means geometrically that we compute the slope at y_i and use it to predict y_{i+1} at a later time t_{i+1}. We introduce $k_1 = f(t_i, y_i)$ and rewrite our prediction for y_{i+1} as

$$y_{i+1} \approx y(t_i) + hk_1.$$

We can then use the prediction y_{i+1} to compute a new slope at t_{i+1} by defining $k_2 = f(t_{i+1}, y_{i+1})$. We define the new value of y_{i+1} by taking the average of the two slopes, resulting in

$$y_{i+1} \approx y(t_i) + \frac{h}{2}(k_1 + k_2).$$

The algorithm is very simple,namely

1. Compute the slope at t_i, that is define the quantity $k_1 = f(t_i, y_i)$.
2. Make a predicition for the solution by computing $y_{i+1} \approx y(t_i) + hk_1$ by Euler's method.
3. Use the predicition y_{i+1} to compute a new slope at t_{i+1} defining the quantity $k_2 = f(t_{i+1}, y_{i+1})$.
4. Correct the value of y_{i+1} by taking the average of the two slopes yielding $y_{i+1} \approx y(t_i) + \frac{h}{2}(k_1 + k_2)$.

It can be shown [30] that this procedure results in a mathematical truncation which goes like $O(h^2)$, to be contrasted with Euler's method which runs as $O(h)$. One additional function evaluation yields a better error estimate.

This simple algorithm conveys the philosophy of a large class of methods called predictor-corrector methods, see chapter 15 of Ref. [36] for additional algorithms. A simple extension is obviously to use Simpson's method to approximate the integral

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt,$$

when we solve the differential equation by successive integrations. The next section deals with a particular class of efficient methods for solving ordinary differential equations, namely various Runge-Kutta methods.

13.4 More on finite difference methods, Runge-Kutta methods

Runge-Kutta (RK) methods are based on Taylor expansion formulae, but yield in general better algorithms for solutions of an ODE. The basic philosophy is that it provides an intermediate step in the computation of y_{i+1}.

To see this, consider first the following definitions

$$\frac{dy}{dt} = f(t, y), \tag{13.34}$$

and

$$y(t) = \int f(t, y)dt, \tag{13.35}$$

and

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt. \tag{13.36}$$

To demonstrate the philosophy behind RK methods, let us consider the second-order RK method, RK2. The first approximation consists in Taylor expanding $f(t, y)$ around the center of the integration interval t_i to t_{i+1}, i.e., at $t_i + h/2$, h being the step. Using the midpoint formula for an integral, defining $y(t_i + h/2) = y_{i+1/2}$ and $t_i + h/2 = t_{i+1/2}$, we obtain

$$\int_{t_i}^{t_{i+1}} f(t, y)dt \approx hf(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{13.37}$$

This means in turn that we have

$$y_{i+1} = y_i + hf(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{13.38}$$

However, we do not know the value of $y_{i+1/2}$. Here comes thus the next approximation, namely, we use Euler's method to approximate $y_{i+1/2}$. We have then

$$y_{(i+1/2)} = y_i + \frac{h}{2}\frac{dy}{dt} = y(t_i) + \frac{h}{2}f(t_i, y_i). \tag{13.39}$$

This means that we can define the following algorithm for the second-order Runge-Kutta method, RK2.

$$k_1 = hf(t_i, y_i), \tag{13.40}$$

$$k_2 = hf(t_{i+1/2}, y_i + k_1/2), \tag{13.41}$$

with the final value

$$y_{i+i} \approx y_i + k_2 + O(h^3). \tag{13.42}$$

The difference between the previous one-step methods is that we now need an intermediate step in our evaluation, namely $t_i + h/2 = t_{(i+1/2)}$ where we evaluate the derivative f. This involves more operations, but the gain is a better stability in the solution. The fourth-order Runge-Kutta, RK4, which we will employ in the solution of various differential equations below, is easily derived. The steps are as follows. We start again with the equation

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y) dt,$$

but instead of approximating the integral with the midpoint rule, we use now Simpsons' rule at $t_i + h/2$, h being the step. Using Simpson's formula for an integral, defining $y(t_i + h/2) = y_{i+1/2}$ and $t_i + h/2 = t_{i+1/2}$, we obtain

$$\int_{t_i}^{t_{i+1}} f(t, y) dt \approx \frac{h}{6}\left[f(t_i, y_i) + 4f(t_{i+1/2}, y_{i+1/2}) + f(t_{i+1}, y_{i+1})\right] + O(h^5). \tag{13.43}$$

This means in turn that we have

$$y_{i+1} = y_i + \frac{h}{6}\left[f(t_i, y_i) + 4f(t_{i+1/2}, y_{i+1/2}) + f(t_{i+1}, y_{i+1})\right] + O(h^5). \tag{13.44}$$

However, we do not know the values of $y_{i+1/2}$ and y_{i+1}. The fourth-order Runge-Kutta method splits the midpoint evaluations in two steps, that is we have

$$y_{i+1} \approx y_i + \frac{h}{6}\left[f(t_i, y_i) + 2f(t_{i+1/2}, y_{i+1/2}) + 2f(t_{i+1/2}, y_{i+1/2}) + f(t_{i+1}, y_{i+1})\right],$$

since we want to approximate the slope at $y_{i+1/2}$ in two steps. The first two function evaluations are as for the second order Runge-Kutta method. The algorithm is as follows

1. We compute first
$$k_1 = hf(t_i, y_i), \tag{13.45}$$
which is nothing but the slope at t_i.If we stop here we have Euler's method.

2. Then we compute the slope at the midpoint using Euler's method to predict $y_{i+1/2}$, as in the second-order Runge-Kutta method. This leads to the computation of
$$k_2 = hf(t_i + h/2, y_i + k_1/2). \tag{13.46}$$

3. The improved slope at the midpoint is used to further improve the slope of $y_{i+1/2}$ by computing
$$k_3 = hf(t_i + h/2, y_i + k_2/2). \tag{13.47}$$

4. With the latter slope we can in turn predict the value of y_{i+1} via the computation of
$$k_4 = hf(t_i + h, y_i + k_3). \tag{13.48}$$

5. The final algorithm becomes then

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \tag{13.49}$$

Thus, the algorithm consists in first calculating k_1 with t_i, y_1 and f as inputs. Thereafter, we increase the step size by $h/2$ and calculate k_2, then k_3 and finally k_4. With this caveat, we can then obtain the new value for the variable y. It results in four function evaluations, but the accuracy is increased by two orders compared with the second-order Runge-Kutta method. The fourth order Runge-Kutta method has a global truncation error which goes like $O(h^4)$. Fig. 13.1 gives a geometrical interpretation of the fourth-order Runge-Kutta method.

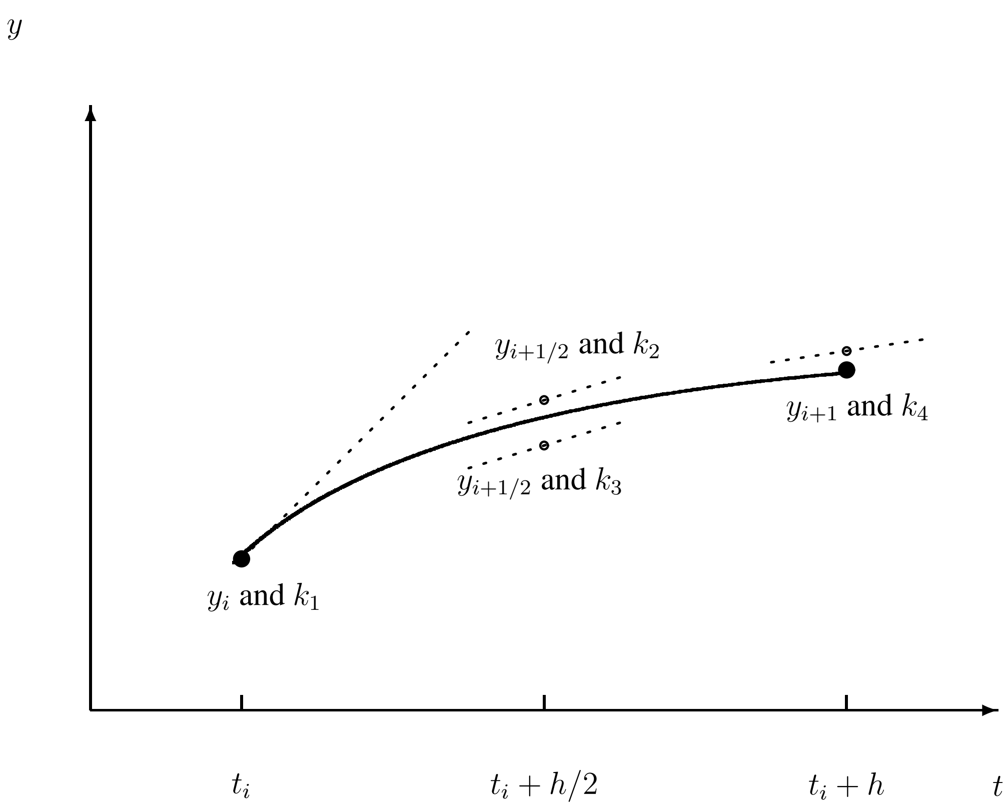


Figure 13.1: Geometrical interpretation of the fourth-order Runge-Kutta method. The derivative is evaluated at four points, once at the intial point, twice at the trial midpoint and once at the trial end-point. These four derivatives constitute one Runge-Kutta step resulting in the final value for $y_{i+1} = y_i + 1/6(k_1 + 2k_2 + 2k_3 + k_4)$.

13.5 Adaptive Runge-Kutta and multistep methods

In preparation.

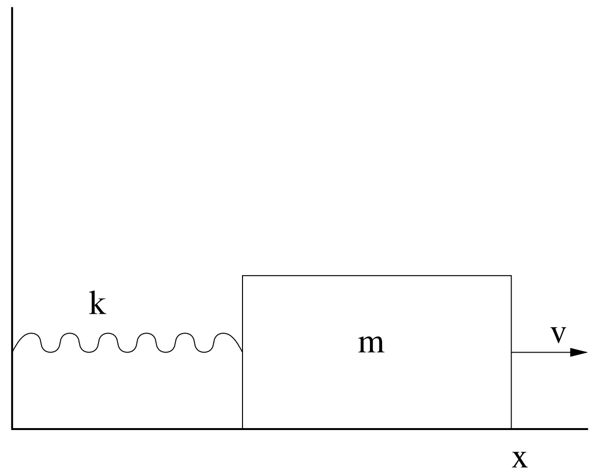


Figure 13.2: Block tied to a wall with a spring tension acting on it.

13.6 Physics examples

13.6.1 Ideal harmonic oscillations

Our first example is the classical case of simple harmonic oscillations, namely a block sliding on a horizontal frictionless surface. The block is tied to a wall with a spring, portrayed in e.g., Fig. 13.2. If the spring is not compressed or stretched too far, the force on the block at a given position x is

$$F = -kx. \tag{13.50}$$

The negative sign means that the force acts to restore the object to an equilibrium position. Newton's equation of motion for this idealized system is then

$$m\frac{d^2x}{dt^2} = -kx, \tag{13.51}$$

or we could rephrase it as

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x = -\omega_0^2 x, \tag{13.52}$$

with the angular frequency $\omega_0^2 = k/m$.

The above differential equation has the advantage that it can be solved analytically with solutions on the form

$$x(t) = Acos(\omega_0 t + \nu),$$

where A is the amplitude and ν the phase constant. This provides in turn an important test for the numerical solution and the development of a program for more complicated cases which cannot be solved analytically.

As mentioned earlier, in certain cases it is possible to rewrite a second-order differential equation as two coupled first-order differential equations. With the position $x(t)$ and the velocity $v(t) = dx/dt$ we can reformulate Newton's equation in the following way

$$\frac{dx(t)}{dt} = v(t), \tag{13.53}$$

and

$$\frac{dv(t)}{dt} = -\omega_0^2 x(t). \tag{13.54}$$

We are now going to solve these equations using the Runge-Kutta method to fourth order discussed previously. Before proceeding however, it is important to note that in addition to the exact solution, we have at least two further tests which can be used to check our solution.

Since functions like cos are periodic with a period 2π, then the solution $x(t)$ has also to be periodic. This means that

$$x(t+T) = x(t), \tag{13.55}$$

with T the period defined as

$$T = \frac{2\pi}{\omega_0} = \frac{2\pi}{\sqrt{k/m}}. \tag{13.56}$$

Observe that T depends only on k/m and not on the amplitude of the solution or the constant ν.

In addition to the periodicity test, the total energy has also to be conserved.

Suppose we choose the initial conditions

$$x(t=0) = 1 \text{ m} \qquad v(t=0) = 0 \text{ m/s}, \tag{13.57}$$

meaning that block is at rest at $t = 0$ but with a potential energy

$$E_0 = \frac{1}{2}kx(t=0)^2 = \frac{1}{2}k. \tag{13.58}$$

The total energy at any time t has however to be conserved, meaning that our solution has to fulfil the condition

$$E_0 = \frac{1}{2}kx(t)^2 + \frac{1}{2}mv(t)^2. \tag{13.59}$$

An algorithm which implements these equations is included below.

1. Choose the initial position and speed, with the most common choice $v(t=0) = 0$ and some fixed value for the position. Since we are going to test our results against the periodicity requirement, it is convenient to set the final time equal $t_f = 2\pi$, where we choose $k/m = 1$. The initial time is set equal to $t_i = 0$. You could alternatively read in the ratio k/m.

2. Choose the method you wish to employ in solving the problem. In the enclosed program we have chosen the fourth-order Runge-Kutta method. Subdivide the time interval $[t_i, t_f]$ into a grid with step size

   $$h = \frac{t_f - t_i}{N},$$

   where N is the number of mesh points.

3. Calculate now the total energy given by

   $$E_0 = \frac{1}{2}kx(t=0)^2 = \frac{1}{2}k.$$

   and use this when checking the numerically calculated energy from the Runge-Kutta iterations.

4. The Runge-Kutta method is used to obtain x_{i+1} and v_{i+1} starting from the previous values x_i and v_i..

5. When we have computed $x(v)_{i+1}$ we upgrade $t_{i+1} = t_i + h$.
6. This iterative process continues till we reach the maximum time $t_f = 2\pi$.
7. The results are checked against the exact solution. Furthermore, one has to check the stability of the numerical solution against the chosen number of mesh points N.

Program to solve the differential equations for a sliding block

The program which implements the above algorithm is presented here, with a corresponding

`http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter13/cpp/program1.cpp`

```
/*     This program solves Newton's equation for a block
       sliding on a horizontal frictionless surface. The block
       is tied  to a wall with a spring, and Newton's equation
       takes the form
            m d^2x/dt^2 =-kx
       with k the spring tension and m the mass of the block.
       The angular frequency is omega^2 = k/m and we set it equal
       1 in this example program.

       Newton's equation is rewritten as two coupled differential
       equations, one for the position x  and one for the velocity v
            dx/dt = v    and
            dv/dt = -x   when we set k/m=1

       We use therefore a two-dimensional array to represent x and v
       as functions of t
       y[0] == x
       y[1] == v
       dy[0]/dt = v
       dy[1]/dt = -x

       The derivatives are calculated by the user defined function
       derivatives.

       The user has to specify the initial velocity (usually v_0=0)
       the number of steps and the initial position. In the programme
       below we fix the time interval [a,b] to [0,2*pi].

*/
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace  std;
// output file as global variable
ofstream ofile;
// function declarations
void derivatives(double, double *, double *);
void initialise ( double&, double&, int&);
void output( double, double *, double);
```

```
void runge_kutta_4(double *, double *, int, double, double,
                   double *, void (*)(double, double *, double *));

int main(int argc, char* argv[])
{
//  declarations of variables
  double *y, *dydt, *yout, t, h, tmax, E0;
  double initial_x, initial_v;
  int i, number_of_steps, n;
  char *outfilename;
  // Read in output file, abort if there are too few command-line
     arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  //  this is the number of differential equations
  n = 2;
  //  allocate space in memory for the arrays containing the derivatives
  dydt = new double[n];
  y = new double[n];
  yout = new double[n];
  // read in the initial position, velocity and number of steps
  initialise (initial_x, initial_v, number_of_steps);
  //  setting initial values, step size and max time tmax
  h = 4.*acos(-1.)/( (double) number_of_steps);   // the step size
  tmax = h*number_of_steps;                        // the final time
  y[0] = initial_x;                                // initial position
  y[1] = initial_v;                                // initial velocity
  t=0.;                                            // initial time
  E0 = 0.5*y[0]*y[0]+0.5*y[1]*y[1];                // the initial total energy
  // now we start solving the differential equations using the RK4 method
  while (t <= tmax){
    derivatives(t, y, dydt);   // initial derivatives
    runge_kutta_4(y, dydt, n, t, h, yout, derivatives);
    for (i = 0; i < n; i++) {
      y[i] = yout[i];
    }
    t += h;
    output(t, y, E0);    // write to file
  }
  delete [] y; delete [] dydt; delete [] yout;
  ofile.close();   // close output file
  return 0;
}   //  End of main function

//      Read in from screen the number of steps,
//      initial position and initial speed
```

```
void initialise (double& initial_x , double& initial_v , int&
   number_of_steps)
{
 cout << "Initial position = ";
 cin >> initial_x;
 cout << "Initial speed = ";
 cin >> initial_v;
 cout << "Number of steps = ";
 cin >> number_of_steps;
}  // end of function initialise

//   this function sets up the derivatives for this special case
void derivatives(double t, double *y, double *dydt)
{
  dydt[0]=y[1];     // derivative of x
  dydt[1]=-y[0]; // derivative of v
} // end of function derivatives

//    function to write out the final results
void output(double t, double *y, double E0)
{
  ofile << setiosflags(ios::showpoint | ios::uppercase);
  ofile << setw(15) << setprecision(8) << t;
  ofile << setw(15) << setprecision(8) << y[0];
  ofile << setw(15) << setprecision(8) << y[1];
  ofile << setw(15) << setprecision(8) << cos(t);
  ofile << setw(15) << setprecision(8) <<
    0.5*y[0]*y[0]+0.5*y[1]*y[1]-E0 << endl;
}  // end of function output

/*   This function upgrades a function y (input as a pointer)
     and returns the result yout, also as a pointer. Note that
     these variables are declared as arrays.  It also receives as
     input the starting value for the derivatives in the pointer
     dydx. It receives also the variable n which represents the
     number of differential equations, the step size h and
     the initial value of x. It receives also the name of the
     function *derivs where the given derivative is computed
*/
void runge_kutta_4(double *y, double *dydx, int n, double x, double h,
                   double *yout, void (*derivs)(double, double *, double
                      *))
{
  int i;
  double      xh,hh,h6;
  double *dym, *dyt, *yt;
  //   allocate space for local vectors
  dym = new double [n];
  dyt =  new double [n];
  yt =  new double [n];
  hh = h*0.5;
  h6 = h/6.;
  xh = x+hh;
```

```
    for (i = 0; i < n; i++) {
      yt[i] = y[i]+hh*dydx[i];
    }
    (*derivs)(xh,yt,dyt);      // computation of k2, eq. 3.60
    for (i = 0; i < n; i++) {
      yt[i] = y[i]+hh*dyt[i];
    }
    (*derivs)(xh,yt,dym); //  computation of k3, eq. 3.61
    for (i=0; i < n; i++) {
      yt[i] = y[i]+h*dym[i];
      dym[i] += dyt[i];
    }
    (*derivs)(x+h,yt,dyt);     // computation of k4, eq. 3.62
    //      now we upgrade y in the array yout
    for (i = 0; i < n; i++){
      yout[i] = y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i]);
    }
    delete []dym;
    delete [] dyt;
    delete [] yt;
  }      //  end of function Runge-kutta 4
```

In Fig. 13.3 we exhibit the development of the difference between the calculated energy and the exact energy at $t = 0$ after two periods and with $N = 1000$ and $N = 10000$ mesh points. This figure demonstrates clearly the need of developing tests for checking the algorithm used. We see that even for $N = 1000$ there is an increasing difference between the computed energy and the exact energy after only two periods.

13.6.2 Damping of harmonic oscillations and external forces

Most oscillatory motion in nature does decrease until the displacement becomes zero. We call such a motion for damped and the system is said to be dissipative rather than conservative. Considering again the simple block sliding on a plane, we could try to implement such a dissipative behavior through a drag force which is proportional to the first derivative of x, i.e., the velocity. We can then expand Eq. (13.52) to

$$\frac{d^2x}{dt^2} = -\omega_0^2 x - \nu\frac{dx}{dt}, \tag{13.60}$$

where ν is the damping coefficient, being a measure of the magnitude of the drag term.

We could however counteract the dissipative mechanism by applying e.g., a periodic external force

$$F(t) = Bcos(\omega t), \tag{13.61}$$

and we rewrite Eq. (13.60) as

$$\frac{d^2x}{dt^2} = -\omega_0^2 x - \nu\frac{dx}{dt} + F(t). \tag{13.62}$$

Although we have specialized to a block sliding on a surface, the above equations are rather general for quite many physical systems.

If we replace x by the charge Q, ν with the resistance R, the velocity with the current I, the inductance L with the mass m, the spring constant with the inverse capacitance C and the force F with the voltage

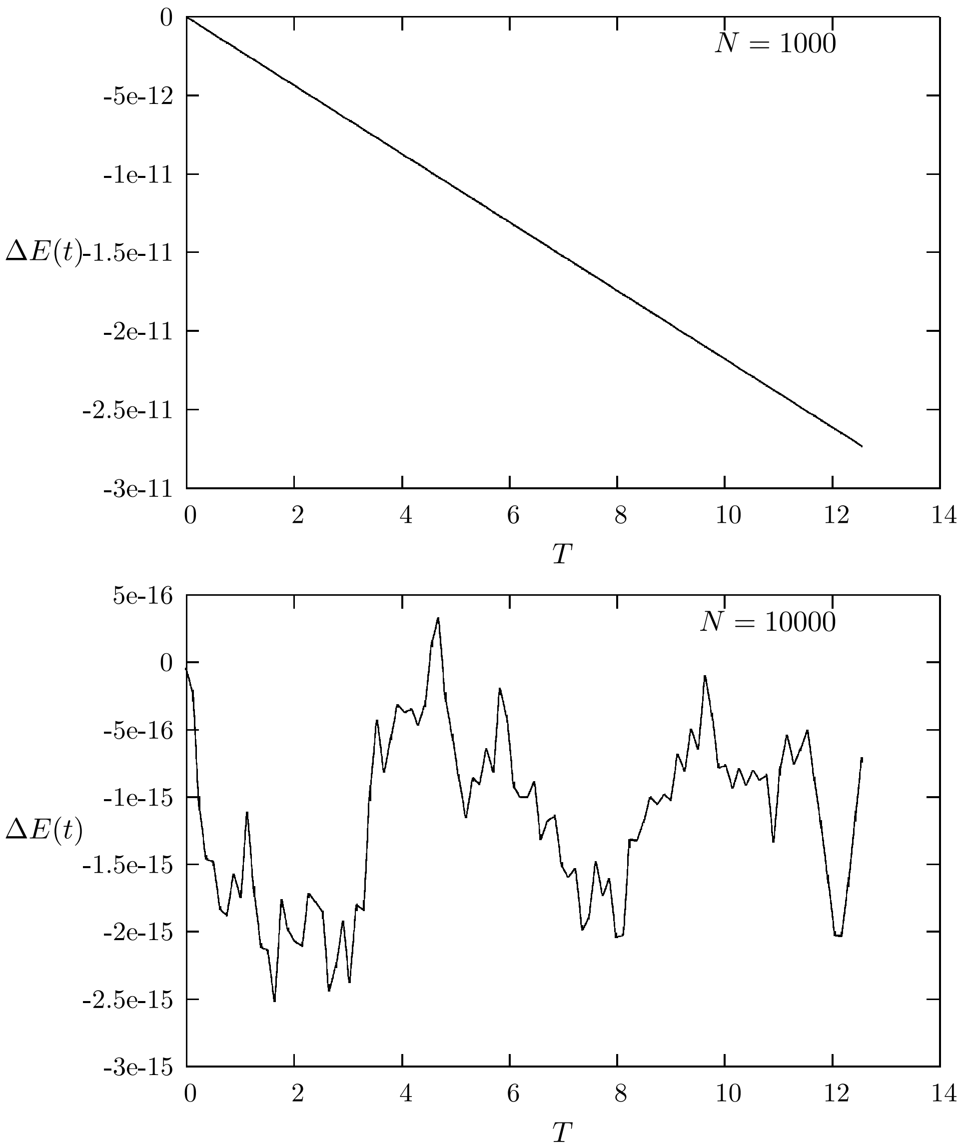


Figure 13.3: Plot of $\Delta E(t) = E_0 - E_{\text{computed}}$ for $N = 1000$ and $N = 10000$ time steps up to two periods. The initial position $x_0 = 1$ m and initial velocity $v_0 = 0$ m/s. The mass and spring tension are set to $k = m = 1$.

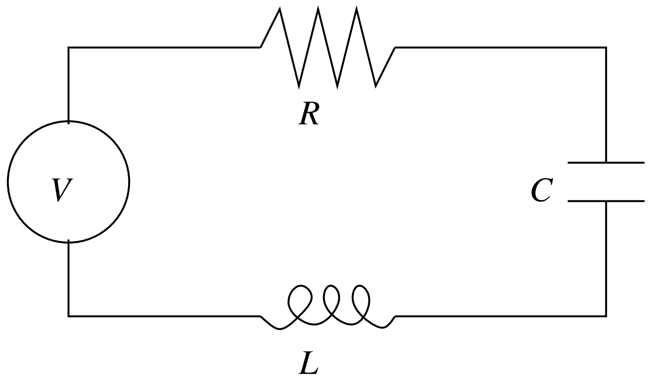


Figure 13.4: Simple RLC circuit with a voltage source V.

drop V, we rewrite Eq. (13.62) as

$$L\frac{d^2Q}{dt^2} + \frac{Q}{C} + R\frac{dQ}{dt} = V(t). \tag{13.63}$$

The circuit is shown in Fig. 13.4.

How did we get there? We have defined an electric circuit which consists of a resistance R with voltage drop IR, a capacitor with voltage drop Q/C and an inductor L with voltage drop LdI/dt. The circuit is powered by an alternating voltage source and using Kirchhoff's law, which is a consequence of energy conservation, we have

$$V(t) = IR + LdI/dt + Q/C, \tag{13.64}$$

and using

$$I = \frac{dQ}{dt}, \tag{13.65}$$

we arrive at Eq. (13.63).

This section was meant to give you a feeling of the wide range of applicability of the methods we have discussed. However, before leaving this topic entirely, we'll dwelve into the problems of the pendulum, from almost harmonic oscillations to chaotic motion!

13.6.3 The pendulum, a nonlinear differential equation

Consider a pendulum with mass m at the end of a rigid rod of length l attached to say a fixed frictionless pivot which allows the pendulum to move freely under gravity in the vertical plane as illustrated in Fig. 13.5.

The angular equation of motion of the pendulum is again given by Newton's equation, but now as a nonlinear differential equation

$$ml\frac{d^2\theta}{dt^2} + mgsin(\theta) = 0, \tag{13.66}$$

with an angular velocity and acceleration given by

$$v = l\frac{d\theta}{dt}, \tag{13.67}$$

and

$$a = l\frac{d^2\theta}{dt^2}. \tag{13.68}$$

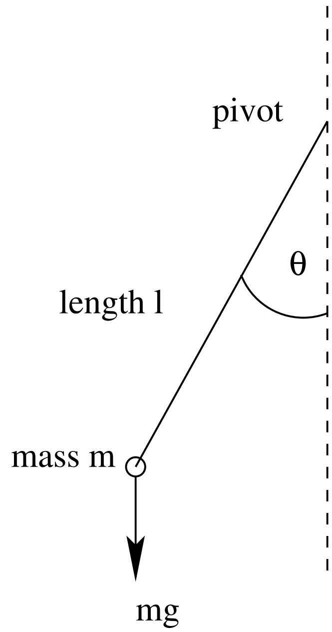


Figure 13.5: A simple pendulum.

For small angles, we can use the approximation

$$sin(\theta) \approx \theta.$$

and rewrite the above differential equation as

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\theta, \tag{13.69}$$

which is exactly of the same form as Eq. (13.52). We can thus check our solutions for small values of θ against an analytical solution. The period is now

$$T = \frac{2\pi}{\sqrt{l/g}}. \tag{13.70}$$

We do however expect that the motion will gradually come to an end due a viscous drag torque acting on the pendulum. In the presence of the drag, the above equation becomes

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = 0, \tag{13.71}$$

where ν is now a positive constant parameterizing the viscosity of the medium in question. In order to maintain the motion against viscosity, it is necessary to add some external driving force. We choose here, in analogy with the discussion about the electric circuit, a periodic driving force. The last equation becomes then

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = Acos(\omega t), \tag{13.72}$$

with A and ω two constants representing the amplitude and the angular frequency respectively. The latter is called the driving frequency.

If we now define the natural frequency

$$\omega_0 = \sqrt{g/l}, \tag{13.73}$$

the so-called natural frequency and the new dimensionless quantities

$$\hat{t} = \omega_0 t, \tag{13.74}$$

with the dimensionless driving frequency

$$\hat{\omega} = \frac{\omega}{\omega_0}, \tag{13.75}$$

and introducing the quantity Q, called the *quality factor*,

$$Q = \frac{mg}{\omega_0 \nu}, \tag{13.76}$$

and the dimensionless amplitude

$$\hat{A} = \frac{A}{mg} \tag{13.77}$$

we can rewrite Eq. (13.72) as

$$\frac{d^2\theta}{d\hat{t}^2} + \frac{1}{Q}\frac{d\theta}{d\hat{t}} + sin(\theta) = \hat{A}cos(\hat{\omega}\hat{t}). \tag{13.78}$$

This equation can in turn be recast in terms of two coupled first-order differential equations as follows

$$\frac{d\theta}{d\hat{t}} = \hat{v}, \tag{13.79}$$

and

$$\frac{d\hat{v}}{d\hat{t}} = -\frac{\hat{v}}{Q} - sin(\theta) + \hat{A}cos(\hat{\omega}\hat{t}). \tag{13.80}$$

These are the equations to be solved. The factor Q represents the number of oscillations of the undriven system that must occur before its energy is significantly reduced due to the viscous drag. The amplitude $\hat{A}$ is measured in units of the maximum possible gravitational torque while $\hat{\omega}$ is the angular frequency of the external torque measured in units of the pendulum's natural frequency.

13.6.4 Spinning magnet

Another simple example is that of e.g., a compass needle that is free to rotate in a periodically reversing magnetic field perpendicular to the axis of the needle. The equation is then

$$\frac{d^2\theta}{dt^2} = -\frac{\mu}{I}B_0 cos(\omega t)sin(\theta), \tag{13.81}$$

where θ is the angle of the needle with respect to a fixed axis along the field, μ is the magnetic moment of the needle, I its moment of inertia and B_0 and ω the amplitude and angular frequency of the magnetic field respectively.

13.7 Physics Project: the pendulum

13.7.1 Analytic results for the pendulum

Although the solution to the equations for the pendulum can only be obtained through numerical efforts, it is always useful to check our numerical code against analytic solutions. For small angles θ, we have $sin\theta \approx \theta$ and our equations become

$$\frac{d\theta}{d\hat{t}} = \hat{v}, \tag{13.82}$$

and

$$\frac{d\hat{v}}{d\hat{t}} = -\frac{\hat{v}}{Q} - \theta + \hat{A}cos(\hat{\omega}\hat{t}). \tag{13.83}$$

These equations are linear in the angle θ and are similar to those of the sliding block or the RLC circuit. With given initial conditions $\hat{v}_0$ and θ_0 they can be solved analytically to yield

$$\begin{aligned}\theta(t) &= \left[\theta_0 - \frac{\hat{A}(1-\hat{\omega}^2)}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}\right] e^{-\tau/2Q} cos(\sqrt{1-\tfrac{1}{4Q^2}}\tau) \\ &+ \left[\hat{v}_0 + \frac{\theta_0}{2Q} - \frac{\hat{A}(1-3\hat{\omega}^2)/2Q}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}\right] e^{-\tau/2Q} sin(\sqrt{1-\tfrac{1}{4Q^2}}\tau) + \frac{\hat{A}(1-\hat{\omega}^2)cos(\hat{\omega}\tau)+\frac{\hat{\omega}}{Q}sin(\hat{\omega}\tau)}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2},\end{aligned} \tag{13.84}$$

and

$$\begin{aligned}\hat{v}(t) &= \left[\hat{v}_0 - \frac{\hat{A}\hat{\omega}^2/Q}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}\right] e^{-\tau/2Q} cos(\sqrt{1-\tfrac{1}{4Q^2}}\tau) \\ &- \left[\theta_0 + \frac{\hat{v}_0}{2Q} - \frac{\hat{A}[(1-\hat{\omega}^2)-\hat{\omega}^2/Q^2]}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}\right] e^{-\tau/2Q} sin(\sqrt{1-\tfrac{1}{4Q^2}}\tau) + \frac{\hat{\omega}\hat{A}[-(1-\hat{\omega}^2)sin(\hat{\omega}\tau)+\frac{\hat{\omega}}{Q}cos(\hat{\omega}\tau)]}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2},\end{aligned} \tag{13.85}$$

with $Q > 1/2$. The first two terms depend on the initial conditions and decay exponentially in time. If we wait long enough for these terms to vanish, the solutions become independent of the initial conditions and the motion of the pendulum settles down to the following simple orbit in phase space

$$\theta(t) = \frac{\hat{A}(1-\hat{\omega}^2)cos(\hat{\omega}\tau) + \frac{\hat{\omega}}{Q}sin(\hat{\omega}\tau)}{(1-\hat{\omega}^2)^2 + \hat{\omega}^2/Q^2}, \tag{13.86}$$

and

$$\hat{v}(t) = \frac{\hat{\omega}\hat{A}[-(1-\hat{\omega}^2)sin(\hat{\omega}\tau) + \frac{\hat{\omega}}{Q}cos(\hat{\omega}\tau)]}{(1-\hat{\omega}^2)^2 + \hat{\omega}^2/Q^2}, \tag{13.87}$$

tracing the closed phase-space curve

$$\left(\frac{\theta}{\tilde{A}}\right)^2 + \left(\frac{\hat{v}}{\hat{\omega}\tilde{A}}\right)^2 = 1 \tag{13.88}$$

with

$$\tilde{A} = \frac{\hat{A}}{\sqrt{(1-\hat{\omega}^2)^2 + \hat{\omega}^2/Q^2}}. \tag{13.89}$$

This curve forms an ellipse whose principal axes are θ and $\hat{v}$. This curve is closed, as we will see from the examples below, implying that the motion is periodic in time, the solution repeats itself exactly after each period $T = 2\pi/\hat{\omega}$. Before we discuss results for various frequencies, quality factors and amplitudes, it is instructive to compare different numerical methods. In Fig. 13.6 we show the angle θ as function of time τ for the case with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 0.5$. The length is set equal to 1 m and mass of the pendulum is set equal to 1 kg. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.01$. Four different methods have been used to solve the equations, Euler's method from Eq. (13.17), Euler-Richardson's method in Eqs. (13.32)-(13.33) and finally the fourth-order Runge-Kutta scheme RK4. We note that after few time steps, we obtain the classical harmonic motion. We would have obtained a similar picture if we were to switch off the external force, $\hat{A} = 0$ and set the frictional damping to zero, i.e., $Q = 0$. Then, the qualitative picture is that of an idealized harmonic oscillation without damping. However, we see that Euler's method performs poorly and after a few steps its algorithmic simplicity leads to results which deviate considerably from the other methods. In the discussion hereafter we will thus limit ourselves to

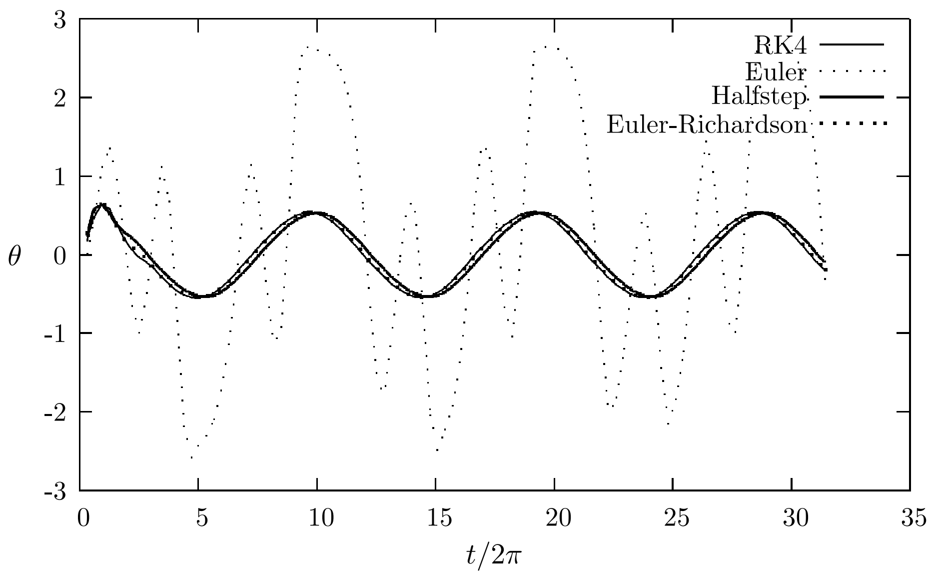


Figure 13.6: Plot of θ as function of time τ with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 0.5$. The mass and length of the pendulum are set equal to 1. The initial velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.01$. Four different methods have been used to solve the equations, Euler's method from Eq. (13.17), the half-step method, Euler-Richardson's method in Eqs. (13.32)-(13.33) and finally the fourth-order Runge-Kutta scheme RK4. Only $N = 100$ integration points have been used for a time interval $t \in [0, 10\pi]$.

present results obtained with the fourth-order Runge-Kutta method.

The corresponding phase space plot is shown in Fig. 13.7, for the same parameters as in Fig. 13.6. We observe here that the plot moves towards an ellipse with periodic motion. This stable phase-space curve is called a periodic attractor. It is called attractor because, irrespective of the initial conditions, the trajectory in phase-space tends asymptotically to such a curve in the limit $\tau \to \infty$. It is called periodic, since it exhibits periodic motion in time, as seen from Fig. 13.6. In addition, we should note that this periodic motion shows what we call resonant behavior since the the driving frequency of the force approaches the natural frequency of oscillation of the pendulum. This is essentially due to the fact that we are studying a linear system, yielding the well-known periodic motion. The non-linear system exhibits a much richer set of solutions and these can only be studied numerically.

In order to go beyond the well-known linear approximation we change the initial conditions to say $\theta_0 = 0.3$ but keep the other parameters equal to the previous case. The curve for θ is shown in Fig. 13.8. The corresponding phase-space curve is shown in Fig. 13.9. This curve demonstrates that with the above given sets of parameters, after a certain number of periods, the phase-space curve stabilizes to the same

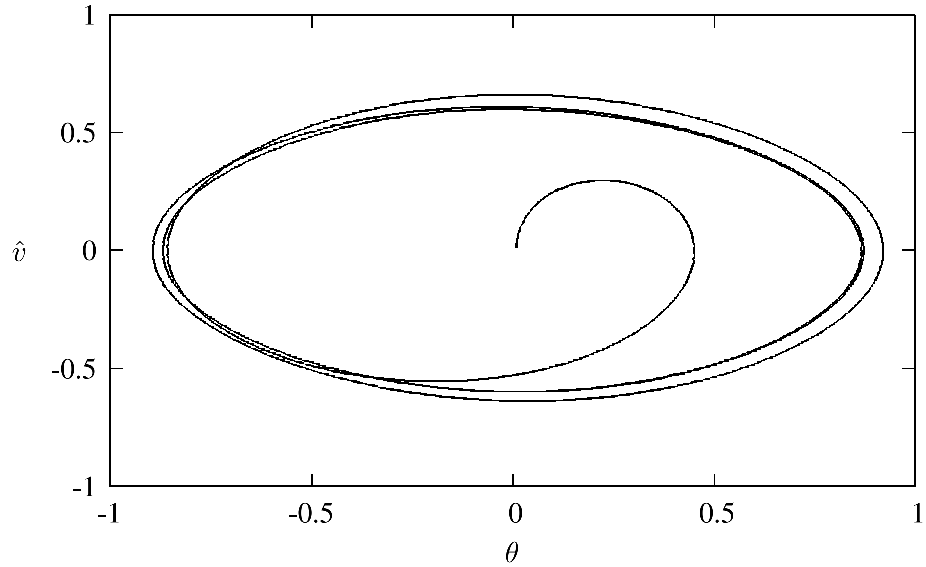


Figure 13.7: Phase-space curve of a linear damped pendulum with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 0.5$. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.01$.

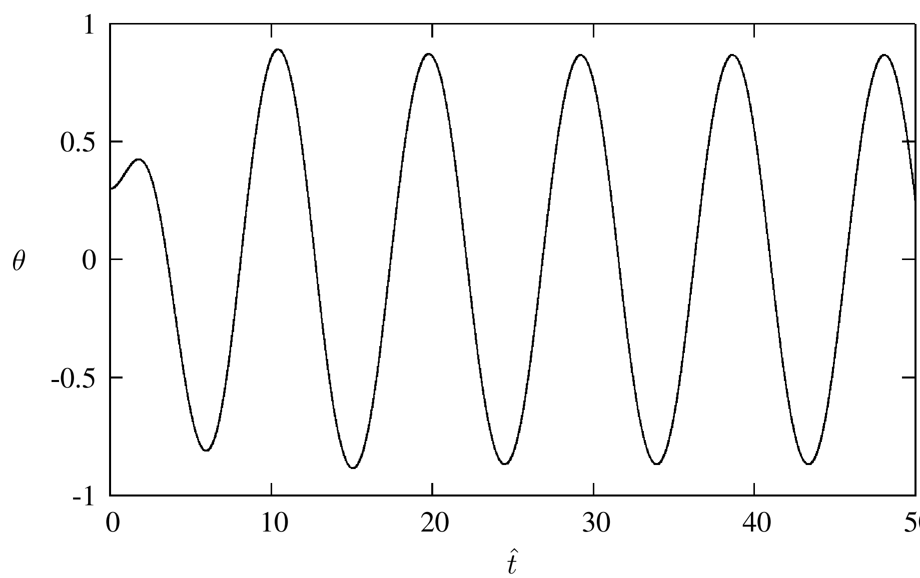


Figure 13.8: Plot of θ as function of time τ with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 0.5$. The mass of the pendulum is set equal to 1 kg and its length to 1 m. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.3$.

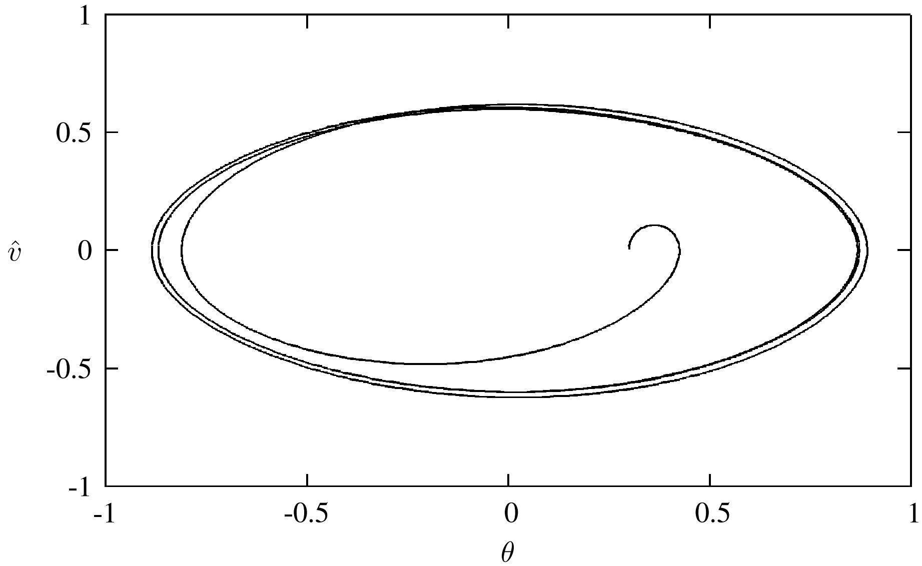


Figure 13.9: Phase-space curve with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 0.5$. The mass of the pendulum is set equal to 1 kg and its length $l = 1$ m.. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.3$.

curve as in the previous case, irrespective of initial conditions. However, it takes more time for the pendulum to establish a periodic motion and when a stable orbit in phase-space is reached the pendulum moves in accordance with the driving frequency of the force. The qualitative picture is much the same as previously. The phase-space curve displays again a final periodic attractor.

If we now change the strength of the amplitude to $\hat{A} = 1.35$ we see in Fig. 13.10 that θ as function of time exhibits a rather different behavior from Fig. 13.8, even though the initial conditions and all other parameters except $\hat{A}$ are the same. The phase-space curve is shown in Fig. 13.11.

We will explore these topics in more detail in Section 13.8.1 where we extend our discussion to the phenomena of period doubling and its link to chaotic motion.

13.7.2 The pendulum code

The program used to obtain the results discussed above is presented here. The enclosed code solves the pendulum equations for any angle θ with an external force $Acos(\omega t)$. It employes several methods for solving the two coupled differential equations, from Euler's method to adaptive size methods coupled with fourth-order Runge-Kutta. It is straightforward to apply this program to other systems which exhibit harmonic oscillations or change the functional form of the external force.

We have also introduced the class concept, where we define various methods for solving ordinary and coupled first order differential equations via the . class pendulum. This methods access variables which belong only to this particular class via the **private** declaration. As such, the methods we list here can easily be reused by other types of ordinary differential equations. In the code below, we list only the fourth order Runge Kutta method, which was used to generate the above figures. For the full code see programs/chapter13/program2.cpp.

http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter13/cpp/program2.cpp

```
#include <stdio.h>
```

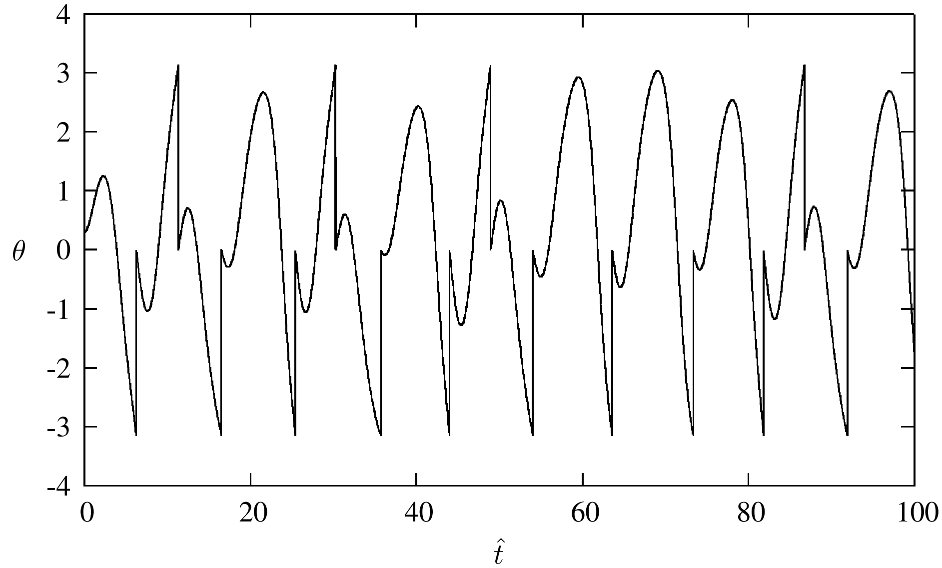


Figure 13.10: Plot of θ as function of time τ with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 1.35$. The mass of the pendulum is set equal to 1 kg and its length to 1 m. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.3$. Every time θ passes the value $\pm\pi$ we reset its value to swing between $\theta \in [-\pi, pi]$. This gives the vertical jumps in amplitude.

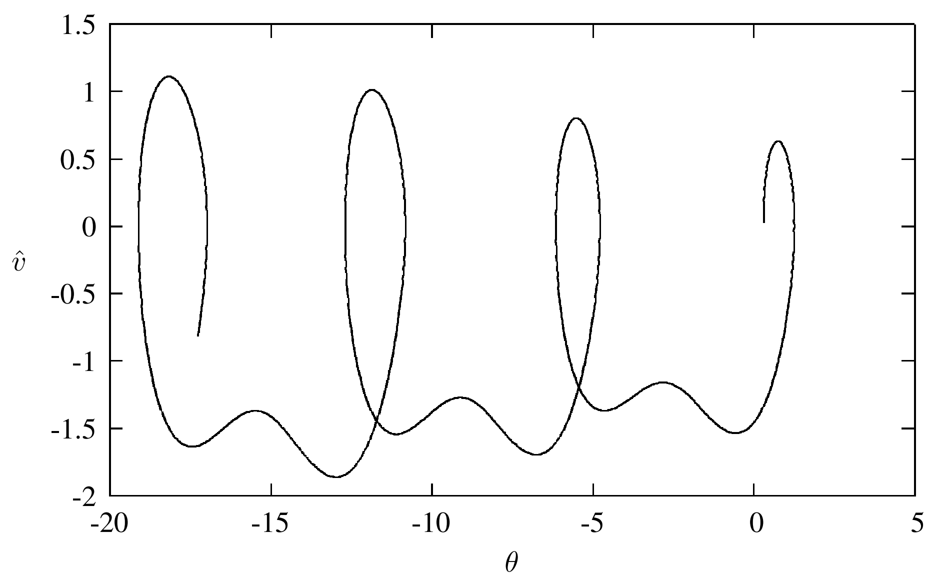


Figure 13.11: Phase-space curve after 10 periods with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 1.35$. The mass of the pendulum is set equal to 1 kg and its length $l = 1$ m. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.3$.

```
#include <iostream.h>
#include <math.h>
#include <fstream.h>
/*
Different methods for solving ODEs are presented
We are solving the following eqation:

m*l*(phi)'' + viscosity*(phi)' + m*g*sin(phi) = A*cos(omega*t)

If you want to solve similar equations with other values you have to
rewrite the methods 'derivatives' and 'initialise' and change the
    variables in the private
part of the class Pendulum

At first we rewrite the equation using the following definitions:

omega_0 = sqrt(g*l)
t_roof = omega_0*t
omega_roof = omega/omega_0
Q = (m*g)/(omega_0*reib)
A_roof = A/(m*g)

and we get a dimensionless equation

(phi)'' + 1/Q*(phi)' + sin(phi) = A_roof*cos(omega_roof*t_roof)

This equation can be written as two equations of first order:

(phi)' = v
(v)' = -v/Q - sin(phi) +A_roof*cos(omega_roof*t_roof)

All numerical methods are applied to the last two equations.
The algorithms are taken from the book "An introduction to computer
    simulation methods"
*/

class pendelum
 {
 private:
   double Q, A_roof, omega_0, omega_roof,g; //
   double y[2];          //for the initial-values of phi and v
   int n;                // how many steps
   double delta_t,delta_t_roof;
// Definition of methods to solve ODEs
 public:
   void derivatives(double,double*,double*);
   void initialise();
   void euler();
   void euler_cromer();
   void midpoint();
   void euler_richardson();
   void half_step();
   void rk2(); //runge-kutta-second-order
```

```
    void rk4_step(double ,double *,double *,double ); // we need it in
        function rk4() and asc()
    void rk4(); //runge-kutta-fourth-order
    void asc(); //runge-kutta-fourth-order with adaptive stepsize control
  };

//  This function defines the particular coupled first order ODEs
  void pendelum::derivatives(double t, double* in, double* out)
  { /* Here we are calculating the derivatives at (dimensionless) time t
       'in' are the values of phi and v, which are used for the calculation
       The results are given to 'out' */

    out[0]=in[1];                 //out[0] = (phi)'  = v
    if(Q)
      out[1]=-in[1]/((double)Q)-sin(in[0])+A_roof*cos(omega_roof*t);  //out
          [1] = (phi)''
    else
      out[1]=-sin(in[0])+A_roof*cos(omega_roof*t);  //out[1] = (phi)''
  }
// Here we define all input parameters.
  void pendelum::initialise()
  {
    double m,l,omega,A,viscosity,phi_0,v_0,t_end;
    cout<<"Solving the differential eqation of the pendulum!\n";
    cout<<"We have a pendulum with mass m, length l. Then we have a
       periodic force with amplitude A and omega\n";
    cout<<"Furthermore there is a viscous drag coefficient.\n";
    cout<<"The initial conditions at t=0 are phi_0 and v_0\n";
    cout<<"Mass m: ";
    cin>>m;
    cout<<"length l: ";
    cin>>l;
    cout<<"omega of the force: ";
    cin>>omega;
    cout<<"amplitude of the force: ";
    cin>>A;
    cout<<"The value of the viscous drag constant (viscosity): ";
    cin>>viscosity;
    cout<<"phi_0: ";
    cin>>y[0];
    cout<<"v_0: ";
    cin>>y[1];
    cout<<"Number of time steps or integration steps:";
    cin>>n;
    cout<<"Final time steps as multiplum of pi:";
    cin>>t_end;
    t_end *= acos(-1.);
    g=9.81;
    // We need the following values:
    omega_0=sqrt(g/((double)l));      // omega of the pendulum
    if (viscosity)  Q= m*g/((double)omega_0*viscosity);
    else Q=0; //calculating Q
    A_roof=A/((double)m*g);
```

```
    omega_roof=omega/((double)omega_0);
    delta_t_roof=omega_0*t_end/((double)n);    //delta_t without dimension
    delta_t=t_end/((double)n);
  }
// fourth order Run
  void pendelum::rk4_step(double t,double *yin,double *yout,double delta_t)
  {
    /*
      The function calculates one step of fourth-order-runge-kutta-method
      We will need it for the normal fourth-order-Runge-Kutta-method and
      for RK-method with adaptive stepsize control

      The function calculates the value of y(t + delta_t) using fourth-
          order-RK-method
      Input: time t and the stepsize delta_t, yin (values of phi and v at
          time t)
      Output: yout (values of phi and v at time t+delta_t)

    */
    double k1[2],k2[2],k3[2],k4[2],y_k[2];
    // Calculation of k1
    derivatives(t,yin,yout);
    k1[1]=yout[1]*delta_t;
    k1[0]=yout[0]*delta_t;
    y_k[0]=yin[0]+k1[0]*0.5;
    y_k[1]=yin[1]+k1[1]*0.5;
    /*Calculation of k2 */
    derivatives(t+delta_t*0.5,y_k,yout);
    k2[1]=yout[1]*delta_t;
    k2[0]=yout[0]*delta_t;
    y_k[0]=yin[0]+k2[0]*0.5;
    y_k[1]=yin[1]+k2[1]*0.5;
    /* Calculation of k3 */
    derivatives(t+delta_t*0.5,y_k,yout);
    k3[1]=yout[1]*delta_t;
    k3[0]=yout[0]*delta_t;
    y_k[0]=yin[0]+k3[0];
    y_k[1]=yin[1]+k3[1];
    /*Calculation of k4 */
    derivatives(t+delta_t,y_k,yout);
    k4[1]=yout[1]*delta_t;
    k4[0]=yout[0]*delta_t;
    /*Calculation of new values of phi and v */
    yout[0]=yin[0]+1.0/6.0*(k1[0]+2*k2[0]+2*k3[0]+k4[0]);
    yout[1]=yin[1]+1.0/6.0*(k1[1]+2*k2[1]+2*k3[1]+k4[1]);
  }

  void pendelum::rk4()
  {
    /*We are using the fourth-order-Runge-Kutta-algorithm
      We have to calculate the parameters k1, k2, k3, k4 for v and phi,
      so we use to arrays k1[2] and k2[2] for this
      k1[0], k2[0] are the parameters for phi,
```

```
      k1[1], k2[1] are the parameters for v
    */

    int i;
    double t_h;
    double yout[2],y_h[2]; //k1[2],k2[2],k3[2],k4[2],y_k[2];

    t_h=0;
    y_h[0]=y[0];  //phi
    y_h[1]=y[1];  //v
    ofstream fout("rk4.out");
    fout.setf(ios::scientific);
    fout.precision(20);
    for(i=1; i<=n; i++){
      rk4_step(t_h,y_h,yout,delta_t_roof);
      fout<<i*delta_t<<"\t\t"<<yout[0]<<"\t\t"<<yout[1]<<"\n";
      t_h+=delta_t_roof;
      y_h[0]=yout[0];
      y_h[1]=yout[1];
    }
    fout.close;
  }

  int main()
  {
    pendelum testcase;
    testcase.initialise();
    testcase.rk4();
    return 0;
  }  // end of main function
```

13.8 Exercises and projects

Project 13.1: studies of neutron stars

In the pendulum example we rewrote the equations as two differential equations in terms of so-called dimensionless variables. One should always do that. There are at least two good reasons for doing this.

- By rewriting the equations as dimensionless ones, the program will most likely be easier to read, with hopefully a better possibility of spotting eventual errors. In addtion, the various constants which are pulled out of the equations in the process of rendering the equations dimensionless, are reintroduced at the end of the calculation. If one of these constants is not correctly defined, it is easier to spot an eventual error.

- In many physics applications, variables which enter a differential equation, may differ by orders of magnitude. If we were to insist on not using dimensionless quantities, such differences can cause serious problems with respect to loss of numerical precision.

An example which demonstrates these features is the set of equations for gravitational equilibrium of a neutron star. We will not solve these equations numerically here, rather, we will limit ourselves to merely rewriting these equations in a dimensionless form.

The equations for a neutron star

The discovery of the neutron by Chadwick in 1932 prompted Landau to predict the existence of neutron stars. The birth of such stars in supernovae explosions was suggested by Baade and Zwicky 1934. First theoretical neutron star calculations were performed by Tolman, Oppenheimer and Volkoff in 1939 and Wheeler around 1960. Bell and Hewish were the first to discover a neutron star in 1967 as a *radio pulsar*. The discovery of the rapidly rotating Crab pulsar (rapidly rotating neutron star) in the remnant of the Crab supernova observed by the chinese in 1054 A.D. confirmed the link to supernovae. Radio pulsars are rapidly rotating with periods in the range 0.033 s $\leq P \leq 4.0$ s. They are believed to be powered by rotational energy loss and are rapidly spinning down with period derivatives of order $\dot{P} \sim 10^{-12} - 10^{-16}$. Their high magnetic field B leads to dipole magnetic braking radiation proportional to the magnetic field squared. One estimates magnetic fields of the order of $B \sim 10^{11} - 10^{13}$ G. The total number of pulsars discovered so far has just exceeded 1000 before the turn of the millenium and the number is increasing rapidly.

The physics of compact objects like neutron stars offers an intriguing interplay between nuclear processes and astrophysical observables, see Refs. [71–73] for further information and references on the physics of neutron stars. Neutron stars exhibit conditions far from those encountered on earth; typically, expected densities ρ of a neutron star interior are of the order of 10^3 or more times the density $\rho_d \approx 4 \cdot 10^{11}$ g/cm^3 at 'neutron drip', the density at which nuclei begin to dissolve and merge together. Thus, the determination of an equation of state (EoS) for dense matter is essential to calculations of neutron star properties. The EoS determines properties such as the mass range, the mass-radius relationship, the crust thickness and the cooling rate. The same EoS is also crucial in calculating the energy released in a supernova explosion.

Clearly, the relevant degrees of freedom will not be the same in the crust region of a neutron star, where the density is much smaller than the saturation density of nuclear matter, and in the center of the star, where density is so high that models based solely on interacting nucleons are questionable. Neutron star models including various so-called realistic equations of state result in the following general picture of the interior of a neutron star. The surface region, with typical densities $\rho < 10^6$ g/cm^3, is a region in which temperatures and magnetic fields may affect the equation of state. The outer crust for 10^6 g/cm^3 $< \rho < 4 \cdot 10^{11}$g/cm^3 is a solid region where a Coulomb lattice of heavy nuclei coexist in β-equilibrium with a relativistic degenerate electron gas. The inner crust for $4 \cdot 10^{11}$ g/cm^3 $< \rho < 2 \cdot 10^{14}$g/cm^3 consists of a lattice of neutron-rich nuclei together with a superfluid neutron gas and an electron gas. The neutron liquid for $2 \cdot 10^{14}$ g/cm^3 $< \rho < \cdot 10^{15}$g/cm^3 contains mainly superfluid neutrons with a smaller concentration of superconducting protons and normal electrons. At higher densities, typically $2-3$ times nuclear matter saturation density, interesting phase transitions from a phase with just nucleonic degrees of freedom to quark matter may take place. Furthermore, one may have a mixed phase of quark and nuclear matter, kaon or pion condensates, hyperonic matter, strong magnetic fields in young stars etc.

Equilibrium equations

If the star is in thermal equilibrium, the gravitational force on every element of volume will be balanced by a force due to the spacial variation of the pressure P. The pressure is defined by the equation of state (EoS), recall e.g., the ideal gas $P = Nk_BT$. The gravitational force which acts on an element of volume at a distance r is given by

$$F_{Grav} = -\frac{Gm}{r^2}\rho/c^2, \tag{13.90}$$

where G is the gravitational constant, $\rho(r)$ is the mass density and $m(r)$ is the total mass inside a radius r. The latter is given by

$$m(r) = \frac{4\pi}{c^2}\int_0^r \rho(r')r'^2 dr' \tag{13.91}$$

which gives rise to a differential equation for mass and density

$$\frac{dm}{dr} = 4\pi r^2\rho(r)/c^2. \tag{13.92}$$

When the star is in equilibrium we have

$$\frac{dP}{dr} = -\frac{Gm(r)}{r^2}\rho(r)/c^2. \tag{13.93}$$

The last equations give us two coupled first-order differential equations which determine the structure of a neutron star when the EoS is known.

The initial conditions are dictated by the mass being zero at the center of the star, i.e., when $r = 0$, we have $m(r = 0) = 0$. The other condition is that the pressure vanishes at the surface of the star. This means that at the point where we have $P = 0$ in the solution of the differential equations, we get the total radius R of the star and the total mass $m(r = R)$. The mass-energy density when $r = 0$ is called the central density ρ_s. Since both the final mass M and total radius R will depend on ρ_s, a variation of this quantity will allow us to study stars with different masses and radii.

Dimensionless equations

When we now attempt the numerical solution, we need however to rescale the equations so that we deal with dimensionless quantities only. To understand why, consider the value of the gravitational constant G and the possible final mass $m(r = R) = M_R$. The latter is normally of the order of some solar masses $M_\odot$, with $M_\odot = 1.989 \times 10^{30}$ Kg. If we wish to translate the latter into units of MeV/c^2, we will have that $M_R \sim 10^{60}$ MeV/c^2. The gravitational constant is in units of $G = 6.67 \times 10^{-45} \times \hbar c\ (MeV/c^2)^{-2}$. It is then easy to see that including the relevant values for these quantities in our equations will most likely yield large numerical roundoff errors when we add a huge number $\frac{dP}{dr}$ to a smaller number P in order to obtain the new pressure. We list here the units of the various quantities and in case of physical constants, also their values. A bracketed symbol like $[P]$ stands for the unit of the quantity inside the brackets.

We introduce therefore dimensionless quantities for the radius $\hat{r} = r/R_0$, mass-energy density $\hat{\rho} = \rho/\rho_s$, pressure $\hat{P} = P/\rho_s$ and mass $\hat{m} = m/M_0$.

The constants M_0 and R_0 can be determined from the requirements that the equations for $\frac{dm}{dr}$ and $\frac{dP}{dr}$ should be dimensionless. This gives

$$\frac{dM_0\hat{m}}{dR_0\hat{r}} = 4\pi R_0^2\hat{r}^2\rho_s\hat{\rho}, \tag{13.94}$$

yielding

$$\frac{d\hat{m}}{d\hat{r}} = 4\pi R_0^3\hat{r}^2\rho_s\hat{\rho}/M_0. \tag{13.95}$$

If these equations should be dimensionless we must demand that

$$4\pi R_0^3\rho_s/M_0 = 1. \tag{13.96}$$

Quantity	Units
$[P]$	MeVfm^{-3}
$[\rho]$	MeVfm^{-3}
$[n]$	fm^{-3}
$[m]$	MeVc^{-2}
$M_\odot$	1.989×10^{30} Kg= 1.1157467×10^{60} MeVc^{-2}
1 Kg	= $10^{30}/1.78266270D0$ MeVc^{-2}
$[r]$	m
G	$\hbar c 6.67259 \times 10^{-45}$ MeV^{-2}c^{-4}
$\hbar c$	197.327 MeVfm

Correspondingly, we have for the pressure equation

$$\frac{d\rho_s \hat{P}}{dR_0\hat{r}} = -GM_0\frac{\hat{m}\rho_s\hat{\rho}}{R_0^2\hat{r}^2} \tag{13.97}$$

and since this equation should also be dimensionless, we will have

$$GM_0/R_0 = 1. \tag{13.98}$$

This means that the constants R_0 and M_0 which will render the equations dimensionless are given by

$$R_0 = \frac{1}{\sqrt{\rho_s G 4\pi}}, \tag{13.99}$$

and

$$M_0 = \frac{4\pi\rho_s}{(\sqrt{\rho_s G 4\pi})^3}. \tag{13.100}$$

However, since we would like to have the radius expressed in units of 10 km, we should multiply R_0 by 10^{-19}, since 1 fm = 10^{-15} m. Similarly, M_0 will come in units of MeV/c^2, and it is convenient therefore to divide it by the mass of the sun and express the total mass in terms of solar masses $M_\odot$.

The differential equations read then

$$\frac{d\hat{P}}{d\hat{r}} = -\frac{\hat{m}\hat{\rho}}{\hat{r}^2}, \qquad \frac{d\hat{m}}{d\hat{r}} = \hat{r}^2\hat{\rho}. \tag{13.101}$$

In the solution of our problem, we will assume that the mass-energy density is given by a simple parametrization from Bethe and Johnson [74]. This parametrization gives ρ as a function of the number density $n = N/V$, with N the total number of baryons in a volume V. It reads

$$\rho(n) = 236 \times n^{2.54} + nm_n, \tag{13.102}$$

where $m_n = 938.926$MeV/c^2, the mass of the neutron (averaged). This means that since $[n]$ =fm^{-3}, we have that the dimension of ρ is $[\rho]$ =MeV/c^{2}fm^{-3}. Through the thermodynamic relation

$$P = -\frac{\partial E}{\partial V}, \tag{13.103}$$

where E is the energy in units of MeV/c^2 we have

$$P(n) = n\frac{\partial \rho(n)}{\partial n} - \rho(n) = 363.44 \times n^{2.54}. \tag{13.104}$$

We see that the dimension of pressure is the same as that of the mass-energy density, i.e., $[P]$ =MeV/c^{2}fm^{-3}.

Here comes an important point you should observe when solving the two coupled first-order differential equations. When you obtain the new pressure given by

$$P_{new} = \frac{dP}{dr} + P_{old}, \tag{13.105}$$

this comes as a function of r. However, having obtained the new pressure, you will need to use Eq. (13.104) in order to find the number density n. This will in turn allow you to find the new value of the mass-energy density $\rho(n)$ at the relevant value of r.

In solving the differential equations for neutron star equilibrium, you should proceed as follows

1. Make first a dimensional analysis in order to be sure that all equations are really dimensionless.

2. Define the constants R_0 and M_0 in units of 10 km and solar mass $M_\odot$. Find their values. Explain why it is convenient to insert these constants in the final results and not at each intermediate step.

3. Set up the algorithm for solving these equations and write a main program where the various variables are defined.

4. Write thereafter a small function which uses the expressions for pressure and mass-energy density from Eqs. (13.104) and (13.102).

5. Write then a function which sets up the derivatives

$$-\frac{\hat{m}\hat{\rho}}{\hat{r}^2}, \qquad \hat{r}^2\hat{\rho}. \tag{13.106}$$

6. Employ now the fourth order Runge-Kutta algorithm to obtain new values for the pressure and the mass. Play around with different values for the step size and compare the results for mass and radius.

7. Replace the fourth order Runge-Kutta method with the simple Euler method and compare the results.

8. Replace the non-relativistic expression for the derivative of the pressure with that from General Relativity (GR), the so-called Tolman-Oppenheimer-Volkov equation

$$\frac{d\hat{P}}{d\hat{r}} = -\frac{(\hat{P}+\hat{\rho})(\hat{r}^3\hat{P}+\hat{m})}{\hat{r}^2 - 2\hat{m}\hat{r}}, \tag{13.107}$$

and solve again the two differential equations.

9. Compare the non-relatistic and the GR results by plotting mass and radius as functions of the central density.

Project 13.2: studies of white dwarf stars

This project contains a long description of the physics of compact objects such as white dwarfs. It serves as a background for understanding the final differential equations you need to solve. This project is taken from the text of Koonin and Meredith [4].

White dwarfs are cold objects which consist mainly of heavy nuclei such as ^{56}Fe, with 26 protons, 30 neutrons and their respective electrons, see for example Ref. [71]. Charge equilibrium results in an equal quantity of electrons and protons. You can read more about white dwarfs, neutron stars and black holes at the website of the Joint Institute for Nuclear Astrophysics www.jinaweb.org or NASA's website www.nasa.org. These stars are the endpoints of stars with masses of the size or smaller than our sun. They are the outcome of standard nuclear processes and end their lives as cold objects like white dwarfs when they have used up all their nuclear fuel.

Where a star ends up at the end of its life depends on the mass, or amount of matter, it was born with. Stars that have a lot of mass may end their lives as black holes or neutron stars. Low and medium mass stars will become something called a white dwarf. A typical white dwarf is half as massive as the Sun, yet only slightly bigger than the Earth. This makes white dwarfs one of the densest forms of matter, surpassed only by neutron stars.

Medium mass stars, like our Sun, live by burning the hydrogen that dwells within their cores, turning it into helium. This is what our Sun is doing now. The heat the Sun generates by its nuclear fusion of hydrogen into helium creates an outward pressure. In another 5 billion years, the Sun will have used up all the hydrogen in its core.

This situation in a star is similar to a pressure cooker. Heating something in a sealed container causes a build up in pressure. The same thing happens in the Sun. Although the Sun may not strictly be a sealed container, gravity causes it to act like one, pulling the star inward, while the pressure created by the hot gas in the core pushes to get out. The balance between pressure and gravity is very delicate.

Because a white dwarf is no longer able to create internal pressure, gravity unopposedly crushes it down until even the very electrons that make up a white dwarf's atoms are mashed together. In normal circumstances, identical electrons (those with the same "spin") are not allowed to occupy the same energy level. Since there are only two ways an electron can spin, only two electrons can occupy a single energy level. This is what's know in physics as the Pauli Exclusion Principle. And in a normal gas, this isn't a problem; there aren't enough electrons floating around to completely fill up all the energy levels. But in a white dwarf, all of its electrons are forced close together; soon all the energy levels in its atoms are filled up with electrons. If all the energy levels are filled, and it is impossible to put more than two electrons in each level, then our white dwarf has become degenerate. For gravity to compress the white dwarf anymore, it must force electrons where they cannot go. Once a star is degenerate, gravity cannot compress it any more because quantum mechanics tells us there is no more available space to be taken up. So our white dwarf survives, not by internal combustion, but by quantum mechanical principles that prevent its complete collapse.

With a surface gravity of 100,000 times that of the earth, the atmosphere of a white dwarf is very strange. The heavier atoms in its atmosphere sink and the lighter ones remain at the surface. Some white dwarfs have almost pure hydrogen or helium atmospheres, the lightest of elements. Also, the very strong gravity pulls the atmosphere close around it in a very thin layer, that, if were it on earth, would be lower than the tops of our skyscrapers!

13.8.1 Equilibrium equations

We assume that the star is in thermal equilibrium. It exhibits also charge equilibrium, meaning the number of electrons has to balance the number of protons. The gravitational pull on every element of volume is balanced by the pressure set up by a degenerate gas of electrons at $T = 0$, since the temperature of the star is well below the so-called Fermi temperature of the electrons. The electrons are assumed to be relativistic and since the protons and neutrons have much lower kinetic energy, we assume that the pressure which balances the gravitational force is mainly set up by the relativistic electrons. The kinetic energy of the electrons is also much larger than the electron-electron repulsion or the attraction from the nuclei. This means that we can treat the system as a gas of free degenerate electrons at $T = 0$ moving in between a lattice of nuclei like iron. This is our ansatz. Based on this we can derive the pressure which counterbalances the gravitational force given by (for every element of volume in a distance r from the center of the star)

$$F_{Grav} = -\frac{Gm(r)}{r^2}\rho(r),$$

with G being the gravitational constant, $\rho(r)$ the mass density (mass per volume) of a volume element a distance r from the center of the star, and $m(r)$ is the integrated mass within a radius r. The latter reads

$$m(r) = 4\pi \int_0^r \rho(r')r'^2 dr'$$

which yields a differential equation between the total mass and the mass density

$$\frac{dm}{dr} = 4\pi r^2\rho(r).$$

In equilibrium, the pressure P balances the gravitational force

$$\frac{dP}{dr} = -\frac{Gm(r)}{r^2}\rho(r),$$

and using $dP/d\rho = (d\rho/dr)(dP/d\rho)$ we obtain

$$\frac{d\rho}{dr} = -\left(\frac{dP}{d\rho}\right)^{-1}\frac{Gm}{r^2}\rho.$$

Together with $\frac{dm}{dr} = 4\pi r^2\rho(r)$ we have now two coupled first-order ordinary differential equations which determine the structure of the white dwarf given an equation of state $P(\rho)$. The total radius is given by the condition $\rho(R) = 0$. Similarly, the mass for $r = 0$ is $m = 0$. The density at $r = 0$ is given by the central density ρ_c, a parameter you will have to play with as input parameter.

By integrating the last equation, we find the density profile of the star. The radius R is determined by the point where the density distribution is $\rho = 0$. The mass is then given by $M = m(R)$. Since both the total mass and the radius R will depend on the central density ρ_c, a variation of this parameter will allow us to study stars with different masses. However, before we can proceed, we need the pressure for a relativistic gas of electrons.

Equation of state for a white dwarf

We will treat the electrons as a relativistic gas of fermions at $T = 0$. From statistical physics we can then obtain the particle density as

$$n = N/V = \frac{1}{\pi^2}\int_0^{k_F} k^2 dk = \frac{k_F^3}{3\pi^2},$$

where k_F is the Fermi momentum, here represented by the wave number k_F. The wave number is connected to the momentum via $k_F = p_F/\hbar$. The energy density is given by

$$\varepsilon = E/V = \frac{1}{\pi^2}\int_0^{k_F} k^2 dk\sqrt{(\hbar ck)^2 + m_e^2c^4}.$$

This expression is of the form $\int y^2\sqrt{y^2+a^2}$. Performing the integration we obtain

$$E/V = n_0 m_e c^2 x^3 \epsilon(x),$$

where we have defined

$$\epsilon(x) = \frac{3}{8x^3}\left(x(1+2x^2)\sqrt{1+x^2} - ln(x+\sqrt{1+x^2})\right),$$

with the variable x defined as

$$x = \frac{\hbar k_F}{m_e c}.$$

We can rewrite x in terms of the particle density as well

$$n = N/V = \frac{k_F^3}{3\pi^2},$$

so that

$$\frac{\hbar k_F}{m_e c} = \left(\frac{n\hbar 3\pi^2}{m_e^3 c^3}\right)^{1/3},$$

where we define $n_0 = \frac{(mc)_e^3}{3\pi^2(\hbar)^3}$ with m_e the electron mass. Using the constant n_0 results finally in

$$x = \frac{\hbar k_F}{m_e c} = \left(\frac{n}{n_0}\right)^{1/3}.$$

Since the mass of the protons and neutrons are larger by a factor 10^3 than the mass of the electrons m_e, we can approximate the total mass of the star by the mass density of the nucleons (protons and neutrons). This mass density is given by

$$\rho = M_p n_p,$$

with M_p being the mass of the proton and n_p the particle density of the nucleons. The mass of the proton and the neutron are almost equal and we have set them equal here. The particle density n_p can be related to the electron density n, which is the quantity we can calculate. The relation is simple,

$$n_p = n/Y_e,$$

where Y_e is the number of electrons per nucleon. For ^{56}Fe we get $Y_e = \frac{26}{56} = 0.464$, since we need to have as many electrons as protons in order to obtain a total charge of zero. Inserting numerical values for the electron mass we get

$$n_0 = 5.89 \times 10^{29}\mathrm{cm}^{-3}.$$

The mass density is now

$$\rho = M_p n/Y_e,$$

and with

$$x = \left(\frac{n}{n_0}\right)^{1/3} = \left(\frac{\rho}{\rho_0}\right)^{1/3},$$

and inserting the numerical value for the proton mass we obtain

$$\rho_0 = \frac{M_p n_0}{Y_e} = 9.79 \times 10^5 Y_e^{-1} \mathrm{g\ cm}^{-3}.$$

Using the parameter Y_e we can then study stars with different compositions. The only input parameters to your code are then ρ_c and Y_e.

Now we want the equation for the pressure, based on the energy density. Using the thermodynamical relation

$$P = -\frac{\partial E}{\partial V} = -\frac{\partial E}{\partial x}\frac{\partial x}{\partial V},$$

we can find the pressure as a function of the mass density ρ. Thereafter we can find $\frac{dP}{d\rho}$, which allows us to determine the mass and the radius of the star.

The term

$$\frac{\partial x}{\partial V},$$

can be found using the fact that $x \propto n^{1/3} \propto V^{-3}$. This results in

$$\frac{\partial x}{\partial V} = -\frac{x}{3V}.$$

Taking the derivative with respect to x we obtain

$$P = \frac{1}{3} n_0 m_e c^2 x^4 \frac{d\epsilon}{dx}.$$

We want the derivative of P in terms of the mass density ρ. Using $x = \left(\frac{\rho}{\rho_0}\right)^{1/3}$, we obtain

$$\frac{dP}{d\rho} = \frac{dP}{dx}\frac{dx}{d\rho}.$$

With

$$\frac{dP}{dx} = \frac{1}{3} n_0 m_e \left(\frac{dx^4 \frac{d\epsilon}{dx}}{dx}\right),$$

and

$$\frac{dx}{d\rho} = \frac{1 \rho_0^{2/3}}{3\rho_0 \rho^{2/3}} = \frac{1}{3\rho_0 x^2},$$

we find

$$\frac{dP}{d\rho} = Y_e \frac{m_e c^2}{M_p} \gamma(x),$$

where we defined

$$\gamma(x) = \frac{x^2}{3\sqrt{1+x^2}}.$$

This is the equation for the derivative of the pressure to be used to find

$$\frac{d\rho}{dr} = -\left(\frac{dP}{d\rho}\right)^{-1} \frac{Gm}{r^2}\rho.$$

Note that x and $\gamma(x)$ are dimensionless quantities.

Dimensionless form of the differential equations

In the numerical treatment of the two differential equations we need to rescale our equations in terms of dimensionless quantities, since several of the involved constants are either extremely large or very small. Furthermore, the total mass is of the order of the mass of the sun, approximately 2×10^{30}kg while the mass of the electron is 9×10^{-31} kg.

We introduce therefore a dimensionless radius $\overline{r} = r/R_0$, a dimensionless density $\overline{\rho} = \rho/\rho_0$ (recall that $x^3 = \rho/\rho_0$) and a dimensionless mass $\overline{m} = m/M_0$.

We determine below the constants M_0 and R_0 by requiring that the equations for $\frac{dm}{dr}$ and $\frac{d\rho}{dr}$ have to be dimensionless. We get then

$$\frac{dM_0\overline{m}}{dR_0\overline{r}} = 4\pi R_0^2\overline{r}^2\rho_0\overline{\rho},$$

resulting in

$$\frac{d\overline{m}}{d\overline{r}} = 4\pi R_0^3\overline{r}^2\rho_0\overline{\rho}/M_0.$$

If we want this equation to be dimensionless, we must require

$$4\pi R_0^3\rho_0/M_0 = 1.$$

Correspondingly, we have

$$\frac{d\rho_0\overline{\rho}}{dR_0\overline{r}} = -\left(\frac{GM_0M_p}{Y_em_ec^2}\right)\frac{\overline{m}}{\gamma R_0^2\overline{r}^2}\rho_0\overline{\rho},$$

with R_0

$$R_0 = \left(\frac{Y_em_ec^2}{4\pi\rho_0 GM_p}\right)^{1/2} = 7.72 \times 10^8 Y_e\text{cm}.$$

in order to yield a dimensionless equation. This results in

$$M_0 = 4\pi R_0^3\rho_0 = 5.67 \times 10^{33} Y_e^2\text{g}.$$

The radius of the sun is $R_\odot = 6.95 \times 10^{10}$ cm and the mass of the sun is $M_\odot = 1.99 \times 10^{33}$ g.

Our final differential equations $\overline{\rho}$ and $\overline{m}$ read

$$\frac{d\overline{\rho}}{d\overline{r}} = -\frac{\overline{m}}{\gamma}\frac{\overline{\rho}}{\overline{r}^2}, \qquad \frac{d\overline{m}}{d\overline{r}} = \overline{r}^2\overline{\rho}.$$

These are the equations you need to code.

a) Verify the steps in the above derivations. Write a program which solves the two coupled differential equations

$$\frac{d\overline{\rho}}{d\overline{r}} = -\frac{\overline{m}}{\gamma}\frac{\overline{\rho}}{\overline{r}^2},$$

and

$$\frac{d\overline{m}}{d\overline{r}} = \overline{r}^2\overline{\rho},$$

using the fourth order Runge-Kutta method by integrating outward from $\overline{r} = 0$. Choose $Y_e = 1$ and calculate the mass and radius of the star by varying the central density $\overline{\rho}_c$ ranging from 10^{-1} to 10^6. Check the stability of your solutions by varying the radial step h. Discuss your results.

b) Compute also the density profiles for the above input parameters and calculate the total kinetic energy and rest energy of the electrons given by

$$U = \int_0^R 4\pi \left(\frac{E}{V}\right) r^2 dr,$$

where we have defined

$$E/V = n_0 m_e c^2 x^3 \epsilon(x),$$

with

$$\epsilon(x) = \frac{3}{8x^3}\left(x(1+2x^2)\sqrt{1+x^2} - ln(x+\sqrt{1+x^2})\right),$$

and the variable x defined as

$$x = \frac{\hbar k_F}{m_e c}.$$

Compute also the gravitational energy

$$W = -\int_0^R \frac{Gm(r)\rho(r)}{r} 4\pi r^2 dr.$$

You need to make these equations dimensionless.

Try to discuss your results and trends through simple physical reasoning.

c) Scale the mass-radius relation you found in a) to the cases corresponding to ^{56}Fe and ^{12}C. Three white dwarf stars, Sirius B, 40 Eri B and Stein 2051, have masses and radii in units of solar values determined from observations to be $(1.053 \pm 0.028 M_\odot, 0.0074 \pm 0.0006 R_\odot)$, $(0.48 \pm 0.02 M_\odot, 0.0124 \pm 0.0005 R_\odot)$, and $(0.72 \pm 0.08 M_\odot, 0.0115 \pm 0.0012 R_\odot)$, respectively. Verify that these values are consistent with the model you have developed. Can you say something about the compositions of these stars?

Project 13.3: Period doubling and chaos

The angular equation of motion of the pendulum is given by Newton's equation and with no external force it reads

$$ml\frac{d^2\theta}{dt^2} + mgsin(\theta) = 0, \tag{13.108}$$

with an angular velocity and acceleration given by

$$v = l\frac{d\theta}{dt}, \tag{13.109}$$

and

$$a = l\frac{d^2\theta}{dt^2}. \tag{13.110}$$

We do however expect that the motion will gradually come to an end due a viscous drag torque acting on the pendulum. In the presence of the drag, the above equation becomes

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = 0, \tag{13.111}$$

where ν is now a positive constant parameterizing the viscosity of the medium in question. In order to maintain the motion against viscosity, it is necessary to add some external driving force. We choose here a periodic driving force. The last equation becomes then

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = Asin(\omega t), \qquad (13.112)$$

with A and ω two constants representing the amplitude and the angular frequency respectively. The latter is called the driving frequency.

a) Rewrite Eqs. (13.111) and (13.112) as dimensionless equations.

b) Write then a code which solves Eq. (13.111) using the fourth-order Runge Kutta method. Perform calculations for at least ten periods with $N = 100$, $N = 1000$ and $N = 10000$ mesh points and values of $\nu = 1$, $\nu = 5$ and $\nu = 10$. Set $l = 1.0$ m, $g = 1$ m/s^2 and $m = 1$ kg. Choose as initial conditions $\theta(0) = 0.2$ (radians) and $v(0) = 0$ (radians/s). Make plots of θ (in radians) as function of time and phase space plots of θ versus the velocity v. Check the stability of your results as functions of time and number of mesh points. Which case corresponds to damped, underdamped and overdamped oscillatory motion? Comment your results.

c) Now we switch to Eq. (13.112) for the rest of the project. Add an external driving force and set $l = g = 1$, $m = 1$, $\nu = 1/2$ and $\omega = 2/3$. Choose as initial conditions $\theta(0) = 0.2$ and $v(0) = 0$ and $A = 0.5$ and $A = 1.2$. Make plots of θ (in radians) as function of time for at least 300 periods and phase space plots of θ versus the velocity v. Choose an appropriate time step. Comment and explain the results for the different values of A.

d) Keep now the constants from the previous exercise fixed but set now $A = 1.35$, $A = 1.44$ and $A = 1.465$. Plot θ (in radians) as function of time for at least 300 periods for these values of A and comment your results.

e) We want to analyse further these results by making phase space plots of θ versus the velocity v using only the points where we have $\omega t = 2n\pi$ where n is an integer. These are normally called the drive periods. This is an example of what is called a Poincare section and is a very useful way to plot and analyze the behavior of a dynamical system. Comment your results.

Chapter 14

Two point boundary value problems

14.1 Introduction

When diffential equations are required to satisfy boundary conditions at more than one value of the independent variable, the resulting problem is called a *boundary value problem*. The most common case by far is when boundary conditions are supposed to be satisfied at two points - usually the starting and ending values of the integration. The Schrödinger equation is an important example of such a case. Here the eigenfunctions are typically restricted to be finite everywhere (in particular at $r = 0$) and for bound states the functions must go to zero at infinity.

In the previous chapter we discussed the solution of differential equations determined by conditions imposed at one point only, the so-called initial condition. Here we move on to differential equations where the solution is required to satisfy conditions at more than one point. Typically these are the endpoints of the interval under consideration. When discussing differential equations with boundary conditions, there are three main groups of numerical methods, shooting methods, finite difference and finite element methods. In this chapter we focus on the so-called shooting method, whereas chapters 12 and 15 focus on finite difference methods. Chapter 12 solves the finite difference problem as an eigenvalue problem for a one variable differential equation while in chapter 15 we present the simplest finite difference methods for solving partial differential equations with more than one variable. The finite element method is not discussed in this text, see for example Ref. [75] for a computational presentation of the finite element method.

In the discussion here we will limit ourselves to the simplest possible case, that of a linear second-order differential equation whose solution is specified at two distinct points, for more complicated systems and equations see for example Refs. [76, 77]. The reader should also note that the techniques discussed in this chapter are restricted to ordinary differential equations only, while finite difference and finite element methods can also be applied to boundary value problems for partial differential equations. The discussion in this chapter and chapter 12 serves therefore as an intermediate step and model to the chapter on partial differential equations. Partial differential equations involve both boundary conditions and differential equations with functions depending on more than one variable.

In this chapter we will discuss in particular the solution of the one-particle Schödinger equation and apply the method to hydrogen-atom like problems. We start however with a familiar problem from mechanics, namely that of a tightly stretched and flexible string or rope, fixed at the endpoints. This problem has an analytic solution which allows us to define our numerical algorithms based on the shooting methods.

14.2 Shooting methods

In many physics applications we encounter differential equations like

$$\frac{d^2y}{dx^2} + k(x)y = F(x);\ a \le x \le b, \tag{14.1}$$

with boundary conditions

$$y(a) = \alpha,\ y(b) = \beta. \tag{14.2}$$

We can interpret $F(x)$ as an inhomogenous driving force while $k(x)$ is a real function. If it is positive the solutions $y(x)$ will be oscillatory functions, and if negative they are exponentionally growing or decaying functions.

To solve this equation we could start with for example the Runge-Kutta method or various improvements to Euler's method, as discussed in the previous chapter. Then we would need to transform this equation to a set of coupled first equations. We could however start with the discretized version for the second derivative. We discretise our equation and introduce a step length $h = (b-a)/N$, with N being the number of equally spaced mesh points. Our discretised second derivative reads at a step $x_i = a + ih$ with $i = 0, 1, \dots$

$$y_i'' = \frac{y_{i+1} + y_{i-1} - 2y_i}{h^2} + O(h^2),$$

leading to a discretised differential equation

$$\frac{y_{i+1} + y_{i-1} - 2y_i}{h^2} + O(h^2) + k_i y_i = F_i. \tag{14.3}$$

Recall that the fourth-order Runge-Kutta method has a local error of $O(h^4)$.

Since we want to integrate our equation from $x_0 = a$ to $x_N = b$, we rewrite it as

$$y_{i+1} \approx -y_{i-1} + y_i \left(2 - h^2 k_i + h^2 F_i\right). \tag{14.4}$$

Starting at $i = 1$ we have after one step

$$y_2 \approx -y_0 + y_1 \left(2 - h^2 k_1 + h^2 F_1\right).$$

Irrespective of method to approximate the second derivative, this equation uncovers our first problem. While $y_0 = y(a) = 0$, our function value y_1 is unknown, unless we have an analytic expression for $y(x)$ at $x = 0$. Knowing y_1 is equivalent to knowing y' at $x = 0$ since the first derivative is given by

$$y_i' \approx \frac{y_{i+1} - y_i}{h}. \tag{14.5}$$

This means that we have $y_1 \approx y_0 + hy_0'$.

14.2.1 Improved approximation to the second derivative, Numerov's method

Before we proceed, we mention how to improve the local truncation error from $O(h^2)$ to $O(h^6)$ without too many additional function evaluations.

Our equation is a second order differential equation without any first order derivatives. Let us also for the sake of simplicity assume that $F(x) = 0$. Numerov's method is designed to solve such an equation numerically, achieving a local truncation error $O(h^6)$.

We start with the Taylor expansion of the desired solution

$$y(x+h) = y(x) + hy^{(1)}(x) + \frac{h^2}{2!}y^{(2)}(x) + \frac{h^3}{3!}y^{(3)}(x) + \frac{h^4}{4!}y^{(4)}(x) + \cdots$$

We have that $y^{(n)}(x)$ is a shorthand notation for the nth derivative d^ny/dx^n. Because the corresponding Taylor expansion of $y(x-h)$ has odd powers of h appearing with negative signs, all odd powers cancel when we add $y(x+h)$ and $y(x-h)$

$$y(x+h) + y(x-h) = 2y(x) + h^2 y^{(2)}(x) + \frac{h^4}{12}y^{(4)}(x) + O(h^6).$$

We obtain

$$y^{(2)}(x) = \frac{y(x+h) + y(x-h) - 2y(x)}{h^2} - \frac{h^2}{12}y^{(4)}(x) + O(h^4).$$

To eliminate the fourth-derivative term we apply the operator $(1 + \frac{h^2}{12}\frac{d^2}{dx^2})$ on the differential equation

$$y^{(2)}(x) + \frac{h^2}{12}y^{(4)}(x) + k^2(x)y(x) + \frac{h^2}{12}\frac{d^2}{dx^2}\left(k^2(x)y(x)\right) \approx 0.$$

In this expression the $y^{(4)}$ terms cancel. To treat the general x dependence of $k^2(x)$ we approximate the second derivative of $(k^2(x)y(x)$ by

$$\frac{d^2(k^2y(x))}{dx^2} \approx \frac{\left(k^2(x+h)y(x+h) + k^2(x)y(x)\right) + \left(k^2(x-h)y(x-h) + k^2(x)y(x)\right)}{h^2}.$$

We replace then $y(x+h)$ with the shorthand y_{i+1} (and similarly for the other variables) and obtain a final discretised algorithm for obtaining y_{i+1}

$$y_{i+1} = \frac{2\left(1 - \frac{5}{12}h^2k_i^2\right)y_i - \left(1 + \frac{1}{12}h^2k_{i-1}^2\right)y_{i-1}}{1 + \frac{h^2}{12}k_{i+1}^2} + O(h^6),$$

where $x_i = ih$, $k_i = k(x_i = ih)$ and $y_i = y(x_i = ih)$ etc.

It is easy to add the term F_i since we need only to take the second derivative. The final algorithm reads then

$$y_{i+1} = \frac{2\left(1 - \frac{5}{12}h^2k_i^2\right)y_i - \left(1 + \frac{1}{12}h^2k_{i-1}^2\right)y_{i-1}}{1 + \frac{h^2}{12}k_{i+1}^2} + \frac{h^2}{12}\left(F_{i+1} + F_{i-1} - 2F_i\right) + O(h^6).$$

Starting at $i = 1$ results in, using the boundary condition $y_0 = 0$,

$$y_2 = \frac{2\left(1 - \frac{5}{12}h^2k_1y_1\right) - \left(1 + \frac{1}{12}h^2k_0y_0\right)}{1 + \frac{h^2}{12}k_2} + \frac{h^2}{12}\left(F_2 + F_0 - 2F_1\right) + O(h^6).$$

This equation carries a local truncation error proportional to h^6. This is an order better than the fourth-order Runge-Kutta method which has a local error proportional to h^5. The global for the fourth-order Runge-Kutta is proportional to h^4 while Numerov's method has an error proportional to h^5. With few additional function evulations, we have achieved an increased accuracy.

But even with an improved accuracy we end up with one unknown on the right hand side, namely y_1. The value of y_1 can again be determined from the derivative at y_0, or by a good guess on its value. We need therefore an additional constraint on our set of equations before we start. We could then add to the boundary conditions

$$y(a) = \alpha,\ y(b) = \beta,$$

the requirement $y'(a) = \delta$, where δ could be an arbitrary constant. In quantum mechanical applications with homogenous differential equations the normalization of the solution is normally not known. The choice of the constant δ can therefore reflect specific symmetry requirements of the solution.

14.2.2 Wave equation with constant acceleration

We start with a well-known problem from mechanics, that of a whirling string or rope fixed at both ends. We could think of this as an idealization of a jumping rope and ask questions about its shape as it spins. Obviously, in deriving the equations we will make several assumptions in order to obtain an analytic solution. However, the general differential equation it leads to, with added complications not allowing an analytic solution, can be solved numerically. We discuss the shooting methods as one possible numerical approach in the next section.

Our aim is to arrive at a differential equation which takes the following form

$$y'' + \lambda y = 0;\ y(0) = 0,\ y(L) = 0,$$

where L is the length of the string and λ a constant or function of the variable x to be defined below.

We derive an equation for $y(x)$ using Newton's second law $F = ma$ acting on a piece of the string with mass $\rho\Delta x$, where ρ is the mass density per unit length and Δx is small displacement in the interval $x, x+\Delta x$. The change Δx is our step length.

We assume that the only force acting on this string element is a constant tension T acting on both ends. The net vertical force in the positive y-direction is

$$F = Tsin(\theta + \Delta\theta) - Tsin(\theta) = Tsin(\theta_{i+1}) - Tsin(\theta_i).$$

For the angles we employ a finite difference approximation

$$sin(\theta_{i+1}) = \frac{y_{i+1} - y_i}{\Delta x} + O(\Delta x^2).$$

Using Newton's second law $F = ma$, with $m = \rho\Delta x = \rho h$ and a constant angular velocity ω which relates to the acceleration as $a = -\omega^2 y$ we arrive at

$$T\frac{y_{i+1} + y_{i-1} - 2y_i}{\Delta x^2} \approx -\rho\omega^2 y,$$

and taking the limit $\Delta x \to 0$ we can rewrite the last equation as

$$Ty'' + \rho\omega^2 y = 0,$$

and defining $\lambda = \rho\omega^2/T$ and imposing the condition that the ends of the string are fixed we arrive at our final second-order differential equation with boundary conditions

$$y'' + \lambda y = 0;\ y(0) = 0,\ y(L) = 0. \tag{14.6}$$

The reader should note that we have assumed a constant acceleration. Replacing the constant acceleration with the second derivative of y as function of both position and time, we arrive at the well-known wave equation for $y(x,t)$ in $1+1$ dimension, namely

$$\frac{\partial^2 y}{\partial t^2} = \lambda\frac{\partial^2 y}{\partial x^2}.$$

We discuss the solution of this equation in chapter 15.

If $\lambda > 0$ this equation has a solution of the form

$$y(x) = Acos(\alpha x) + Bsin(\alpha x), \tag{14.7}$$

and imposing the boundary conditions results in an infinite sequence of solutions of the form

$$y_n(x) = sin(\frac{n\pi x}{L}),\ n = 1, 2, 3, \ldots \tag{14.8}$$

with eigenvalues

$$\lambda_n = \frac{n^2\pi^2}{L^2},\ n = 1, 2, 3, \ldots \tag{14.9}$$

For $\lambda = 0$ we have

$$y(x) = Ax + B, \tag{14.10}$$

and due to the boundary conditions we have $y(x) = 0$, the trivial solution, which is not an eigenvalue of the problem. The classical problem has no negative eigenvalues, viz we cannot find a solution for $\lambda < 0$. The trivial solution means that the string remains in its equilibrium position with no deflection.

If we relate the constant angular speed ω to the eigenvalues λ_n we have

$$\omega_n = \sqrt{\frac{\lambda_n T}{\rho}} = \frac{n\pi}{L}\sqrt{\frac{T}{\rho}},\ n = 1, 2, 3, \ldots, \tag{14.11}$$

resulting in a series of discretised critical speeds of angular rotation. Only at these critical speeds can the string change from its equilibrium position.

There is one important observation to made here, since later we will discuss Schrödinger's equation. We observe that the eigenvalues and solutions exist only for certain discretised values $\lambda_n, y_n(x)$. This is a consequence of the fact that we have imposed boundary conditions. Thus, the boundary conditions, which are a consequence of the physical case we wish to explore, yield only a set of possible solutions. In quantum physics, we would say that the eigenvalues λ_n are quantized, which is just another word for discretised eigenvalues.

We have then an analytic solution

$$y_n(x) = sin(\frac{n\pi x}{L}),$$

from

$$y'' + \frac{n^2\pi^2}{L^2}y = 0;\ y(0) = 0,\ y(1) = 0.$$

Choosing $n = 4$ and $L = 1$, we have $y(x) = sin(4\pi x)$ as our solution. The derivative is obviously $4\pi cos(\pi x)$. We can start to integrate our equation using the exact expression for the derivative at y_1. This yields

$$y_2 \approx -y_0 + y_1\left(2 - h^2k_1 + h\right) = 4h\pi cos(4\pi x_0)\left(2 - 16h^2\pi^2\right) = 4\pi\left(2 - 16h^2\pi^2\right).$$

If we split our interval $x \in [0, 1]$ into 10 equally spaced points we arrive at the results displayed in Table 14.1. We note that the error at the endpoint is much larger than the chosen mathematical approximation $O(h^2)$, resulting in an error of approximately 0.01. We would have expected a smaller error. We can obviously get better precision by increasing the number of integration points, but it would not cure the

Table 14.1: Integrated and exact solution of the differential equation $y'' + \lambda y = 0$ with boundary conditions $y(0) = 0$ and $y(1) = 0$.

$x_i = ih$	$sin(\pi x_i)$	$y(x_i)$
0.000000E+00	0.000000E+00	0.000000E+00
0.100000E+00	0.951057E+00	0.125664E+01
0.200000E+00	0.587785E+00	0.528872E+00
0.300000E+00	-.587785E+00	-.103405E+01
0.400000E+00	-.951056E+00	-.964068E+00
0.500000E+00	0.268472E-06	0.628314E+00
0.600000E+00	0.951057E+00	0.122850E+01
0.700000E+00	0.587785E+00	-.111283E+00
0.800000E+00	-.587786E+00	-.127534E+01
0.900000E+00	-.951056E+00	-.425460E+00
0.100000E+01	0.000000E+00	0.109628E+01

increasing discrepancy we see towards the endpoints. With $N = 100$, we have $0.829944E - 02$ at $x = 1.0$, while the error is $\sim 10^{-4}$ with 100 integration points.

It is also important to notice that in general we do not know the eigenvalue and the eigenfunctions, except some of their limiting behaviors close to the boundaries. One method for searching for these eigenvalues is to set up an iterative process. We guess a trial eigenvalue and generate a solution by integrating the differential equation as an initial value problem, as we did above except that we have here the exact solution. If the resulting solution does not satisfy the boundary conditions, we change the trial eigenvalue and integrate again. We repeat this process until a trial eigenvalue satisfies the boundary conditions to within a chosen numerical error. This approach is what constitutes the so-called shooting method.

Upon integrating to our other boundary, $x = 1$ in the above example, we obtain normally a non-vanishing value for $y(1)$, since the trial eigenvalue is normally not the correct one. We can then readjust the guess for the eigenvalue and integrate and repeat this process till we obtain a value for $y(1)$ which agrees to within the precision we have chosen. As we will show in the next section, this results in a root-finding problem, which can be solved with for example the bisection or Newton methods discussed in chapter 5.

The example we studied here hides however an important problem. Our two solutions are rather similar, they are either represented by a $sin(x)$ form or a $cos(x)$ solution. This means that the solutions do not differ dramatically in behavior at the boundaries. Furthermore, the wave function is zero beyond the boundaries. For a quantum mechanical system, we would get the same solutions if a particle is trapped in an infinitely high potential well. Then the wave function cannot exist outside the potential. However, for a finite potential well, there is always a quantum mechanical probability that the particle can be found outside the classical region. The classical region defines the so-called turning points, viz points from where a classical solution cannot exist. These turning points are useful when we want to solve quantum mechanical problems.

Let us however perform our brute force integration for another differential equation as well, namely that of the quantum mechanical harmonic oscillator.

The situation worsens dramatically now. We have then a one-dimensional differential equation of the

Table 14.2: Integrated and exact solution of the differential equation $-y'' + x^2 y = 2\epsilon y$ with boundary conditions $y(-\infty) = 0$ and $y(\infty) = 0$.

$x_i = ih$	$\exp(-x^2/2)$	$y(x_i)$
-.100000E+02	0.192875E-21	0.192875E-21
-.800000E+01	0.126642E-13	0.137620E-13
-.600000E+01	0.152300E-07	0.157352E-07
-.400000E+01	0.335462E-03	0.331824E-03
-.200000E+01	0.135335E+00	0.128549E+00
0.000000E-00	0.100000E+01	0.912665E+00
0.200000E+01	0.135335E+00	0.118573E+00
0.400000E+01	0.335463E-03	-.165045E-01
0.600000E+01	0.152300E-07	-.250865E+03
0.800000E+01	0.126642E-13	-.231385E+09
0.900000E+01	0.257677E-17	-.101904E+13

type, see Eq. (11.21), (all physical costants are set equal to one, that is $m = c = \hbar = k = 1$)

$$-\frac{1}{2}\frac{d^2y}{dx^2} + \frac{1}{2}x^2 y = \epsilon y; \; -\infty < x < \infty, \tag{14.12}$$

with boundary conditions $y(-\infty) = y(\infty) = 0$. For the lowest lying state, the eigenvalue is $\epsilon = 1/2$ and the eigenfunction is

$$y(x) = \left(\frac{1}{\pi}\right)^{1/4} \exp(-x^2/2).$$

The reader should observe that this solution is imposed by the boundary conditions, which again follow from the quantum mechanical properties we require for the solution. We repeat the integration exercise which we did for the previous example, starting from a large negative number ($x_0 = -10$, which gives a value for the eigenfunction close to zero) and choose the lowest energy and its corresponding eigenfunction. We obtain for y_2

$$y_2 \approx -y_0 + y_1\left(2 + h^2x^2 - h^2\right),$$

and using the exact eigenfunction we can replace y_1 with the derivative at x_0. We use now $N = 1000$ and integrate our equation from $x_0 = -10$ to $x_N = 10$. The results are shown in Table 14.2 for selected values of x_i. In the beginning of our integrational interval, we obtain an integrated quantity which resembles the analytic solution, but then our integrated solution simply explodes and diverges. What is happening? We started with the exact solution for both the eigenvalue and the eigenfunction!

The problem is due to the fact that our differential equation has two possible solution for eigenvalues which are very close ($-1/2$ and $+1/2$), either

$$y(x) \sim \exp(-x^2/2),$$

or

$$y(x) \sim \exp(x^2/2).$$

The boundary conditions, imposed by our physics requirements, rule out the last possibility. However, our algorithm, which is nothing but an approximation to the differential equation we have chosen, picks up democratically both solutions. Thus, although we start with the correct solution, when integrating we pick up the undesired solution. In the next subsections we discuss how to cure this problem.

14.2.3 Schrödinger equation for spherical potentials

We discuss the numerical solution of the Schrödinger equation for the case of a particle with mass m moving in a spherical symmetric potential.

The initial eigenvalue equation reads

$$\widehat{\mathbf{H}}\psi(\vec{r}) = (\widehat{\mathbf{T}} + \widehat{\mathbf{V}})\psi(\vec{r}) = E\psi(\vec{r}). \tag{14.13}$$

In detail this gives

$$\left(-\frac{\hbar^2}{2m}\nabla^2 + V(r)\right)\psi(\vec{r}) = E\psi(\vec{r}). \tag{14.14}$$

The eigenfunction in spherical coordinates takes the form

$$\psi(\vec{r}) = R(r)Y_l^m(\theta, \phi), \tag{14.15}$$

and the radial part $R(r)$ is a solution to

$$-\frac{\hbar^2}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr} - \frac{l(l+1)}{r^2}\right)R(r) + V(r)R(r) = ER(r). \tag{14.16}$$

Then we substitute $R(r) = (1/r)u(r)$ and obtain

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r) + \left(V(r) + \frac{l(l+1)}{r^2}\frac{\hbar^2}{2m}\right)u(r) = Eu(r). \tag{14.17}$$

We introduce a dimensionless variable $\rho = (1/\alpha)r$ where α is a constant with dimension length and get

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(r) + \left(V(\rho) + \frac{l(l+1)}{\rho^2}\frac{\hbar^2}{2m\alpha^2}\right)u(\rho) = Eu(\rho). \tag{14.18}$$

In our case we are interested in attractive potentials

$$V(r) = -V_0 f(r), \tag{14.19}$$

where $V_0 > 0$ and analyze bound states where $E < 0$. The final equation can be written as

$$\frac{d^2}{d\rho^2}u(\rho) + k(\rho)u(\rho) = 0, \tag{14.20}$$

where

$$\begin{aligned} k(\rho) &= \gamma\left(f(\rho) - \frac{1}{\gamma}\frac{l(l+1)}{\rho^2} - \epsilon\right) \\ \gamma &= \frac{2m\alpha^2 V_0}{\hbar^2} \\ \epsilon &= \frac{|E|}{V_0} \end{aligned} \tag{14.21}$$

Schrödinger equation for a spherical box potential

Let us now specify the spherical symmetric potential to

$$f(r) = \begin{cases} 1 \\ -0 \end{cases} \quad \text{for} \quad \begin{matrix} r \le a \\ r > a \end{matrix} \tag{14.22}$$

and choose $\alpha = a$. Then

$$k(\rho) = \gamma \begin{cases} 1 - \epsilon - \frac{1}{\gamma}\frac{l(l+1)}{\rho^2} \\ -\epsilon - -\frac{1}{\gamma}\frac{l(l+1)}{\rho^2} \end{cases} \quad \text{for} \quad \begin{matrix} r \le a \\ r > a \end{matrix} \tag{14.23}$$

The eigenfunctions in Eq. (14.14) are subject to conditions which limit the possible solutions. Of importance for the present example is that $u(\vec{r})$ must be finite everywhere and $\int |u(\vec{r})|^2 d\tau$ must be finite. The last condition means that $rR(r) \longrightarrow 0$ for $r \longrightarrow \infty$. These conditions imply that $u(r)$ must be finite at $r = 0$ and $u(r) \longrightarrow 0$ for $r \longrightarrow \infty$.

Analysis of $u(\rho)$ at $\rho = 0$

For small ρ Eq. (14.20) reduces to

$$\frac{d^2}{d\rho^2}u(\rho) - \frac{l(l+1)}{\rho^2}u(\rho) = 0, \tag{14.24}$$

with solutions $u(\rho) = \rho^{l+1}$ or $u(\rho) = \rho^{-l}$. Since the final solution must be finite everywhere we get the condition for our numerical solution

$$u(\rho) = \rho^{l+1} \quad \text{for small } \rho \tag{14.25}$$

Analysis of $u(\rho)$ for $\rho \longrightarrow \infty$

For large ρ Eq. (14.20) reduces to

$$\frac{d^2}{d\rho^2}u(\rho) - \gamma\epsilon u(\rho) = 0 \quad \gamma > 0, \tag{14.26}$$

with solutions $u(\rho) = \exp(\pm\gamma\epsilon\rho)$ and the condition for large ρ means that our numerical solution must satisfy

$$u(\rho) = e^{-\gamma\epsilon\rho} \quad \text{for large } \rho \tag{14.27}$$

As for the harmonic oscillator, we have two solutions at the boundaries which are very different and can easily lead to totally worng and even diverging solutions if we just integrate from one endpoint to the other. In the next section we discuss how to solve such problems.

14.3 Numerical procedure, shooting and matching

The eigenvalue problem in Eq. (14.20) can be solved by the so-called shooting methods. In order to find a bound state we start integrating, with a trial negative value for the energy, from small values of the variable ρ, usually zero, and up to some large value of ρ. As long as the potential is significantly different from zero the function oscillates. Outside the range of the potential the function will approach an exponential form. If we have chosen a correct eigenvalue the function decreases exponentially as $u(\rho) = e^{-\gamma\epsilon\rho}$. However, due to numerical inaccuracy the solution will contain small admixtures of the undesireable exponential growing function $u(\rho) = e^{+\gamma\epsilon\rho}$. The final solution will then become unstable. Therefore, it is better to generate two solutions, with one starting from small values of ρ and integrate outwards to some matching point $\rho = \rho_m$. We call that function $u^<(\rho)$. The next solution $u^>(\rho)$ is then obtained by integrating from some large value ρ where the potential is of no importance, and inwards to the same matching point ρ_m. Due to the quantum mechanical requirements the logarithmic derivative at the matching point ρ_m should be well defined. We obtain the following condition

$$\frac{\frac{d}{d\rho}u^<(\rho)}{u^<(\rho)} = \frac{\frac{d}{d\rho}u^>(\rho)}{u^>(\rho)} \quad \text{at} \quad \rho = \rho_m. \tag{14.28}$$

We can modify this expression by normalizing the function $u^< u^<(\rho_m) = C u^> u^<(\rho_m)$. Then Eq. (14.28) becomes

$$\frac{d}{d\rho}u^<(\rho) = \frac{d}{d\rho}u^>(\rho) \quad \text{at} \quad \rho = \rho_m \tag{14.29}$$

For an arbitary value of the eigenvalue Eq. (14.28) will not be satisfied. Thus the numerical procedure will be to iterate for different eigenvalues until Eq. (14.29) is satisfied.

We can calculate the first order derivatives by

$$\begin{aligned} \frac{d}{d\rho}u^<(\rho_m) &\approx \frac{u^<(\rho_m) - u^<(\rho_m - h)}{h} \\ \frac{d}{d\rho}u^>(\rho_m) &\approx \frac{u^>(\rho_m) - u^>(\rho_m + h)}{h} \end{aligned} \tag{14.30}$$

Thus the criterium for a proper eigenfunction will be

$$f = u^>(\rho_m + h) - u^<(\rho_m - h) \tag{14.31}$$

14.3.1 Algorithm for solving Schrödinger's equation

Here we outline the solution of Schrödinger's equation as a common differential equation but with boundary conditions. The method combines shooting and matching. The shooting part involves a guess on the exact eigenvalue. This trial value is then combined with a standard method for root searching, e.g., the secant or bisection methods discussed in chapter 5.

The algorithm could then take the following form

– Initialise the problem by choosing minimum and maximum values for the energy, $E_{\min}$ and $E_{\max}$, the maximum number of iterations $\mathrm{max\_iter}$ and the desired numerical precision.

– Search then for the roots of the function f, where the root(s) is(are) in the interval $E \in [E_{\min}, E_{\max}]$ using for example the bisection method. Newton's method, also discussed in chapter 5 requires an analytic expression for f. A possible approach is to use the standard bisection method for localizing the eigenvalue and then use the secant method to obtain a better estimate.

The pseudocode for such an approach can be written as

```
do {
     i++;
     e = (e_min+e_max)/2.;     /* bisection   */
     if ( f(e)*f(e_max) > 0 ) {
```

```
            e_max = e;            /* change search interval */
        }
        else {
            e_min = e;
        }
    } while ( (fabs(f(e) > convergence_test) !! (i <= max_iterations))
```

The use of a root-searching method forms the shooting part of the algorithm. We have however not yet specified the matching part.

– The matching part is given by the function $f(e)$ which receives as argument the present value of E. This function forms the core of the method and is based on an integration of Schrödinger's equation from $\rho = 0$ and $\rho = \infty$. If our choice of E satisfies Eq. (14.31) we have a solution. The matching code is given below. To choose the matching point it is convenient to start integrating inwards, that is from the large r-values. When the wave function turns, we use that point to define the matching point. The reason for this is that we start integrating from a region which corresponds normally to classically forbidden ones, and integrating into such regions leads normally to inaccurate solutions and the pick up of the undesired solutions. The consequence is that the solution diverges. We can therefore define as a matching point the classical turning point and start to integrate from large r-values. In the absence of such a point, we can use the point where the wave function turns.

The function $f(E)$ above receives as input a guess for the energy. In the version implemented below, we use the standard three-point formula for the second derivative, namely

$$f_0'' \approx \frac{f_h - 2f_0 + f_{-h}}{h^2}.$$

We leave it as an exercise to the reader to implement Numerov's algorithm.

```
//
// The function
//          f()
// calculates the wave function at fixed energy eigenvalue.
//

void f(double step, int max_step, double energy, double *w, double *wf)
{
    int       loop, loop_1, match;
    double    const sqrt_pi = 1.77245385091;
    double    fac, wwf, norm;
// adding the energy guess to the array containing the potential
    for(loop = 0; loop <= max_step; loop ++) {
       w[loop] = (w[loop] - energy) * step * step + 2;
    }
// integrating from large r-values
    wf[max_step]      = 0.0;
    wf[max_step - 1] = 0.5 * step * step;
// search for matching point
    for(loop = max_step - 2; loop > 0; loop--) {
       wf[loop] = wf[loop + 1] * w[loop + 1] - wf[loop + 2];
       if(wf[loop] <= wf[loop + 1]) break;
    }
```

```
    match = loop + 1;
    wwf   = wf[match];
//  start integrating up to matching point from r =0
    wf[0] = 0.0;
    wf[1] = 0.5 * step * step;
    for(loop = 2; loop <= match; loop++) {
       wf[loop] = wf[loop -1] * w[loop - 1] - wf[loop - 2];
       if(fabs(wf[loop]) > INFINITY) {
          for(loop_1 = 0; loop_1 <= loop; loop_1++) {
             wf[loop_1] /= INFINITY;
          }
       }
    }
// now implement the test of Eq. (10.25)
    return fabs(wf[match-1]-wf[match+1]);
} // End: funtion plot()
```

The approach we have described here suffers from the fact that the matching point is not properly defined. Using a Green's function approach we can easily determine the matching point as the midpoint of the integration interval and compute safely the solution. This is the topic of the next section.

14.4 Green's function approach

A slightly different approach, which however still keeps the matching procedure discussed above, is based on the computation of the Green's function and its relation to the solution of a differential equation with boundary values.

Consider the differential equation

$$-u(x)'' = f(x), \qquad x \in (0,1), \qquad u(0) = u(1) = 0, \tag{14.32}$$

and using the fundamental theorem of calculus, there is a constant c_1 such that

$$u(x) = c_1 + \int_0^x u'(y)dy,$$

and a constant c_2

$$u'(y) = c_2 + \int_0^y u''(z)dz.$$

This is true for any twice continuously differentiable function u

If u satisfies the above differential equation we have then

$$u'(y) = c_2 - \int_0^y f(z)dz.$$

which inserted into the equation for u gives

$$u(x) = c_1 + c_2 x - \int_0^x \left(\int_0^y f(z)dz \right) dy,$$

and defining

$$F(y) = \int_0^y f(z)dz,$$

and performing an integration by parts we obtain

$$\int_0^x F(y)dy = \int_0^x \left(\int_0^y f(z)dz\right) dy = \int_0^x (x-y)f(y)dy.$$

This gives us

$$u(x) = c_1 + c_2 x - \int_0^x (x-y)f(y)dy.$$

The boundary condition $u(0) = 0$ yields $c_1 = 0$ and $u(1) = 0$, resulting in

$$c_2 = \int_0^1 (1-y)f(y)dy,$$

meaning that we can write the solution as

$$u(x) = x\int_0^1 (1-y)f(y)dy - \int_0^x (x-y)f(y)dy$$

The solution to our differential equation can be represented in a compact way using the so-called Green's functions, which are also solutions to our differential equation with $f(x) = 0$. If we then define the Green's function as

$$G(x,y) = \begin{cases} y(1-x) & \text{if } 0 \le y \le x \\ x(1-y) & \text{if } x \le y \le 1 \end{cases}$$

we can write the solution as

$$u(x) = \int_0^1 G(x,y)f(y)dy,$$

The Green's function, see for example Refs. [77, 78] is

1. continuous
2. it is symmetric in the sense that $G(x,y) = G(y,x)$
3. it has the properties $G(0,y) = G(1,y) = G(x,0) = G(x,1) = 0$
4. it is a piecewise linear function of x for fixed y and vice versa. G' is discontinuos at $y = x$.
5. $G(x,y) \ge 0$ for all $x, y \in [0,1]$
6. it is the solution of the differential equation

$$\frac{d^2}{dx^2}G(x,y) = -\delta(x-y).$$

The Green's function can now be used to define the solution before and after a specific matching point in the domain.

The Green's function satisfies the homogeneous equation for $y \neq x$ and its derivative is discontinuous at $x = y$. We can see this if we integrate the differential equation

$$\frac{d^2}{dx^2}G(x,y) = -\delta(x-y)$$

from $x = y - \epsilon$ to $x = y + \epsilon$, with ϵ as an infinitesmally small number. We obtain then

$$\frac{dG}{dx}|_{x=y+\epsilon} - \frac{dG}{dx}|_{x=y-\epsilon} = 1.$$

The problem is obvioulsy to find G.

We can obtain this by considering two solutions of the homogenous equation. We choose a general domain $x \in [a, b]$ with a boundary condition on the general solution $u(a) = u(b) = 0$.

One solution is obtained by integrating from a to b (called $u_<$) and one by integrating inward from b to a, labelled $u_>$.

Using the continuity requirement on the function and its derivative we can compute the Wronskian [77, 78]

$$W = \frac{du_>}{dx}u_< - \frac{du_<}{dx}u_>,$$

and using

$$\frac{dG}{dx}|_{x=y+\epsilon} - \frac{dG}{dx}|_{x=y-\epsilon} = 1,$$

and one can then show that the Green's function reads

$$G(x, y) = u_<(x_<)u_>(x_>), \tag{14.33}$$

where $x_<$ is defined for $x = y - \epsilon$ and $x_> = y + \epsilon$. Using the definition of the Green's function in Eq. (14.33) we can now solve Eq. (14.32) for $x \in [a, b]$ using

$$u(x) = u^>(x)\int_a^x u^<(x')f(x')dx' + u^<(x)\int_x^b u^>(x')f(x')dr' \tag{14.34}$$

The algorithm for solving Eq. (14.32) proceed now as follows: Your task is to choose a matching point, say the midpoint, and then compute the Greens' function after you have used Numerov's algo to find u (inward and outward integration for all points). Find u integrating with the Green's function.

- Compute the solution of the homogeneous part of Eq. (14.32) using Numerov's method. You should then have both the inward and the outward solutions.

- Compute the Wronskian at the matching point using

  $$\frac{du}{dx} \approx \frac{u(x+h) - u(x+h)}{2h},$$

  for the first derivative and choose the matching point as the midpoint. You should try the stability of the solution by choosing other matching points as well.

- Compute then the outward integral of the Green's function approach, including the inhomogeneous term. For the integration one can employ for example Simpson's rule discussed in chapter 7.

- Compute thereafter the inward integral of the Green's function approach. Adding these two integrals gives the resulting wave function of Eq. (14.34).

An example of a code which performs all these steps is listed here

```
void wfn(Array<double,2> &k, Array<double,2> &ubasis, Array<double,1> &r,
    Array<double,2> &F, Array<double,1> &uin, Array<double,1> &uout)
{
  int        loop, loop_1, midpoint, j;
  double     norm, wronskian, sum, term;

  ubasis=0;uin=0;uout=0;

  // Compute inwards homogenous solution
  for(j=0;j<mat_size;j++){

  uin(max_step)       = 0.0;
  uin(max_step-1)     = 1.0E-10;

  for(loop = max_step-2; loop >= 0; loop--) {
    uin(loop) = (2.0*(1.0-5.0*k(loop+1,j)/12.0)* uin(loop+1)- (1.0+k(loop+2,
        j)/12.0)* uin(loop+2))/(1.0+k(loop,j)/12.0);
  }

  // Compute outwards homogenous solution

  uout(0) = 0.0;
  uout(1) = 1.0E-10;

  for(loop = 2; loop <= max_step; loop++) {
    uout(loop) = (2.0*(1.0-5.0*k(loop-1,j)/12.0)* uout(loop-1)- (1.0+k(loop
        -2,j)/12.0)*uout(loop-2))/(1.0+k(loop,j)/12.0);
  }

  // Compute Wronskian at matching mid-point

  midpoint = (max_step)/2;
  // first part of Wronskian
  wronskian = (uin(midpoint+1)-uin(midpoint-1))* uout(midpoint)/(2*step);
  // second part
  wronskian -= (uout(midpoint+1)-uout(midpoint-1))* uin(midpoint)/(2*step);

  // Outward integral of Greens function

  sum = 0.0;
  for(loop = 0; loop <= max_step; loop++) {
    term = uout(loop)*F(loop,j);
    sum += term;
    ubasis(loop,j) = uin(loop)*sum*step;
  }

  // Inward integral of Greens function

  sum = 0.0;
  for(loop = max_step; loop >= 0; loop--) {
    term = uin(loop)*F(loop,j);
```

```
    sum += term;
    ubasis(loop,j) = (ubasis(loop,j)+uout(loop)*sum*step)/wronskian;
  }

  //  Compute the norm

  for(loop = 0, norm = 0.0; loop <= max_step; loop++) {
    norm += ubasis(loop,j)*ubasis(loop,j) * step;//wf[loop] * step;//fabs(wf
        [loop] * step);//wf[loop]* wf[loop] * step;
  }

  if(fabs(norm) < 1.0e-15) {
    printf("\n\nError in norm in function wfn(): ");
    //exit(1);
  }

  norm = 1./sqrt(norm); //

  for(loop = 0; loop <= max_step; loop++) {
    ubasis(loop,j) *= norm;
  }
 }
} // End: funtion wfn()
```

14.5 Projects and exercises

Exercise 14.1: Solution of Poisson's equation with the Green's function method

In this project we will solve the one-dimensional Poissson equation

$$-u''(x) = f(x), \qquad x \in (0,1), \qquad u(0) = u(1) = 0.$$

with the inhomogeneous given by $f(x) = 100e^{-10x}$. This equation has $u(x) = 1-(1-e^{-10})x - e^{-10x}$ as analytic solution.

Write a code which solves the above differential equation using Numerov's algorithm and the Green's function method. Can you find an analytic expression for the Green's function?

Compare these results with those obtained by solving the above differential equation as a set of linear equations, as done in chapter 4. Which method would you prefer?

Project 14.1: Solution of Schrö dinger's equation

We are going to study the solution of the Schrödinger equation for a system with a neutron and a proton (the deuteron) for a simple box potential. This potential will later be replaced with a realistic one fitted to experimental phase shifts.

We begin our discussion of the Schrödinger equation with the neutron-proton (deuteron) system with a box potential $V(r)$. We define the radial part of the wave function $R(r)$ and introduce the definition $u(r) = rR(R)$ The radial part of the SE for two particles in their center-of-mass system and with orbital momentum $l = 0$ is then

$$-\frac{\hbar^2}{2m}\frac{d^2u(r)}{dr^2} + V(r)u(r) = Eu(r), \tag{14.35}$$

with

$$m = 2\frac{m_p m_n}{m_p + m_n}, \tag{14.36}$$

where m_p and m_n are the masses of the proton and neutron, respectively. We use here $m = 938$ MeV. Our potential is defined as

$$V(r) = \begin{cases} 0 & r > a \\ -V_0 & 0 < r \le a \\ \infty & r \le 0 \end{cases}, \tag{14.37}$$

displayed in Fig 14.1.

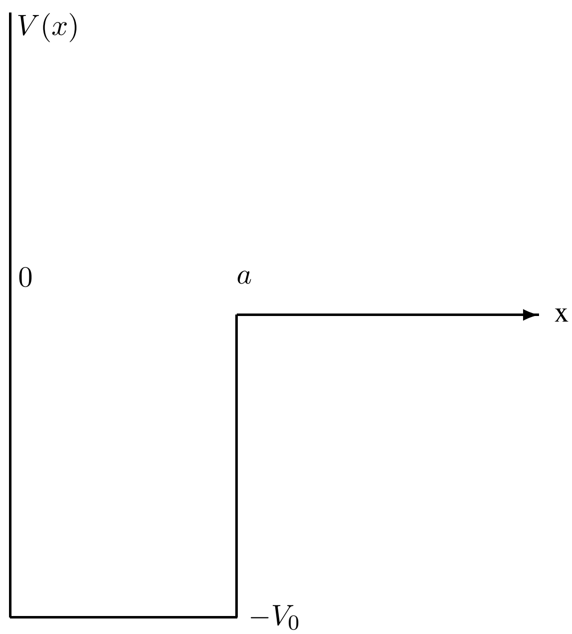


Figure 14.1: Example of a finite box potential with value $-V_0$ in $0 < x \le a$, infinitely large for $x \le 0$ and zero else.

Bound states correspond to negative energy E and scattering states are given by positive energies. The SE takes the form (without specifying the sign of E)

$$\frac{d^2u(r)}{dr^2} + \frac{m}{\hbar^2}(V_0 + E)\, u(r) = 0 \qquad r < a, \tag{14.38}$$

and

$$\frac{d^2u(r)}{dr^2} + \frac{m}{\hbar^2}Eu(r) = 0 \qquad r > a. \tag{14.39}$$

a) We are now going to search for eventual bound states, i.e., $E < 0$. The deuteron has only one bound state at energy $E = -2.223$ MeV. Discuss the boundary conditions on the wave function and use these to show that the solution to the SE is

$$u(r) = Asin(kr) \qquad r < a, \tag{14.40}$$

and

$$u(r) = B\exp\left(-\beta r\right) \qquad r > a, \tag{14.41}$$

where A and B are constants. We have also defined

$$k = \sqrt{m(V_0 - |E|)}/\hbar, \tag{14.42}$$

and

$$\beta = \sqrt{m|E|}/\hbar. \tag{14.43}$$

Show then, using the continuity requirement on the wave function that at $r = a$ you obtain the transcendental equation

$$kcot(ka) = -\beta. \tag{14.44}$$

b) Insert values of $V_0 = 60$ MeV and $a = 1.45$ fm (1 fm = 10^{-15} m) and make a plot of Eq. (14.44) as function of energy E in order to find eventual eigenvalues. See if these values result in a bound state for E.

When you have localized on your plot the point(s) where Eq. (14.44) is satisfied, obtain a numerical value for E using for example Newton-Raphson's method or similar methods, see chapter 5. To use these functions you need to provide the function $kcot(ka) + \beta$ and its derivative as function of E.

What is smallest possible value of V_0 which gives one bound state only?

c) Write a program which implements the Green's function method using Numerov's method for this potential and find the lowest eigenvalue for the case that V_0 supports only one bound state. Use the results from b) to guide your choice of trial eigenvalues. Plot the wave function and discuss your results.

d) We turn now to a fitted interaction which reproduces the low-lying phase shifts for scattering between a proton and neutron. The parametrized version of this potential fits the experimental phase-shifts. It is given by

$$V(r) = V_a\frac{e^{-ax}}{x} + V_b\frac{e^{-bx}}{x} + V_c\frac{e^{-cx}}{x} \tag{14.45}$$

with $x = \mu r$, $\mu = 0.7$ fm^{-1} (the inverse of the pion mass), $V_a = -10.463$ MeV and $a = 1$, $V_b = -1650.6$ MeV and $b = 4$ and $V_c = 6484.3$ MeV and $c = 7$. Replace the box potential from point c) and find the wave function and possible eigenvalues for this potential as well. Discuss your results.

al differential equations, finite difference

Chapter 15

Partial differential equations

15.1 Introduction

In the Natural Sciences we often encounter problems with many variables constrained by boundary conditions and initial values. Many of these problems can be modelled as partial differential equations. One case which arises in many situations is the so-called wave equation whose one-dimensional form reads

$$\frac{\partial^2 u}{\partial x^2} = A\frac{\partial^2 u}{\partial t^2}, \tag{15.1}$$

where A is a constant. The solution u depends on both spatial and temporal variables, viz. $u = u(x,t)$. In two dimension we have $u = u(x,y,t)$. We will, unless otherwise stated, simply use u in our discussion below. Familiar situations which this equation can model are waves on a string, pressure waves, waves on the surface of a fjord or a lake, electromagnetic waves and sound waves to mention a few. For e.g., electromagnetic waves we have the constant $A = c^2$, with c the speed of light. It is rather straightforward to extend this equation to two or three dimension. In two dimensions we have

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = A\frac{\partial^2 u}{\partial t^2}, \tag{15.2}$$

In Chapter 9 we saw another case of a partial differential equation widely used in the Natural Sciences, namely the diffusion equation whose one-dimensional version we derived from a Markovian random walk. It reads

$$\frac{\partial^2 u}{\partial x^2} = A\frac{\partial u}{\partial t}, \tag{15.3}$$

and A is in this case called the diffusion constant. It can be used to model a wide selection of diffusion processes, from molecules to the diffusion of heat in a given material.

Another familiar equation from electrostatics is Laplace's equation, which looks similar to the wave equation in Eq. (15.1) except that we have set $A = 0$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \tag{15.4}$$

or if we have a finite electric charge represented by a charge density $\rho(\mathbf{x})$ we have the familiar Poisson equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -4\pi\rho(\mathbf{x}). \tag{15.5}$$

Other famous partial differential equations are the Helmholtz (or eigenvalue) equation, here specialized to two dimensions only

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = \lambda u, \tag{15.6}$$

the linear transport equation (in $2+1$ dimensions) familiar from Brownian motion as well

$$\frac{\partial u}{\partial x} + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = 0, \tag{15.7}$$

and Schrödinger's equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} + f(x,y)u = \imath\frac{\partial u}{\partial t}.$$

Important systems of linear partial differential equations are the famous Maxwell equations

$$\frac{\partial \mathbf{E}}{\partial t} = \mathrm{curl}\mathbf{B}; \qquad -\mathrm{curl}\mathbf{E} = \mathbf{B}; \qquad \mathrm{div}\mathbf{E} = \mathrm{div}\mathbf{B} = 0.$$

Similarly, famous systems of non-linear partial differential equations are for example Euler's equations for incompressible, inviscid flow

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\nabla\mathbf{u} = -Dp; \qquad \mathrm{div}\mathbf{u} = 0,$$

with p being the pressure and

$$\nabla = \frac{\partial}{\partial x}e_x + \frac{\partial}{\partial y}e_y,$$

in the two dimensions. The unit vectors are e_x and e_y. Another example is the set of Navier-Stokes equations for incompressible, viscous flow

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\nabla\mathbf{u} - \Delta\mathbf{u} = -Dp; \qquad \mathrm{div}\mathbf{u} = 0.$$

Ref. [79] contains a long list of interesting partial differential equations.

In this chapter we focus on so-called finite difference schemes and explicit and implicit methods. The more advanced topic of finite element methods are not treated in this text. For texts with several numerical examples, see for example Refs. [75, 80].

As in the previous chapters we will focus mainly on widely used algorithms for solutions of partial differential equations. A text like Evans' [79] is highly recommended if one wishes to study the mathematical foundations for partial differential equations, in particular how to determine the uniqueness and existence of a solution. We assume namely here that our problems are well-posed, strictly meaning that the problem has a solution, this solution is unique and the solution depends continuously on the data given by the problem. While Evans' text provides a rigorous mathematical exposition, the texts of Langtangen, Ramdas-Mohan, Winther and Tveito and Evans *et al.* contain a more practical algorithmic approach see Refs. [75, 77, 80, 81].

A general partial differential equation in $2+1$-dimensions (with 2 standing for the spatial coordinates x and y and 1 for time) reads

$$A(x,y)\frac{\partial^2 u}{\partial x^2} + B(x,y)\frac{\partial^2 u}{\partial x\partial y} + C(x,y)\frac{\partial^2 u}{\partial y^2} = F(x,y,u,\frac{\partial u}{\partial x},\frac{\partial u}{\partial y}), \tag{15.8}$$

and if we set

$$B = C = 0, \tag{15.9}$$

we recover the $1+1$-dimensional diffusion equation which is an example of a so-called parabolic partial differential equation. With

$$B = 0, \qquad AC < 0 \tag{15.10}$$

we get the $2+1$-dim wave equation which is an example of a so-called elliptic PDE, where more generally we have $B^2 > AC$. For $B^2 < AC$ we obtain a so-called hyperbolic PDE, with the Laplace equation in Eq. (15.4) as one of the classical examples. These equations can all be easily extended to non-linear partial differential equations and $3+1$ dimensional cases.

The aim of this chapter is to present some of the more familiar difference methods and their eventual implementations.

15.2 Diffusion equation

The diffusion equation describes in typical applications the evolution in time of the density u of a quantity like the particle density, energy density, temperature gradient, chemical concentrations etc.

The basis is the assumption that the flux density ρ obeys the Gauss-Green theorem

$$\int_V \mathrm{div}\rho dx = \int_{\partial V} \rho \mathbf{n} dS,$$

where n is the unit outer normal field and V is a smooth region with the space where we seek a solution. The Gauss-Green theorem leads to

$$\mathrm{div}\rho = 0.$$

Assuming that the flux is proportional to the gradient ∇u but pointing in the opposite direction since the flow is from regions of high concetration to lower concentrations, we obtain

$$\rho = -D\nabla u,$$

resulting in

$$\mathrm{div}\nabla u = D\Delta u = 0,$$

which is Laplace's equation, an equation whose one-dimensional version we met in chapter 4. The constant D can be coupled with various physical constants, such as the diffusion constant or the specific heat and thermal conductivity discussed below. We will discuss the solution of the Laplace equation later in this chapter.

If we let u denote the concetration of a particle species, this results in Fick's law of diffusion, see Ref. [52]. If it denotes the temperature gradient, we have Fourier'slaw of heat conduction and if it refers to the electrostatic potential we have Ohm's law of electrical conduction.

Coupling the rate of change (temporal dependence) of u with the flux density we have

$$\frac{\partial u}{\partial t} = -\mathrm{div}\rho,$$

which results in

$$\frac{\partial u}{\partial t} = D\mathrm{div}\nabla u = D\Delta u,$$

the diffusion equation, or heat equation.

If we specialize to the heat equation, we assume that the diffusion of heat through some material is proportional with the temperature gradient $T(\mathbf{x},t)$ and using conservation of energy we arrive at the diffusion equation

$$\frac{\kappa}{C\rho}\nabla^2 T(\mathbf{x},t) = \frac{\partial T(\mathbf{x},t)}{\partial t} \tag{15.11}$$

where C is the specific heat and ρ the density of the material. Here we let the density be represented by a constant, but there is no problem introducing an explicit spatial dependence, viz.,

$$\frac{\kappa}{C\rho(\mathbf{x},t)}\nabla^2 T(\mathbf{x},t) = \frac{\partial T(\mathbf{x},t)}{\partial t}. \tag{15.12}$$

Setting all constants equal to the diffusion constant D, i.e.,

$$D = \frac{C\rho}{\kappa}, \tag{15.13}$$

we arrive at

$$\nabla^2 T(\mathbf{x},t) = D\frac{\partial T(\mathbf{x},t)}{\partial t}. \tag{15.14}$$

Specializing to the $1+1$-dimensional case we have

$$\frac{\partial^2 T(x,t)}{\partial x^2} = D\frac{\partial T(x,t)}{\partial t}. \tag{15.15}$$

We note that the dimension of D is time/length^2. Introducing the dimensional variables $\alpha\hat{x} = x$ we get

$$\frac{\partial^2 T(x,t)}{\alpha^2 \partial \hat{x}^2} = D\frac{\partial T(x,t)}{\partial t}, \tag{15.16}$$

and since α is just a constant we could define $\alpha^2 D = 1$ or use the last expression to define a dimensionless time-variable $\hat{t}$. This yields a simplified diffusion equation

$$\frac{\partial^2 T(\hat{x},\hat{t})}{\partial \hat{x}^2} = \frac{\partial T(\hat{x},\hat{t})}{\partial \hat{t}}. \tag{15.17}$$

It is now a partial differential equation in terms of dimensionless variables. In the discussion below, we will however, for the sake of notational simplicity replace $\hat{x} \to x$ and $\hat{t} \to t$. Moreover, the solution to the $1+1$-dimensional partial differential equation is replaced by $T(\hat{x},\hat{t}) \to u(x,t)$.

15.2.1 Explicit scheme

In one dimension we have the following equation

$$\nabla^2 u(x,t) = \frac{\partial u(x,t)}{\partial t}, \tag{15.18}$$

or

$$u_{xx} = u_t, \tag{15.19}$$

with initial conditions, i.e., the conditions at $t=0$,

$$u(x,0) = g(x) \quad\quad 0 < x < L \tag{15.20}$$

with $L = 1$ the length of the x-region of interest. The boundary conditions are

$$u(0,t) = a(t) \quad t \geq 0, \tag{15.21}$$

and

$$u(L,t) = b(t) \quad t \geq 0, \tag{15.22}$$

where $a(t)$ and $b(t)$ are two functions which depend on time only, while $g(x)$ depends only on the position x. Our next step is to find a numerical algorithm for solving this equation. Here we recur to our familiar equal-step methods discussed in Chapter 3 and introduce different step lengths for the space-variable x and time t through the step length for x

$$\Delta x = \frac{1}{n+1} \tag{15.23}$$

and the time step length Δt. The position after i steps and time at time-step j are now given by

$$\begin{cases} t_j = j\Delta t & j \geq 0 \\ x_i = i\Delta x & 0 \leq i \leq n+1 \end{cases} \tag{15.24}$$

If we then use standard approximations for the derivatives we obtain

$$u_t \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t) - u(x_i,t_j)}{\Delta t} \tag{15.25}$$

with a local approximation error $O(\Delta t)$ and

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2}, \tag{15.26}$$

or

$$u_{xx} \approx \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2}, \tag{15.27}$$

with a local approximation error $O(\Delta x^2)$. Our approximation is to higher order in coordinate space. This can be justified since in most cases it is the spatial dependence which causes numerical problems. These equations can be further simplified as

$$u_t \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t}, \tag{15.28}$$

and

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}. \tag{15.29}$$

The one-dimensional diffusion equation can then be rewritten in its discretized version as

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}. \tag{15.30}$$

Defining $\alpha = \Delta t/\Delta x^2$ results in the explicit scheme

$$u_{i,j+1} = \alpha u_{i-1,j} + (1-2\alpha)u_{i,j} + \alpha u_{i+1,j}. \tag{15.31}$$

Since all the discretized initial values

$$u_{i,0} = g(x_i), \tag{15.32}$$

are known, then after one time-step the only unknown quantity is $u_{i,1}$ which is given by

$$u_{i,1} = \alpha u_{i-1,0} + (1-2\alpha)u_{i,0} + \alpha u_{i+1,0} = \alpha g(x_{i-1}) + (1-2\alpha)g(x_i) + \alpha g(x_{i+1}). \tag{15.33}$$

We can then obtain $u_{i,2}$ using the previously calculated values $u_{i,1}$ and the boundary conditions $a(t)$ and $b(t)$. This algorithm results in a so-called explicit scheme, since the next functions $u_{i,j+1}$ are explicitely given by Eq. (15.31). The procedure is depicted in Fig. 15.1.

We specialize to the case $a(t) = b(t) = 0$ which results in $u_{0,j} = u_{n+1,j} = 0$. We can then reformulate our partial differential equation through the vector V_j at the time $t_j = j\Delta t$

$$V_j = \begin{pmatrix} u_{1,j} \\ u_{2,j} \\ \dots \\ u_{n,j} \end{pmatrix}. \tag{15.34}$$

This results in a matrix-vector multiplication

$$V_{j+1} = \hat{A}V_j \tag{15.35}$$

with the matrix $\hat{A}$ given by

$$\hat{A} = \begin{pmatrix} 1-2\alpha & \alpha & 0 & 0\dots \\ \alpha & 1-2\alpha & \alpha & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & \alpha & 1-2\alpha \end{pmatrix} \tag{15.36}$$

which means we can rewrite the original partial differential equation as a set of matrix-vector multiplications

$$V_{j+1} = \hat{A}V_j = \dots = \hat{A}^{j+1}V_0, \tag{15.37}$$

where V_0 is the initial vector at time $t = 0$ defined by the initial value $g(x)$. In the numerical implementation one should avoid to treat this problem as a matrix vector multiplication since the matrix is triangular and at most three elements in each row are different from zero.

It is rather easy to implement this matrix-vector multiplication as seen in the following piece of code

```
//  First we set initialise the new and old vectors
//  Here we have chosen the boundary conditions to be zero.
//  n+1 is the number of mesh points in x
    u[0] = unew[0] = u[n] = unew = 0.0;
    for (int i = 1; i < n; i++) {
      x =  i*step;
      //  initial condition
      u[i] =  func(x);
      //  intitialise the new vector
      unew[i] = 0;
    }
   // Time iteration
   for (int t = 1; t <= tsteps; t++) {
      for (int i = 1; i < n; i++) {
         // Discretized diff eq
         unew[i] = alpha * u[i-1] + (1 - 2*alpha) * u[i] + alpha * u[i+1];
      }
   //  note that the boundaries are not changed.
```

However, although the explicit scheme is easy to implement, it has a very weak stability condition, given by

$$\Delta t/\Delta x^2 \leq 1/2. \tag{15.38}$$

This means that if $\Delta x^2 = 0.01$, then $\Delta = 5 \times 10^{-5}$. This has obviously bad consequences if our time interval is large. In order to derive this relation we need some results from studies of iterative schemes. If we require that our solution approaches a definite value after a certain amount of time steps we need to require that the so-called spectral radius $\rho(\hat{A})$ of our matrix $\hat{A}$ satisfies the condition

$$\rho(\hat{A}) < 1, \tag{15.39}$$

see for example chapter 10 of Ref. [27] or chapter 4 of [24] for proofs. The spectral radius is defined as

$$\rho(\hat{A}) = \max\left\{|\lambda| : \det(\hat{A} - \lambda\hat{I})\right\}, \tag{15.40}$$

which is interpreted as the smallest number such that a circle with radius centered at zero in the complex plane contains all eigenvalues of $\hat{A}$. If the matrix is positive definite, the condition in Eq. (15.39) is always satisfied.

We can obtain analytic expressions for the eigenvalues of $\hat{A}$. To achieve this it is convenient to rewrite the matrix as

$$\hat{A} = \hat{I} - \alpha\hat{B},$$

with

$$\hat{B} = \begin{pmatrix} 2 & -1 & 0 & 0\dots \\ -1 & 2 & -1 & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & -1 & 2 \end{pmatrix}.$$

The eigenvalues of $\hat{A}$ are $\lambda_i = 1 - \alpha\mu_i$, with μ_i being the eigenvalues of $\hat{B}$. To find μ_i we note that the matrix elements of $\hat{B}$ are

$$b_{ij} = 2\delta_{ij} - \delta_{i+1j} - \delta_{i-1j},$$

meaning that we have the following set of eigenequations for component i

$$(\hat{B}\hat{x})_i = \mu_i x_i, \tag{15.41}$$

resulting in

$$(\hat{B}\hat{x})_i = \sum_{j=1}^{n}\left(2\delta_{ij} - \delta_{i+1j} - \delta_{i-1j}\right)x_j = 2x_i - x_{i+1} - x_{i-1} = \mu_i x_i. \tag{15.42}$$

If we assume that x can be expanded in a basis of $x = (sin(\theta), sin(2\theta), \dots, sin(n\theta))$ with $\theta = l\pi/n+1$, where we have the endpoints given by $x_0 = 0$ and $x_{n+1} = 0$, we can rewrite the last equation as

$$2sin(i\theta) - sin((i+1)\theta) - sin((i-1)\theta) = \mu_i sin(i\theta),$$

or

$$2\left(1 - cos(\theta)\right)sin(i\theta) = \mu_i sin(i\theta),$$

which is nothing but

$$2\left(1 - cos(\theta)\right)x_i = \mu_i x_i,$$

with eigenvalues $\mu_i = 2 - 2cos(\theta)$.

Our requirement in Eq. (15.39) results in

$$-1 < 1 - \alpha 2\left(1 - cos(\theta)\right) < 1,$$

which is satisfied only if $\alpha < \left(1 - cos(\theta)\right)^{-1}$ resulting in $\alpha \leq 1/2$ or $\Delta/\Delta x^2 \leq 1/2$.

15.2.2 Implicit scheme

In deriving the equations for the explicit scheme we started with the so-called forward formula for the first derivative, i.e., we used the discrete approximation

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t}. \tag{15.43}$$

However, there is nothing which hinders us from using the backward formula

$$u_t \approx \frac{u(x_i, t_j) - u(x_i, t_j - \Delta t)}{\Delta t}, \tag{15.44}$$

still with a truncation error which goes like $O(\Delta t)$. We could also have used a midpoint approximation for the first derivative, resulting in

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j - \Delta t)}{2\Delta t}, \tag{15.45}$$

with a truncation error $O(\Delta t^2)$. Here we will stick to the backward formula and come back to the latter below. For the second derivative we use however

$$u_{xx} \approx \frac{u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j)}{\Delta x^2}, \tag{15.46}$$

and define again $\alpha = \Delta t/\Delta x^2$. We obtain now

$$u_{i,j-1} = -\alpha u_{i-1,j} + (1 - 2\alpha)u_{i,j} - \alpha u_{i+1,j}. \tag{15.47}$$

Here $u_{i,j-1}$ is the only unknown quantity. Defining the matrix $\hat{A}$

$$\hat{A} = \left(\begin{array}{cccc} 1+2\alpha & -\alpha & 0 & 0\dots \\ -\alpha & 1+2\alpha & -\alpha & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & -\alpha & 1+2\alpha \end{array} \right), \tag{15.48}$$

we can reformulate again the problem as a matrix-vector multiplication

$$\hat{A}V_j = V_{j-1} \tag{15.49}$$

meaning that we can rewrite the problem as

$$V_j = \hat{A}^{-1}V_{j-1} = \hat{A}^{-1}\left(\hat{A}^{-1}V_{j-2}\right) = \dots = \hat{A}^{-j}V_0. \tag{15.50}$$

This is an implicit scheme since it relies on determining the vector $u_{i,j-1}$ instead of $u_{i,j+1}$. If α does not depend on time t, we need to invert a matrix only once. Alternatively we can solve this system of equations using our methods from linear algebra discussed in chapter 4. These are however very cumbersome ways of solving since they involve $\sim O(N^3)$ operations for a $N \times N$ matrix. It is much faster to solve these linear equations using methods for tridiagonal matrices, since these involve only $\sim O(N)$ operations. The function `tridag` of Ref. [36] is suitbale for these tasks.

The implicit scheme is always stable since the spectral radius satisfies $\rho(\hat{A}) < 1$. We could have inferred this by noting that the matrix is positive definite, viz. all eigenvalues are larger than zero. We see this from the fact that $\hat{A} = \hat{I} + \alpha\hat{B}$ has eigenvalues $\lambda_i = 1 + \alpha(2 - 2cos(\theta))$ which satisfy $\lambda_i > 1$. Since it is the inverse which stands to the right of our iterative equation, we have $\rho(\hat{A}^{-1}) < 1$ and the method is stable for all combinations of Δt and Δx. The calculational molecule for the implicit scheme is shown in Fig. 15.2.

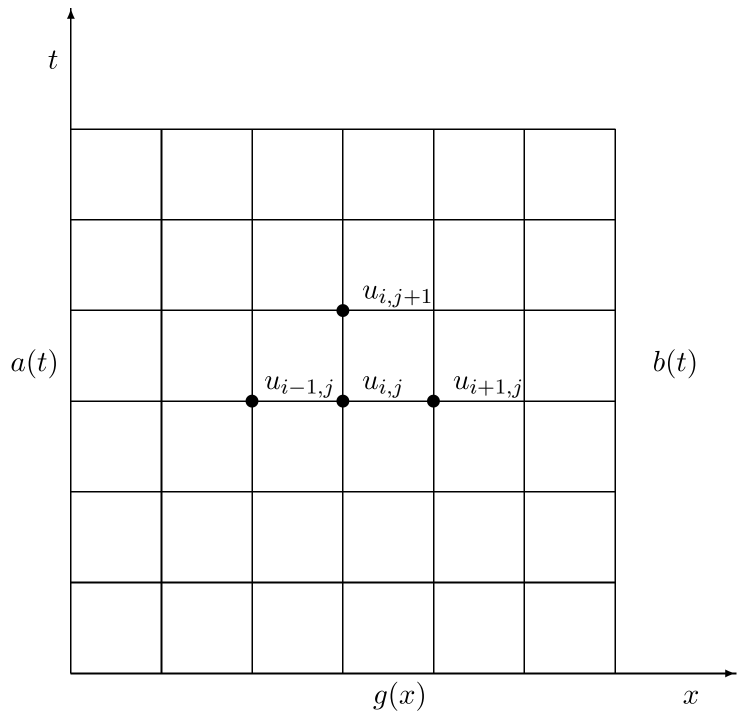


Figure 15.1: Discretization of the integration area used in the solution of the $1 + 1$-dimensional diffusion equation. This discretization is often called calculational molecule.

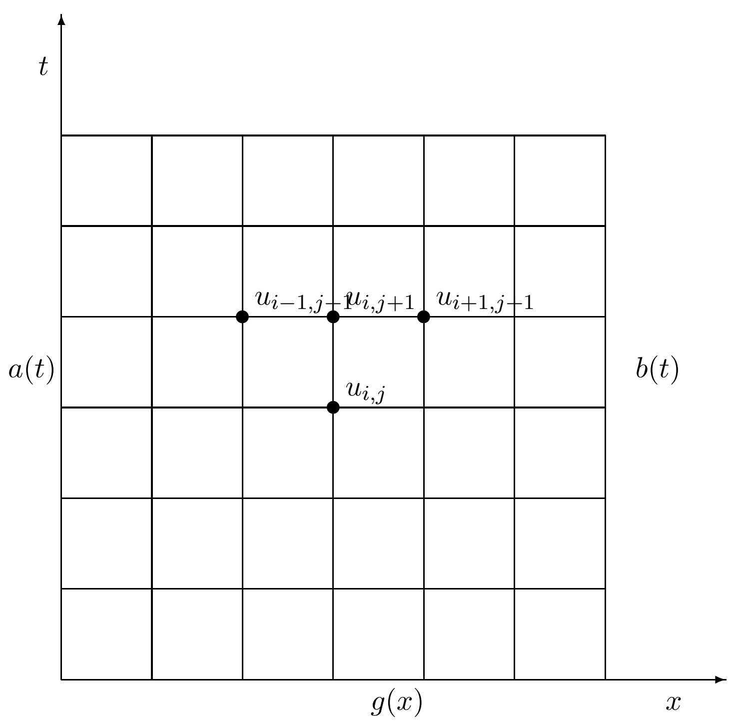


Figure 15.2: Calculational molecule for the implicit scheme.

Program example for implicit equation

We show here parts of a simple example of how to solve the one-dimensional diffusion equation using the implicit scheme discussed above. The program uses the function to solve linear equations with a tridiagonal matrix discussed in chapter 4.

```
//  parts of the function for backward Euler
void backward_euler(int xsteps, int tsteps, double delta_x, double alpha)
{
   double *v, *r, a, b, c;

   v = new double[xsteps+1]; // This is u
   r = new double[xsteps+1]; // Right side of matrix equation Av=r

   // Initialize vectors
   for (int i = 0; i < xsteps; i++) {
      r[i] = v[i] = func(delta_x*i);
   }
   r[xsteps] = v[xsteps] = 0;
   // Matrix A, only constants
   a = c = - alpha;
   b = 1 + 2*alpha;
   // Time iteration
   for (int t = 1; t <= tsteps; t++) {
      //  here we solve the tridiagonal linear set of equations
      tridag(a, b, c, r, v, x_steps+1);
      // boundary conditions
      v[0] = 0;
      v[xsteps] = 0;
      for (int i = 0; i <= x_steps; i++) {
         r[i] = v[i];
      }
   }
   ...
}
//  Function used to solve systems of equations for tridiagonal matrices
void tridag(double a, double b, double c, double *r, double *u, int n)
{
   double bet, *gam;
   gam = new double[n];
   bet = b;
   //  forward substitution
   u[0]=r[0]/bet;
   for (int j=1;j<n;j++) {
      gam[j] = c/bet;
      bet = b - a*gam[j];
      if (bet == 0.0) {cout << "Error 2 in tridag" << endl;}
      u[j] = (r[j] - a*u[j-1])/bet;
   }
   // backward substitution
   for (int j=n-2; j>=0; j--) {u[j] -= gam[j+1]*u[j+1];}
   delete [] gam;
}
```

15.2.3 Crank-Nicolson scheme

It is possible to combine the implicit and explicit methods in a slightly more general approach. Introducing a parameter θ (the so-called θ-rule) we can set up an equation

$$\frac{\theta}{\Delta x^2}\left(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}\right) + \frac{1-\theta}{\Delta x^2}\left(u_{i+1,j-1} - 2u_{i,j-1} + u_{i-1,j-1}\right) = \frac{1}{\Delta t}\left(u_{i,j} - u_{i,j-1}\right), \tag{15.51}$$

which for $\theta = 0$ yields the forward formula for the first derivative and the explicit scheme, while $\theta = 1$ yields the backward formula and the implicit scheme. These two schemes are called the backward and forward Euler schemes, respectively. For $\theta = 1/2$ we obtain a new scheme after its inventors, Crank and Nicolson. This scheme yields a truncation in time which goes like $O(\Delta t^2)$ and it is stable for all possible combinations of Δt and Δx.

Using our previous definition of $\alpha = \Delta t/\Delta x^2$ we can rewrite the latter equation as

$$-\alpha u_{i-1,j} + (2+2\alpha)\, u_{i,j} - \alpha u_{i+1,j} = \alpha u_{i-1,j-1} + (2-2\alpha)\, u_{i,j-1} + \alpha u_{i+1,j-1}, \tag{15.52}$$

or in matrix-vector form as

$$\left(2\hat{I} + \alpha\hat{B}\right) V_j = \left(2\hat{I} - \alpha\hat{B}\right) V_{j-1}, \tag{15.53}$$

where the vector V_j is the same as defined in the implicit case while the matrix $\hat{B}$ is

$$\hat{B} = \begin{pmatrix} 2 & -1 & 0 & 0\dots \\ -1 & 2 & -1 & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & & 2 \end{pmatrix} \tag{15.54}$$

We can rewrite the Crank-Nicolson scheme as follows

$$V_j = \left(2\hat{I} + \alpha\hat{B}\right)^{-1} \left(2\hat{I} - \alpha\hat{B}\right) V_{j-1}. \tag{15.55}$$

We have already obtained the eigenvalues for the two matrices $\left(2\hat{I} + \alpha\hat{B}\right)$ and $\left(2\hat{I} - \alpha\hat{B}\right)$. This means that the spectral function has to satisfy

$$\rho(\left(2\hat{I} + \alpha\hat{B}\right)^{-1} \left(2\hat{I} - \alpha\hat{B}\right)) < 1,$$

meaning that

$$\left|((2+\alpha\mu_i)^{-1}(2-\alpha\mu_i)\right| < 1,$$

and since $\mu_i = 2 - 2cos(\theta)$ we have $0 < \mu_i < 4$. A little algebra shows that the algorithm is stable for all possible values of Δt and Δx.

The calculational molecule for the Crank-Nicolson scheme is shown in Fig. 15.3.

Parts of code for the Crank-Nicolson scheme

We can code in an efficient way the Crank-Nicolson algortihm by first multplying the matrix

$$\tilde{V}_{j-1} = \left(2\hat{I} - \alpha\hat{B}\right) V_{j-1},$$

with our previous vector V_{j-1} using the matrix-vector multiplication algorithm for a tridiagonal matrix, as done in the forward-Euler scheme. Thereafter we can solve the equation

$$\left(2\hat{I}+\alpha\hat{B}\right)V_j=\tilde{V}_{j-1},$$

using our method for systems of linear equations with a tridiagonal matrix, as done for the backward Euler scheme.

We illustrate this in the following part of our program.

```
void crank_nicolson(int xsteps, int tsteps, double delta_x, double alpha)
{
   double *v, a, b, c, *r;

   v = new double[xsteps+1]; // This is u
   r = new double[xsteps+1]; // Right side of matrix equation Av=r
   ....
   // setting up the matrix
   a = c = - alpha;
   b = 2 + 2*alpha;

   // Time iteration
   for (int t = 1; t <= tsteps; t++) {

      // Calculate r for use in tridag, right hand side of the Crank
         Nicolson method
      for (int i = 1; i < xsteps; i++) {
         r[i] = alpha*v[i-1] + (2 - 2*alpha)*v[i] + alpha*v[i+1];
      }
      r[0] = 0;
      r[xsteps] = 0;
      //  Then solve the tridiagonal matrix
      tridag(a, b, c, r, v, xsteps+1);
      v[0] = 0;
      v[xsteps] = 0;
      ....
}
```

15.2.4 Numerical truncation

We start with the forward Euler scheme and Taylor expand $u(x,t+\Delta t)$, $u(x+\Delta x,t)$ and $u(x-\Delta x,t)$

$$\begin{aligned} u(x+\Delta x,t) &= u(x,t)+\frac{\partial u(x,t)}{\partial x}\Delta x+\frac{\partial^2 u(x,t)}{2\partial x^2}\Delta x^2+\mathcal{O}(\Delta x^3), \\ u(x-\Delta x,t) &= u(x,t)-\frac{\partial u(x,t)}{\partial x}\Delta x+\frac{\partial^2 u(x,t)}{2\partial x^2}\Delta x^2+\mathcal{O}(\Delta x^3), \\ u(x,t+\Delta t) &= u(x,t)+\frac{\partial u(x,t)}{\partial t}\Delta t+\mathcal{O}(\Delta t^2). \end{aligned} \tag{15.56}$$

With these Taylor expansions the approximations for the derivatives takes the form

$$\begin{aligned} \left[\frac{\partial u(x,t)}{\partial t}\right]_{\text{approx}} &= \frac{\partial u(x,t)}{\partial t}+\mathcal{O}(\Delta t), \\ \left[\frac{\partial^2 u(x,t)}{\partial x^2}\right]_{\text{approx}} &= \frac{\partial^2 u(x,t)}{\partial x^2}+\mathcal{O}(\Delta x^2). \end{aligned} \tag{15.57}$$

It is easy to convince oneself that the backward Euler method must have the same truncation errors as the forward Euler scheme.

For the Crank-Nicolson scheme we also need to Taylor expand $u(x+\Delta x, t+\Delta t)$ and $u(x-\Delta x, t+\Delta t)$ around $t' = t + \Delta t/2$.

$$
\begin{aligned}
u(x+\Delta x, t+\Delta t) &= u(x,t') + \frac{\partial u(x,t')}{\partial x}\Delta x + \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\frac{\Delta t^2}{4} + \\
&\quad \frac{\partial^2 u(x,t')}{\partial x \partial t}\frac{\Delta t}{2}\Delta x + \mathcal{O}(\Delta t^3) \\
u(x-\Delta x, t+\Delta t) &= u(x,t') - \frac{\partial u(x,t')}{\partial x}\Delta x + \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\frac{\Delta t^2}{4} - \\
&\quad \frac{\partial^2 u(x,t')}{\partial x \partial t}\frac{\Delta t}{2}\Delta x + \mathcal{O}(\Delta t^3) \\
u(x+\Delta x, t) &= u(x,t') + \frac{\partial u(x,t')}{\partial x}\Delta x - \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\frac{\Delta t^2}{4} - \\
&\quad \frac{\partial^2 u(x,t')}{\partial x \partial t}\frac{\Delta t}{2}\Delta x + \mathcal{O}(\Delta t^3) \\
u(x-\Delta x, t) &= u(x,t') - \frac{\partial u(x,t')}{\partial x}\Delta x - \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2 + \frac{\partial^2 u(x,t')}{2\partial t^2}\frac{\Delta t^2}{4} + \\
&\quad \frac{\partial^2 u(x,t')}{\partial x \partial t}\frac{\Delta t}{2}\Delta x + \mathcal{O}(\Delta t^3) \\
u(x, t+\Delta t) &= u(x,t') + \frac{\partial u(x,t')}{\partial t}\frac{\Delta_t}{2} + \frac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2 + \mathcal{O}(\Delta t^3) \\
u(x, t) &= u(x,t') - \frac{\partial u(x,t')}{\partial t}\frac{\Delta t}{2} + \frac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2 + \mathcal{O}(\Delta t^3)
\end{aligned}
$$

We now insert these expansions in the approximations for the derivatives to find

$$
\begin{aligned}
\left[\frac{\partial u(x,t')}{\partial t}\right]_{\text{approx}} &= \frac{\partial u(x,t')}{\partial t} + \mathcal{O}(\Delta t^2), \\
\left[\frac{\partial^2 u(x,t')}{\partial x^2}\right]_{\text{approx}} &= \frac{\partial^2 u(x,t')}{\partial x^2} + \mathcal{O}(\Delta x^2).
\end{aligned}
\tag{15.58}
$$

The following table summarizes the three methods.

Scheme:	*Truncation Error:*	*Stability requirements:*
Crank-Nicolson	$\mathcal{O}(\Delta x^2)$ and $\mathcal{O}(\Delta t^2)$	Stable for all Δt and Δx.
Backward Euler	$\mathcal{O}(\Delta x^2)$ and $\mathcal{O}(\Delta t)$	Stable for all Δt and Δx.
Forward Euler	$\mathcal{O}(\Delta x^2)$ aand $\mathcal{O}(\Delta t)$	$\Delta t \leq \frac{1}{2}\Delta x^2$

Table 15.1: Comparison of the different schemes.

15.2.5 Analytic solution for the one-dimensional diffusion equation

It cannot be repeated enough, it is always useful to find cases where one can compare the numerics and the developed algorithms and codes with analytic solution. The above case is also particularly simple. We have the following partial differential equation with

$$\nabla^2 u(x,t) = \frac{\partial u(x,t)}{\partial t},$$

with initial conditions

$$u(x,0) = g(x) \quad 0 < x < L.$$

The boundary conditions are

$$u(0,t) = 0 \quad t \geq 0, \qquad u(L,t) = 0 \quad t \geq 0,$$

We assume that we have solutions of the form (separation of variable)

$$u(x,t) = F(x)G(t). \tag{15.59}$$

which inserted in the partial differential equation results in

$$\frac{F''}{F} = \frac{G'}{G}, \tag{15.60}$$

where the derivative is with respect to x on the left hand side and with respect to t on right hand side. This equation should hold for all x and t. We must require the rhs and lhs to be equal to a constant. We call this constant $-\lambda^2$. This gives us the two differential equations,

$$F'' + \lambda^2 F = 0; \qquad G' = -\lambda^2 G, \tag{15.61}$$

with general solutions

$$F(x) = A\sin(\lambda x) + B\cos(\lambda x); \qquad G(t) = Ce^{-\lambda^2 t}. \tag{15.62}$$

To satisfy the boundary conditions we require $B = 0$ and $\lambda = n\pi/L$. One solution is therefore found to be

$$u(x,t) = A_n \sin(n\pi x/L)e^{-n^2\pi^2 t/L^2}. \tag{15.63}$$

But there are an infinitely many possible n values (infinite number of solutions). Moreover, the diffusion equation is linear and because of this we know that a superposition of solutions will also be a solution of the equation. We may therefore write

$$u(x,t) = \sum_{n=1}^{\infty} A_n \sin(n\pi x/L)e^{-n^2\pi^2 t/L^2}. \tag{15.64}$$

The coefficient A_n is in turn determined from the initial condition. We require

$$u(x,0) = g(x) = \sum_{n=1}^{\infty} A_n \sin(n\pi x/L). \tag{15.65}$$

The coefficient A_n is the Fourier coefficients for the function $g(x)$. Because of this, A_n is given by (from the theory on Fourier series)

$$A_n = \frac{2}{L}\int_0^L g(x)\sin(n\pi x/L)\mathrm{d}x. \tag{15.66}$$

Different $g(x)$ functions will obviously result in different results for A_n. A good discussion on Fourier series and their links with partial differential equations can be found in Ref. [77].

15.3 Laplace's and Poisson's equations

Laplace's equation reads

$$\nabla^2 u(\mathbf{x}) = u_{xx} + u_{yy} = 0. \tag{15.67}$$

with possible boundary conditions $u(x, y) = g(x, y)$ on the border $\delta\Omega$. There is no time-dependence. We seek a solution in the region Ω and we choose a quadratic mesh with equally many steps in both directions. We could choose the grid tobe rectangular or following polar coordinates r, θ as well. Here we choose equal steps lengths in the x and the y directions. We set

$$\Delta x = \Delta y = \frac{L}{n+1},$$

where L is the length of the sides and we have $n + 1$ points in both directions.

The discretized version reads

$$u_{xx} \approx \frac{u(x+h, y) - 2u(x, y) + u(x-h, y)}{h^2}, \tag{15.68}$$

and

$$u_{yy} \approx \frac{u(x, y+h) - 2u(x, y) + u(x, y-h)}{h^2}, \tag{15.69}$$

which we rewrite as

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \tag{15.70}$$

and

$$u_{yy} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}, \tag{15.71}$$

which gives when inserted in Laplace's equation

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right]. \tag{15.72}$$

This is our final numerical scheme for solving Laplace's equation. Poisson's equation adds only a minor complication to the above equation since in this case we have

$$u_{xx} + u_{yy} = -\rho(x, y),$$

and we need only to add a discretized version of $\rho(\mathbf{x})$ resulting in

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right] + \frac{h^2}{4}\rho_{i,j}. \tag{15.73}$$

The boundary condtions read

$$u_{i,0} = g_{i,0} \quad 0 \le i \le n+1,$$
$$u_{i,L} = g_{i,0} \quad 0 \le i \le n+1,$$
$$u_{0,j} = g_{0,j} \quad 0 \le j \le n+1,$$

and

$$u_{L,j} = g_{L,j} \quad 0 \le j \le n+1.$$

The calculational molecule for the Laplace operator of Eq. (15.72) is shown in Fig. 15.4.

With $n + 1$ mesh points the equations for u result in a system of $(n + 1)^2$ linear equations in the $(n + 1)^2$ unknown $u_{i,j}$. One can show that there exist unique solutions for the Laplace and Poisson problems, see for example Ref. [77] for proofs. However, solving these equations using for example the LU decomposition techniques discussed in chapter 4 becomes inefficient since the matrices are sparse. The relaxation techniques discussed below are more efficient.

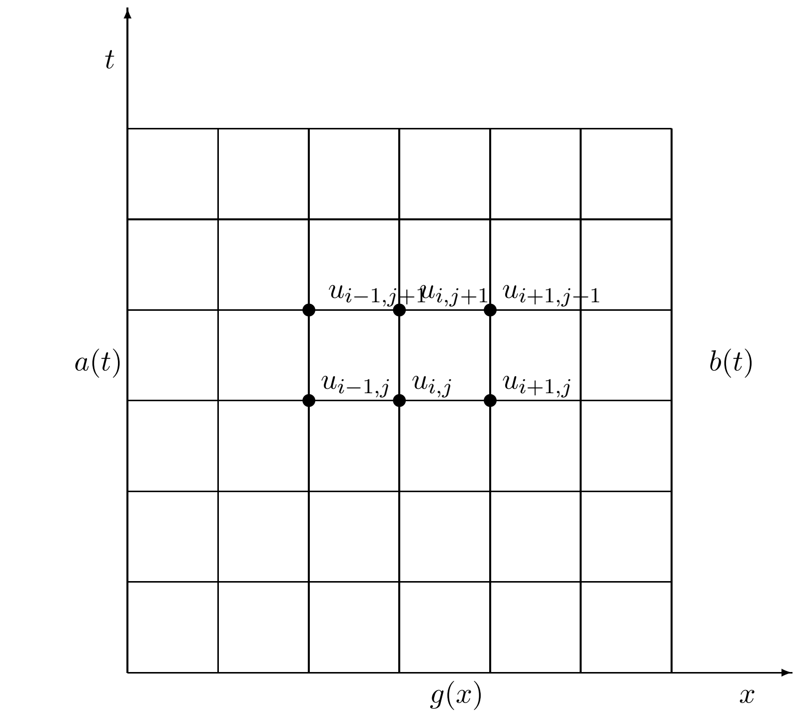


Figure 15.3: Calculational molecule for the Crank-Nicolson scheme.

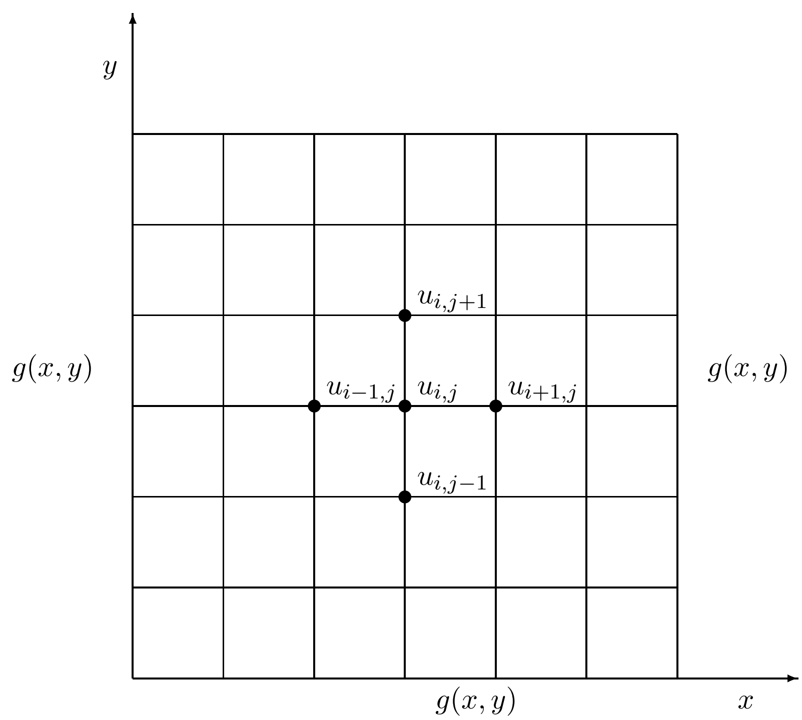


Figure 15.4: Five-point calculational molecule for the Laplace operator of Eq. (15.72). The border $\delta\Omega$ defines the boundary condition $u(x,y) = g(x,y)$.

15.3.1 Jacobi Algorithm for solving Laplace's equation

It is fairly straightforward to extend this equation to the three-dimensional case. Whether we solve Eq. (15.72) or Eq. (15.73), the solution strategy remains the same. We know the values of u at $i = 0$ or $i = n+1$ and at $j = 0$ or $j = n+1$ but we cannot start at one of the boundaries and work our way into and across the system since Eq. (15.72) requires the knowledge of u at all of the neighbouring points in order to calculate u at any given point.

The way we solve these equations is based on an iterative scheme called the Jacobi method or relaxation method. See for example Refs. [77, 79] for a discussion of the relaxation method and its pertinent proofs. Its steps are rather simple. We start with an initial guess for $u_{i,j}^{(0)}$ where all values are known. To obtain a new solution we solve Eq. (15.72) or Eq. (15.73) in order to obtain a new solution $u_{i,j}^{(1)}$. Most likely this solution will not be a solution to Eq. (15.72). This solution is in turn used to obtain a new and improved $u_{i,j}^{(2)}$. We continue this process till we obtain a result which satisfies some specific convergence criterion. Summarized, this algorithm reads

1. Make an initial guess for $u_{i,j}$ at all interior points (i,j) for all $i = 1:n$ and $j = 1:n$
2. Use Eq. (15.72) to compute u^m at all interior points (i,j). The index m stands for iteration number m.
3. Stop if prescribed convergence threshold is reached, otherwise continue on next step.
4. Update the new value of u for the given iteration
5. Go to step 2

A simple example may help in visualizing this method. We consider a condensator with parallel plates separated at a distance L resulting in e.g., the voltage differences $u(x,0) = 100sin(2\pi x/L)$ and $u(x,1) = -100sin(2\pi x/L)$. These are our boundary conditions and we ask what is the voltage u between the plates? To solve this problem numerically we provide below a Fortran program which solves iteratively Eq. (15.72). Only the part which computes Eq. (15.72) is included here.

```fortran
....
!  define the step size
      h = (xmax-xmin)/FLOAT(ndim+1)
      length = xmax-xmin
!  allocate space for the vector u and the temporary vector to
!  be upgraded in every iteration
      ALLOCATE ( u( ndim, ndim) )
      ALLOCATE ( u_temp( ndim, ndim) )
      pi = ACOS(-1.)
!  set up of initial conditions at t = 0 and boundary conditions
      u = 0.
      DO i=1, ndim
         x = i*h*pi/length
         u(i,1) = func(x)
         u(i,ndim) = -func(x)
      ENDDO
!  iteration algorithm starts here
      iterations = 0
      DO WHILE ( (iterations <= 20) .OR. ( diff > 0.00001) )
         u_temp = u; diff = 0.
```

```
        DO j = 2, ndim - 1
           DO l = 2, ndim -1
              u(j,l) = 0.25*(u_temp(j+1,l)+u_temp(j-1,l)+ &
                       u_temp(j,l+1)+u_temp(j,l-1))
              diff = diff + ABS(u_temp(i,j)-u(i,j))
           ENDDO
        ENDDO
        iterations = iterations + 1
        diff = diff/(ndim+1)**2
     ENDDO
```

The important part of the algorithm is applied in the function which sets up the two-dimensional Laplace equation. There we have a do-while statement which tests the difference between the temporary vector and the solution $u_{i,j}$. Moreover, we have fixed the number of iterations to be at most 20. This is sufficient for the above problem, but for more general applications you need to test the convergence of the algorithm.

While the Jacobi iteration scheme is very simple and parallelizable, its slow convergence rate renders it impractical for any "real world" applications. One way to speed up the convergent rate would be to "over predict" the new solution by linear extrapolation. This leads to the Successive Over Relaxation scheme, see chapter 19.5 on relaxation methods for boundary value problems of Ref. [36].

15.3.2 Laplace's equation and the parallel Jacobi algorithm

Ready end fall 2008.

15.3.3 Relaxation methods for boundary value problems with parallel implementation

Ready end fall 2008.

15.4 Wave equation in two dimensions

The $1+1$-dimensional wave equation reads

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial t^2}, \tag{15.74}$$

with $u = u(x,t)$ and we have assumed that we operate with dimensionless variables. Possible boundary and initial conditions with $L = 1$ are

$$\begin{cases} u_{xx} = u_{tt} & x \in (0,1), t > 0 \\ u(x,0) = g(x) & x \in (0,1) \\ u(0,t) = u(1,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = 0 & x \in (0,1) \end{cases}. \tag{15.75}$$

We discretize again time and position,

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2}, \tag{15.76}$$

and

$$u_{tt} \approx \frac{u(x,t+\Delta t) - 2u(x,t) + u(x,t-\Delta t)}{\Delta t^2}, \tag{15.77}$$

which we rewrite as

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}, \tag{15.78}$$

and

$$u_{tt} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta t^2}, \tag{15.79}$$

resulting in

$$u_{i,j+1} = 2u_{i,j} - u_{i,j-1} + \frac{\Delta t^2}{\Delta x^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right). \tag{15.80}$$

If we assume that all values at times $t = j$ and $t = j - 1$ are known, the only unknown variable is $u_{i,j+1}$ and the last equation yields thus an explicit scheme for updating this quantity. We have thus an explicit finite difference scheme for computing the wave function u. The only additional complication in our case is the initial condition given by the first derivative in time, namely $\partial u/\partial t|_{t=0} = 0$. The discretized version of this first derivative is given by

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j - \Delta t)}{2\Delta t}, \tag{15.81}$$

and at $t = 0$ it reduces to

$$u_t \approx \frac{u_{i,+1} - u_{i,-1}}{2\Delta t} = 0, \tag{15.82}$$

implying that $u_{i,+1} = u_{i,-1}$. If we insert this condition in Eq. (15.80) we arrive at a special formula for the first time step

$$u_{i,1} = u_{i,0} + \frac{\Delta t^2}{2\Delta x^2}\left(u_{i+1,0} - 2u_{i,0} + u_{i-1,0}\right). \tag{15.83}$$

We need seemingly two different equations, one for the first time step given by Eq. (15.83) and one for all other time-steps given by Eq. (15.80). However, it suffices to use Eq. (15.80) for all times as long as we provide $u(i, -1)$ using

$$u_{i,-1} = u_{i,0} + \frac{\Delta t^2}{2\Delta x^2}\left(u_{i+1,0} - 2u_{i,0} + u_{i-1,0}\right), \tag{15.84}$$

in our setup of the initial conditions.

The situation is rather similar for the $2 + 1$-dimensional case, except that we now need to discretize the spatial y-coordinate as well. Our equations will now depend on three variables whose discretized versions are now

$$\left\{\begin{array}{ll} t_l = l\Delta t & l \geq 0 \\ x_i = i\Delta x & 0 \leq i \leq n_x \\ y_j = j\Delta y & 0 \leq j \leq n_y \end{array}\right., \tag{15.85}$$

and we will let $\Delta x = \Delta y = h$ and $n_x = n_y$ for the sake of simplicity. The equation with initial and boundary conditions reads now

$$\left\{\begin{array}{cc} u_{xx} + u_{yy} = u_{tt} & x, y \in (0,1), t > 0 \\ u(x,y,0) = g(x,y) & x, y \in (0,1) \\ u(0,0,t) = u(1,1,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = 0 & x, y \in (0,1) \end{array}\right.. \tag{15.86}$$

We have now the following discretized partial derivatives

$$u_{xx} \approx \frac{u^l_{i+1,j} - 2u^l_{i,j} + u^l_{i-1,j}}{h^2}, \tag{15.87}$$

and

$$u_{yy} \approx \frac{u^l_{i,j+1} - 2u^l_{i,j} + u^l_{i,j-1}}{h^2}, \tag{15.88}$$

and

$$u_{tt} \approx \frac{u^{l+1}_{i,j} - 2u^l_{i,j} + u^{l-1}_{i,j}}{\Delta t^2}, \tag{15.89}$$

which we merge into the discretized $2+1$-dimensional wave equation as

$$u^{l+1}_{i,j} = 2u^l_{i,j} - u^{l-1}_{i,j} + \frac{\Delta t^2}{h^2}\left(u^l_{i+1,j} - 4u^l_{i,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1}\right), \tag{15.90}$$

where again we have an explicit scheme with $u^{l+1}_{i,j}$ as the only unknown quantity. It is easy to account for different step lengths for x and y. The partial derivative is treated in much the same way as for the one-dimensional case, except that we now have an additional index due to the extra spatial dimension, viz., we need to compute $u^{-1}_{i,j}$ through

$$u^{-1}_{i,j} = u^0_{i,j} + \frac{\Delta t}{2h^2}\left(u^0_{i+1,j} - 4u^0_{i,j} + u^0_{i-1,j} + u^0_{i,j+1} + u^0_{i,j-1}\right), \tag{15.91}$$

in our setup of the initial conditions.

15.4.1 Analytic solution

We develop here the analytic solution for the $2+1$ dimensional wave equation with the following boundary and initial conditions

$$\left\{\begin{array}{cc} c^2(u_{xx}+u_{yy}) = u_{tt} & x,y \in (0,L), t>0 \\ u(x,y,0) = f(x,y) & x,y \in (0,L) \\ u(0,0,t) = u(L,L,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = g(x,y) & x,y \in (0,L) \end{array}\right. .$$

Our first step is to make the ansatz

$$u(x,y,t) = F(x,y)G(t),$$

resulting in the equation

$$FG_{tt} = c^2(F_{xx}G + F_{yy}G),$$

or

$$\frac{G_{tt}}{c^2G} = \frac{1}{F}(F_{xx}+F_{yy}) = -\nu^2.$$

The lhs and rhs are independent of each other and we obtain two differential equations

$$F_{xx} + F_{yy} + F\nu^2 = 0,$$

and

$$G_{tt} + Gc^2\nu^2 = G_{tt} + G\lambda^2 = 0,$$

with $\lambda = c\nu$. We can in turn make the following ansatz for the x and y dependent part

$$F(x,y) = H(x)Q(y),$$

which results in

$$\frac{1}{H}H_{xx} = -\frac{1}{Q}(Q_{yy} + Q\nu^2) = -\kappa^2.$$

Since the lhs and rhs are again independent of each other, we can separate the latter equation into two independent equations, one for x and one for y, namely

$$H_{xx} + \kappa^2 H = 0,$$

and

$$Q_{yy} + \rho^2 Q = 0,$$

with $\rho^2 = \nu^2 - \kappa^2$.

The second step is to solve these differential equations, which all have trigonometric functions as solutions, viz.

$$H(x) = A\cos(\kappa x) + B\sin(\kappa x),$$

and

$$Q(y) = C\cos(\rho y) + D\sin(\rho y).$$

The boundary conditions require that $F(x,y) = H(x)Q(y)$ are zero at the boundaries, meaning that $H(0) = H(L) = Q(0) = Q(L) = 0$. This yields the solutions

$$H_m(x) = \sin(\frac{m\pi x}{L}) \qquad Q_n(y) = \sin(\frac{n\pi y}{L}),$$

or

$$F_{mn}(x,y) = \sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L}).$$

With $\rho^2 = \nu^2 - \kappa^2$ and $\lambda = c\nu$ we have an eigenspectrum $\lambda = c\sqrt{\kappa^2 + \rho^2}$ or $\lambda_{mn} = c\pi/L\sqrt{m^2 + n^2}$. The solution for G is

$$G_{mn}(t) = B_{mn}\cos(\lambda_{mn}t) + B^*_{mn}\sin(\lambda_{mn}t),$$

with the general solution of the form

$$u(x,y,t) = \sum_{mn=1}^{\infty} u_{mn}(x,y,t) = \sum_{mn=1}^{\infty} F_{mn}(x,y)G_{mn}(t).$$

The final step is to determine the coefficients B_{mn} and its complex conjugate B^*_{mn} from the Fourier coefficients. The equations for these are determined by the initial conditions $u(x,y,0) = f(x,y)$ and $\partial u/\partial t|_{t=0} = g(x,y)$. The final expressions are

$$B_{mn} = \frac{2}{L}\int_0^L\int_0^L dxdy f(x,y)\sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L}),$$

and

$$B^*_{mn} = \frac{2}{L}\int_0^L\int_0^L dxdy g(x,y)\sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L}).$$

Inserting the particular functional forms of $f(x,y)$ and $g(x,y)$ one obtains the final analytic expressions.

15.5 Exercises and projects

Project 15.1: two-dimensional wave equation

Consider the two-dimensional wave equation for a vibrating membrane given by the following initial and boundary conditions

$$\begin{cases} u_{xx}+u_{yy}=u_{tt} & x,y\in(0,1),t>0 \\ u(x,y,0)=sin(x)cos(y) & x,y\in(0,1) \\ u(0,0,t)=u(1,1,t)=0 & t>0 \\ \partial u/\partial t|_{t=0}=0 & x,y\in(0,1) \end{cases}.$$

a) Find the analytic solution for this equation using the technique of separation of variables.

b) Write down the algorithm for solving this equation and set up a program to solve the discretized wave equation. Compare your results with the analytic solution. Use a quadratic grid.

c) Consider thereafter a $2+1$ dimensional wave equation with variable velocity, given by

$$\frac{\partial^2 u}{\partial t^2}=\nabla(\lambda(x,y)\nabla u).$$

If λ is constant, we obtain the standard wave equation discussed in the two previous points. The solution $u(x,y,t)$ could represent a model for water waves. It represents then the surface elevation from still water. The function λ simulates the water depth using for example measurements of still water depths in say a fjord or the north sea. The boundary conditions are then determined by the coast lines. You can discretize

$$\nabla(\lambda(x,y)\nabla u)=\frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right),$$

as follows using again a quadratic domain for x and y:

$$\frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right)\approx\frac{1}{\Delta x}\left(\lambda_{i+1/2,j}\left[\frac{u^l_{i+1,j}-u^l_{i,j}}{\Delta x}\right]-\lambda_{i-1/2,j}\left[\frac{u^l_{i,j}-u^l_{i-1,j}}{\Delta x}\right]\right),$$

and

$$\frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right)\approx\frac{1}{\Delta y}\left(\lambda_{i,j+1/2}\left[\frac{u^l_{i,j+1}-u^l_{i,j}}{\Delta y}\right]-\lambda_{i,j-1/2}\left[\frac{u^l_{i,j}-u^l_{i,j-1}}{\Delta y}\right]\right).$$

Convince yourself that this equation has the same truncation error as the expressions used in a) and b) and that they result in the same equations when λ is a constant.

d) Develop an algorithm for solving the new wave equation and write a program which implements it.

Project 15.2, one- and two-dimensional diffusion equations

We are looking at a one-dimensional problem

$$\frac{\partial^2 u(x,t)}{\partial x^2}=\frac{\partial u(x,t)}{\partial t},t>0,x\in[0,L] \tag{15.92}$$

or

$$u_{xx} = u_t, \tag{15.93}$$

with initial conditions, i.e., the conditions at $t = 0$,

$$u(x,0) = 0 \quad 0 < x < L \tag{15.94}$$

with $L = 1$ the length of the x-region of interest. The boundary conditions are

$$u(0,t) = 0 \quad t > 0, \tag{15.95}$$

and

$$u(L,t) = 1 \quad t > 0. \tag{15.96}$$

The function $u(x,t)$ can be the temperature gradient of a the rod or represent the fluid velocity in a direction parallel to the plates, that is normal to the x-axis. In the latter case, for small t, only the part of the fluid close to the moving plate is set in significant motion, resulting in a thin boundary layer at $x = L$. As time increases, the velocity approaches a linear variation with x. In this case, which can be derived from the incompressible Navier-Stokes, the above equations constitute a model for studying friction between moving surfaces separated by a thin fluid film.

In this project we want to study the numerical stability of three methods for partial differential equations (PDEs). These methods are

1. The explicit forward Euler algorithm with discretized versions of time given by a forward formula and a centered difference in space resulting in

$$u_t \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t) - u(x_i,t_j)}{\Delta t} \tag{15.97}$$

and

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2}, \tag{15.98}$$

or

$$u_{xx} \approx \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2}. \tag{15.99}$$

2. The implicit Backward Euler with

$$u_t \approx \frac{u(x,t) - u(x,t-\Delta t)}{\Delta t} = \frac{u(x_i,t_j) - u(x_i,t_j-\Delta t)}{\Delta t} \tag{15.100}$$

and

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2}, \tag{15.101}$$

or

$$u_{xx} \approx \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2}, \tag{15.102}$$

3. Finally we use the implicit Crank-Nicolson scheme with a time-centered scheme at $(x, t+\Delta t/2)$

$$u_t \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t) - u(x_i,t_j)}{\Delta t}. \tag{15.103}$$

The corresponding spatial second-order derivative reads

$$u_{xx} \approx \frac{1}{2}\left(\frac{u(x_i+\Delta x,t_j)-2u(x_i,t_j)+u(x_i-\Delta x,t_j)}{\Delta x^2}+\right. \tag{15.104}$$

$$\left.\frac{u(x_i+\Delta x,t_j+\Delta t)-2u(x_i,t_j+\Delta t)+u(x_i-\Delta x,t_j+\Delta t)}{\Delta x^2}\right).$$

Note well that we are using a time-centered scheme wih $t+\Delta t/2$ as center.

a) Write down the algorithms for these three methods and the equations you need to implement. For the implicit schemes show that the equations lead to a tridiagonal matrix system for the new values.

b) Find the truncation errors of these three schemes and investigate their stability properties. Find also the analytic solution to the continuous problem. A useful hint here is to solve for $v(x,t) = u(x,t)-x$ instead. The boundary conditions for $v(x,t)$ are simpler, $v(0,t)=v(1,t)=0$ and the initial conditions are $v(x,0)=-x$.

c) Implement the three algorithms in the same code and perform tests of the solution for these three approaches for $\Delta x = 1/10$, $\Delta x = 1/100$ using Δt as dictated by the stability limit of the explicit scheme. Study the solutions at two time points t_1 and t_2 where $u(x,t_1)$ is smooth but still significantly curved and $u(x,t_2)$ is almost linear, close to the stationary state.

d) Compare the solutions at t_1 and t_2 with the analytic result for the continuous problem. Which of the schemes would you classify as the best?

e) Generalize this problem to two dimensions and write down the algorithm for the forward and backward Euler approaches. Write a program which solves the diffusion equation in $2+1$ dimensions. The program should allow for general boundary and initial conditions.

Part II

Advanced topics

Chapter 16

Many-body approaches to studies of atoms and molecules

16.1 Introduction

A theoretical understanding of the behavior of quantum mechanical systems with many interacting particles, normally called many-body systems, is a great challenge and provides fundamental insights into systems governed by quantum mechanics, as well as offering potential areas of industrial applications, from semi-conductor physics to the construction of quantum gates. The capability to simulate quantum mechanical systems with many interacting particles is crucial for advances in such rapidly developing fields like materials science.

However, most quantum mechanical systems of interest in physics consist of a large number of interacting particles. The total number of particles N is usually sufficiently large that an exact solution (viz., in closed form) cannot be found. One needs therefore reliable numerical methods for studying quantum mechanical systems with many particles.

Studies of many-body systems span from our understanding of the strong force with quarks and gluons as degrees of freedom, the spectacular macroscopic manifestations of quantal phenomena such as Bose-Einstein condensation with millions of atoms forming a coherent state, to properties of new materials, with electrons as effective degrees of freedom. The length scales range from few micrometers and nanometers, typical scales met in materials science, to $10^{-15} - 10^{-18}$ m, a relevant length scale for the strong interaction. Energies can span from few meV to GeV or even TeV. In some cases the basic interaction between the interacting particles is well-known. A good example is the Coulomb force, familiar from studies of atoms, molecules and condensed matter physics. In other cases, such as for the strong interaction between neutrons and protons (commonly dubbed as nucleons) or dense quantum liquids one has to resort to parameterizations of the underlying interparticle interactions. But the system can also span over much larger dimensions as well, with neutron stars as one of the classical objects. This star is the endpoint of massive stars which have used up their fuel. A neutron star, as its name suggests, is composed mainly of neutrons, with a small fraction of protons and probably quarks in its inner parts. The star is extremely dense and compact, with a radius of approximately 10 km and a mass which is roughly 1.5 times that of our sun. The quantum mechanical pressure which is set up by the interacting particles counteracts the gravitational forces, hindering thus a gravitational collapse. To describe a neutron star one needs to solve Schrödinger's equation for approximately 10^{54} interacting particles!

With a given interparticle potential and the kinetic energy of the system, one can in turn define the so-called many-particle Hamiltonian $\hat{H}$ which enters the solution of Schrödinger's equation or Dirac's equation in case relativistic effects need to be included. For many particles, Schrödinger's equation is an integro-differential equation whose complexity increases exponentially with increasing numbers of particles and states that the system can access. Unfortunately, apart from some few analytically solvable problems and one and two-particle systems that can be treated numerically exactly via the solution of sets of partial differential equations, the typical absence of an exactly solvable contribution to the many-particle Hamiltonian means that we need reliable numerical many-body methods. These methods should allow for controlled approximations and provide a computational scheme which accounts for successive many-body corrections in a systematic way.

Typical examples of popular many-body methods are coupled-cluster methods [66, 82–85], various types of Monte Carlo methods [86–88], perturbative many-body methods [89–91], Green's function methods [92, 93], the density-matrix renormalization group [94, 95], density functional theory [96] and ab initio density functional theory [97–99], and large-scale diagonalization methods [100–102], just to mention a few. The physics of the system hints at which many-body methods to use. For systems with strong correlations among the constituents methods based on mean-field theory such as Hartree-Fock theory and density functional theory are normally ruled out. This applies also to perturbative methods, unless one can renormalize the parts of the interaction which cause problems.

The aim of this and the next two chapters is to present to you many-body methods which can be used to study properties of atoms, molecules, systems in the solid state and nuclear physics.

In this chapter we limit ourselves to studies of atoms and molecules, as discussed in chapter 11 as well. Using the Born-Oppenheimer approximation we rewrote Schrödinger's equation for N electrons as

$$\left[-\sum_{i=1}^{N}\frac{1}{2}\nabla_i^2-\sum_{i=1}^{N}\frac{Z}{r_i}+\sum_{i<j}^{N}\frac{1}{r_{ij}}\right]\Psi(\mathbf{R})=E\Psi(\mathbf{R}),$$

where we let $\mathbf{R}$ represent the positions which the N electrons can take, that is $\mathbf{R}=\{\mathbf{r}_1,\mathbf{r}_2,\dots,\mathbf{r}_N\}$. With more than one electron present we cannot find an analytical solution and must resort to numerical efforts. In this chapter we will examine the theory of several numerical methods, commonly applied to the atomic problem.

We will solve this equation in two steps,and combine thereby two of the many-body methods discussed above. Since the electron-electron interaction is rather weak compared with the attraction from the nucleus, an independent particle picture is a viable first step towards the solution of Schrödinger's equation. We assume therefore that each electrons sees an effective field set up by the other electrons. This leads to an integro-differential equation and methods like Hartree-Fock theory and density functional theory. Hartree-Fock theory and density functional theories are discussed in the next section.

In practical terms, for the Hartree-Fock method we end up solving a one-particle equation, as is the case for the hydrogen atom but modified due to the screening from the other electrons. This modified single-particle equation reads (see Eq. (11.59 for the hydrogen case) in atomic units

$$-\frac{1}{2}\frac{d^2}{dr^2}u_{nl}(r)+\left(\frac{l(l+1)}{2r^2}-\frac{Z}{r}+\Phi(r)+F_{nl}\right)u_{nl}(r)=e_{nl}u_{nl}(r).$$

The function u_{nl} is the solution of the radial part of the Schrödinger equation and the functions $\Phi(r)$ and F_{nl} are the corrections due to the screening from the other electrons. We will derive these equations in the next section.

The total one-particle wave function, see chapter 11 is

$$\psi_{nlm_lsm_s}=\phi_{nlm_l}(\mathbf{r})\xi_{m_s}(s)$$

with s is the spin ($1/2$ for electrons), m_s is the spin projection $m_s = \pm 1/2$, and the spatial part is

$$\phi_{nlm_l}(\mathbf{r}) = R_{nl}(r) Y_{lm_l}(\hat{\mathbf{r}})$$

with Y the spherical harmonics discussed in chapter 11 and $u_{nl} = rR_{nl}$. The other quantum numbers are the orbital momentum l and its projection $m_l = -l, -l+1, \ldots, l-1, l$ and the principal quantum number $n = n_r + l + 1$, with n_r the number of nodes of a given single-particle wave function. All results are in atomic units, meaning that the energy is given by $e_{nl} = -Z^2/2n^2$ and the radius is dimensionless.

We obtain then a modified single-particle eigenfunction which in turn can be used as an input in a variational Monte Carlo calculation of the ground state of a specific atom. This is the aim of this chapter. Since Hartree-Fock theory does not treat correctly the role of many-body correlations, the hope is that performing a Monte Carlo calculation we may improve our results by obtaining a better agreement with experiment.

We focus here on the variational Monte Carlo method as a way to improve upon the Hartree-Fock results. The method was discussed in chapter 11. In chapter 11 however, we limited ourselves to the implementation of a bruce force Metropolis algorithm. Here we present the concept of importance sampling and improved statistical data analysis methods such as data blocking and the Jack-Knife method [56].

Although the variational Monte Carlo approach will improve our agreement with experiment compared with the Hartree-Fock results, there are still further possibilities for improvement. This is provided by Green's function Monte Carlo methods, which allow for an in principle exact calculation. The diffusion Monte Carlo method is discussed in chapter 17, with an application to studies of Bose-Einstein condensation.

Other many-body methods such as large-scale diagonalization and coupled-cluster theories are discussed in Ref. [103]. Finally, chapter 19 demonstrates how algorithms from quantum information theory can be used to solve Schrödinger's equation for many interacting particles.

16.2 Hartree-Fock theory

Hartree-Fock theory [66, 104] is one of the simplest approximate theories for solving the many-body Hamiltonian. It is based on a simple approximation to the true many-body wave-function; that the wave-function is given by a single Slater determinant of N orthonormal single-particle wave functions

$$\psi_{nlm_l sm_s} = \phi_{nlm_l}(\mathbf{r})\xi_{m_s}(s).$$

We use hereafter the shorthand $\psi_{nlm_l sm_s}(\mathbf{r}) = \psi_\alpha(\mathbf{r})$, where α now contains all the quantum numbers needed to specify a particular single-particle orbital.

The Slater determinant can then be written as

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N, \alpha, \beta, \ldots, \nu) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_\alpha(\mathbf{r}_1) & \psi_\alpha(\mathbf{r}_2) & \ldots & \psi_\alpha(\mathbf{r}_N) \\ \psi_\beta(\mathbf{r}_1) & \psi_\beta(\mathbf{r}_2) & \ldots & \psi_\beta(\mathbf{r}_N) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_\nu(\mathbf{r}_1) & \psi_\beta(\mathbf{r}_2) & \ldots & \psi_\beta(\mathbf{r}_N) \end{vmatrix}. \tag{16.1}$$

Here the variables $\mathbf{r}_i$ include the coordinates of spin and space of particle i. The quantum numbers $\alpha, \beta, \ldots, \nu$ encompass all possible quantum numbers needed to specify a particular system. As an example, consider the Neon atom, with ten electrons which can fill the $1s$, $2s$ and $2p$ single-particle orbitals.

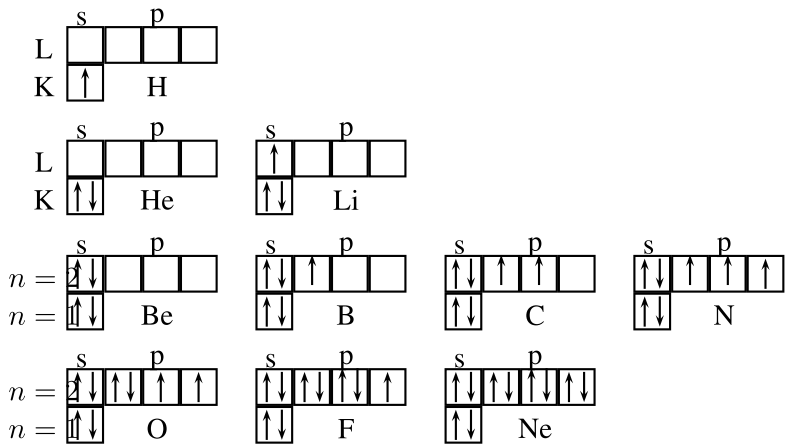


Figure 16.1: The electronic configurations for the ten first elements. We let an arrow which points upward to represent a state with $m_s = 1/2$ while an arrow which points downwards has $m_s = -1/2$.

Due to the spin projections m_s and orbital momentum projections m_l, the $1s$ and $2s$ states have a degeneracy of $2(2l+1) = 2$ while the $2p$ orbital has a degeneracy of $2(2l+1)2(2 \cdot 1+1) = 6$. This leads to ten possible values for $\alpha, \beta, \dots, \nu$. Fig. 16.1 shows the possible quantum numbers which the ten first elements can have.

If we consider the helium atom with two electrons in the $1s$ state, we can write the total Slater determinant as

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, \alpha, \beta) = \frac{1}{\sqrt{2}} \begin{vmatrix} \psi_\alpha(\mathbf{r}_1) & \psi_\alpha(\mathbf{r}_2) \\ \psi_\beta(\mathbf{r}_1) & \psi_\beta(\mathbf{r}_2) \end{vmatrix}, \tag{16.2}$$

with $\alpha = nlm_lsm_s = 1001/21/2$ and $\beta = nlm_lsm_s = 1001/2 - 1/2$ or using $m_s = 1/2 = \uparrow$ and $m_s = -1/2 = \downarrow$ as $\alpha = nlm_lsm_s = 1001/2 \uparrow$ and $\beta = nlm_lsm_s = 1001/2 \downarrow$. Writing out the Slater determinant we obtain

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, \alpha, \beta) = \frac{1}{\sqrt{2}} \left[\psi_\alpha(\mathbf{r}_1)\psi_\beta(\mathbf{r}_2) - \psi_\beta(\mathbf{r}_1)\psi_\gamma(\mathbf{r}_2)\right], \tag{16.3}$$

and we see that the Slater determinant is antisymmetric with respect to the permutation of two particles, that is

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, \alpha, \beta) - \Phi(\mathbf{r}_2, \mathbf{r}_1, \alpha, \beta),$$

For three electrons we have the general expression

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \alpha, \beta, \gamma) = \frac{1}{\sqrt{3!}} \begin{vmatrix} \psi_\alpha(\mathbf{r}_1) & \psi_\alpha(\mathbf{r}_2) & \psi_\alpha(\mathbf{r}_3) \\ \psi_\beta(\mathbf{r}_1) & \psi_\beta(\mathbf{r}_2) & \psi_\beta(\mathbf{r}_3) \\ \psi_\gamma(\mathbf{r}_1) & \psi_\gamma(\mathbf{r}_2) & \psi_\gamma(\mathbf{r}_3) \end{vmatrix}. \tag{16.4}$$

Computing the determinant gives

$$\begin{aligned} \Phi(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \alpha, \beta, \gamma) &= \tfrac{1}{\sqrt{3!}} \left[\psi_\alpha(\mathbf{r}_1)\psi_\beta(\mathbf{r}_2)\psi_\gamma(\mathbf{r}_3) + \psi_\beta(\mathbf{r}_1)\psi_\gamma(\mathbf{r}_2)\psi_\alpha(\mathbf{r}_3) + \psi_\gamma(\mathbf{r}_1)\psi_\alpha(\mathbf{r}_2)\psi_\beta(\mathbf{r}_3) - \right. \\ &\left. \psi_\gamma(\mathbf{r}_1)\psi_\beta(\mathbf{r}_2)\psi_\alpha(\mathbf{r}_3) - \psi_\beta(\mathbf{r}_1)\psi_\alpha(\mathbf{r}_2)\psi_\gamma(\mathbf{r}_3) - \psi_\alpha(\mathbf{r}_1)\psi_\gamma(\mathbf{r}_2)\psi_\beta(\mathbf{r}_3)\right]. \end{aligned} \tag{16.5}$$

We note again that the wave-function is antisymmetric with respect to an interchange of any two electrons, as required by the Pauli principle. For an N-body Slater determinant we have thus (omitting the quantum numbers $\alpha, \beta, \dots, \nu$)

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_i, \dots, \mathbf{r}_j, \dots \mathbf{r}_N) = -\Phi(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_j, \dots, \mathbf{r}_i, \dots \mathbf{r}_N).$$

We rewrite our Hamiltonian

$$\hat{H} = -\sum_{i=1}^{N} \frac{1}{2}\nabla_i^2 - \sum_{i=1}^{N} \frac{Z}{r_i} + \sum_{i<j}^{N} \frac{1}{r_{ij}},$$

as

$$\hat{H} = \hat{H}_1 + \hat{H}_2 = \sum_{i=1}^{N} \hat{h}_i + \sum_{i<j=1}^{N} \frac{1}{r_{ij}}, \tag{16.6}$$

where

$$\hat{h}_i = -\frac{1}{2}\nabla_i^2 - \frac{Z}{r_i}. \tag{16.7}$$

The first term of eq. (16.6), H_1, is the sum of the N identical *one-body* Hamiltonians $\hat{h}_i$. Each individual Hamiltonian $\hat{h}_i$ contains the kinetic energy operator of an electron and its potential energy due to the attraction of the nucleus. The second term, H_2, is the sum of the $N(N-1)/2$ two-body interactions between each pair of electrons. Let us denote the ground state energy by E_0. According to the variational principle we have

$$E_0 \le E[\Phi] = \int \Phi^* \hat{H} \Phi d\tau \tag{16.8}$$

where Φ is a trial function which we assume to be normalized

$$\int \Phi^* \Phi d\tau = 1, \tag{16.9}$$

where we have used the shorthand $d\tau = d\mathbf{r}_1 d\mathbf{r}_2 \dots d\mathbf{r}_N$. In the Hartree-Fock method the trial function is the Slater determinant of Eq. (16.1) which can be rewritten as

$$\Psi(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N, \alpha, \beta, \dots, \nu) = \frac{1}{\sqrt{N!}} \sum_P (-)^P P \psi_\alpha(\mathbf{r}_1)\psi_\beta(\mathbf{r}_2) \dots \psi_\nu(\mathbf{r}_N) = \sqrt{N!}\mathcal{A}\Phi_H, \tag{16.10}$$

where we have introduced the anti-symmetrization operator $\mathcal{A}$ defined by the summation over all possible permutations of two eletrons. It is defined as

$$\mathcal{A} = \frac{1}{N!} \sum_P (-)^P P, \tag{16.11}$$

with the the Hartree-function given by the simple product of all possible single-particle function (two for helium, four for beryllium and ten for neon)

$$\Phi_H(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N, \alpha, \beta, \dots, \nu) = \psi_\alpha(\mathbf{r}_1)\psi_\beta(\mathbf{r}_2) \dots \psi_\nu(\mathbf{r}_N). \tag{16.12}$$

Both $\hat{H}_1$ and $\hat{H}_2$ are invariant under electron permutations, and hence commute with $\mathcal{A}$

$$[H_1, \mathcal{A}] = [H_2, \mathcal{A}] = 0. \tag{16.13}$$

Furthermore, $\mathcal{A}$ satisfies

$$\mathcal{A}^2 = \mathcal{A}, \tag{16.14}$$

since every permutation of the Slater determinant reproduces it. The expectation value of $\hat{H}_1$

$$\int \Phi^* \hat{H}_1 \Phi d\tau = N! \int \Phi_H^* \mathcal{A} \hat{H}_1 \mathcal{A} \Phi_H d\tau$$

is readily reduced to

$$\int \Phi^* \hat{H}_1 \Phi d\tau = N! \int \Phi_H^* \hat{H}_1 \mathcal{A} \Phi_H d\tau,$$

where we have used eqs. (16.13) and (16.14). The next step is to replace the anti-symmetry operator by its definition eq. (16.10) and to replace $\hat{H}_1$ with the sum of one-body operators

$$\int \Phi^* \hat{H}_1 \Phi d\tau = \sum_{i=1}^{N} \sum_{P} (-)^P \int \Phi_H^* \hat{h}_i P \Phi_H d\tau. \tag{16.15}$$

The integral vanishes if two or more electrons are permuted in only one of the Hartree-functions Φ_H because the individual orbitals are orthogonal. We obtain then

$$\int \Phi^* \hat{H}_1 \Phi d\tau = \sum_{i=1}^{N} \int \Phi_H^* \hat{h}_i \Phi_H d\tau. \tag{16.16}$$

Orthogonality allows us to further simplify the integral, and we arrive at the following expression for the expectation values of the sum of one-body Hamiltonians

$$\int \Phi^* \hat{H}_1 \Phi d\tau = \sum_{\mu=1}^{N} \int \psi_\mu^*(\mathbf{r}_i) \hat{h}_i \psi_\mu(\mathbf{r}_i) d\mathbf{r}_i. \tag{16.17}$$

The expectation value of the two-body Hamiltonian is obtained in a similar manner. We have

$$\int \Phi^* \hat{H}_2 \Phi d\tau = N! \int \Phi_H^* \mathcal{A} \hat{H}_2 \mathcal{A} \Phi_H d\tau, \tag{16.18}$$

which reduces to

$$\int \Phi^* \hat{H}_2 \Phi d\tau = \sum_{i \le j=1}^{N} \sum_{P} (-)^P \int \Phi_H^* \frac{1}{r_{ij}} P \Phi_H d\tau, \tag{16.19}$$

by following the same arguments as for the one-body Hamiltonian. Because of the dependence on the inter-electronic distance $1/r_{ij}$, permutations of two electrons no longer vanish, and we get

$$\int \Phi^* \hat{H}_2 \Phi d\tau = \sum_{i<j=1}^{N} \int \Phi_H^* \frac{1}{r_{ij}} (1 - P_{ij}) \Phi_H d\tau. \tag{16.20}$$

where P_{ij} is the permutation operator that interchanges electrons i and j. Again we use the assumption that the orbitals are orthogonal, and obtain

$$\int \Phi^* \hat{H}_2 \Phi d\tau = \frac{1}{2} \sum_{\mu=1}^{N} \sum_{\nu=1}^{N} \left[\int \psi_\mu^*(\mathbf{r}_i) \psi_\nu^*(\mathbf{r}_j) \frac{1}{r_{ij}} \psi_\mu(\mathbf{r}_i) \psi_\nu(\mathbf{r}_j) d\mathbf{x_i x_j} - \int \psi_\mu^*(\mathbf{r}_i) \psi_\nu^*(\mathbf{r}_j) \frac{1}{r_{ij}} \psi_\nu(\mathbf{r}_i) \psi_\mu(\mathbf{r}_i) d\mathbf{x_i x_j} \right]. \tag{16.21}$$

The first term is the so-called direct term or Hartree term, while the second is due to the Pauli principle and is called exchange term or Fock term. The factor $1/2$ is introduced because we now run over all pairs twice.

Combining Eqs. (16.17) and (16.21) we obtain the functional

$$E[\Phi] = \sum_{\mu=1}^{N} \int \psi_\mu^*(\mathbf{r}_i)\hat{h}_i\psi_\mu(\mathbf{r}_i)d\mathbf{r}_i + \frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\left[\int \psi_\mu^*(\mathbf{r}_i)\psi_\nu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\mu(\mathbf{r}_i)\psi_\nu(\mathbf{r}_j)d\mathbf{r}_i d\mathbf{r}_j - \int \psi_\mu^*(\mathbf{r}_i)\psi_\nu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\nu(\mathbf{r}_i)\psi_\mu(\mathbf{r}_j)\right. \tag{16.22}$$

16.2.1 Derivation of the Hartree-Fock equations

Having obtained the functional $E[\Phi]$, we now proceed to the second step of the calculation. Before we proceed we need however to repeat some aspects of the calculus of variations. For more details see for example the text of Arfken [78].

We have already met the variational principle in chapter 11 via the so-called Rayleigh-Ritz variational technique. We give here a brief reminder on the calculus of variations.

Small digression on the calculus of variations

The calculus of variations involves problems where the quantity to be minimized or maximized is an integral.

In the general case we have an integral of the type

$$E[\Phi] = \int_a^b f(\Phi(x), \frac{\partial\Phi}{\partial x}, x)dx,$$

where E is the quantity which is sought minimized or maximized. The problem is that although f is a function of the variables Φ, $\partial\Phi/\partial x$ and x, the exact dependence of Φ on x is not known. This means again that even though the integral has fixed limits a and b, the path of integration is not known. In our case the unknown quantities are the single-particle wave functions and we wish to choose an integration path which makes the functional $E[\Phi]$ stationary. This means that we want to find minima, or maxima or saddle points. In physics we search normally for minima. Our task is therefore to find the minimum of $E[\Phi]$ so that its variation δE is zero subject to specific constraints. In our case the constraints appear as the integral which expresses the orthogonality of the single-particle wave functions. The constraints can be treated via the technique of Lagrangian multipliers

We assume the existence of an optimum path, that is a path for which $E[\Phi]$ is stationary. There are infinitely many such paths. The difference between two paths $\delta\Phi$ is called the variation of Φ.

We call the variation $\eta(x)$ and it is scaled by a factor α. The function $\eta(x)$ is arbitrary except for

$$\eta(a) = \eta(b) = 0,$$

and we assume that we can model the change in Φ as

$$\Phi(x,\alpha) = \Phi(x,0) + \alpha\eta(x),$$

and

$$\delta\Phi = \Phi(x,\alpha) - \Phi(x,0) = \alpha\eta(x).$$

We choose $\Phi(x, \alpha = 0)$ as the unkonwn path that will minimize E. The value $\Phi(x, \alpha \neq 0)$ describes a neighbouring path.

We have

$$E[\Phi(\alpha)] = \int_a^b f(\Phi(x,\alpha), \frac{\partial \Phi(x,\alpha)}{\partial x}, x)dx.$$

We introduce the shorthand

$$\Phi_x(x,\alpha) = \frac{\partial \Phi(x,\alpha)}{\partial x}.$$

In our case $a = 0$ and $b = \infty$ and we know the value of the wave function.

The condition for an extreme of

$$E[\Phi(\alpha)] = \int_a^b f(\Phi(x,\alpha), \Phi_x(x,\alpha), x)dx,$$

is

$$\left[\frac{\partial E[\Phi(\alpha)]}{\partial x}\right]_{\alpha=0} = 0.$$

The α dependence is contained in $\Phi(x,\alpha)$ and $\Phi_x(x,\alpha)$ meaning that

$$\left[\frac{\partial E[\Phi(\alpha)]}{\partial \alpha}\right] = \int_a^b \left(\frac{\partial f}{\partial \Phi}\frac{\partial \Phi}{\partial \alpha} + \frac{\partial f}{\partial \Phi_x}\frac{\partial \Phi_x}{\partial \alpha}\right) dx.$$

We have defined

$$\frac{\partial \Phi(x,\alpha)}{\partial \alpha} = \eta(x)$$

and thereby

$$\frac{\partial \Phi_x(x,\alpha)}{\partial \alpha} = \frac{d(\eta(x))}{dx}.$$

Using

$$\frac{\partial \Phi(x,\alpha)}{\partial \alpha} = \eta(x),$$

and

$$\frac{\partial \Phi_x(x,\alpha)}{\partial \alpha} = \frac{d(\eta(x))}{dx},$$

in the integral gives

$$\left[\frac{\partial E[\Phi(\alpha)]}{\partial \alpha}\right] = \int_a^b \left(\frac{\partial f}{\partial \Phi}\eta(x) + \frac{\partial f}{\partial \Phi_x}\frac{d(\eta(x))}{dx}\right) dx.$$

Integrate the second term by parts

$$\int_a^b \frac{\partial f}{\partial \Phi_x}\frac{d(\eta(x))}{dx}dx = \eta(x)\frac{\partial f}{\partial \Phi_x}|_a^b - \int_a^b \eta(x)\frac{d}{dx}\frac{\partial f}{\partial \Phi_x}dx,$$

and since the first term dissappears due to $\eta(a) = \eta(b) = 0$, we obtain

$$\left[\frac{\partial E[\Phi(\alpha)]}{\partial \alpha}\right] = \int_a^b \left(\frac{\partial f}{\partial \Phi} - \frac{d}{dx}\frac{\partial f}{\partial \Phi_x}\right) \eta(x)dx = 0.$$

$$\left[\frac{\partial E[\Phi(\alpha)]}{\partial \alpha}\right] = \int_a^b \left(\frac{\partial f}{\partial \Phi} - \frac{d}{dx}\frac{\partial f}{\partial \Phi_x}\right) \eta(x)dx = 0,$$

can also be written as

$$\alpha\left[\frac{\partial E[\Phi(\alpha)]}{\partial \alpha}\right]_{\alpha=0} = \int_a^b \left(\frac{\partial f}{\partial \Phi} - \frac{d}{dx}\frac{\partial f}{\partial \Phi_x}\right)\delta\Phi(x)dx = \delta E = 0.$$

The condition for a stationary value is thus a partial differential equation

$$\frac{\partial f}{\partial \Phi} - \frac{d}{dx}\frac{\partial f}{\partial \Phi_x} = 0,$$

known as Euler's equation. Can easily be generalized to more variables.

Consider a function of three independent variables $f(x,y,z)$. For the function f to be an extreme we have

$$df = 0.$$

A necessary and sufficient condition is

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} = \frac{\partial f}{\partial z} = 0,$$

due to

$$df = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy + \frac{\partial f}{\partial z}dz.$$

In physical problems the variables x, y, z are often subject to constraints (in our case Φ and the orthogonality constraint) so that they are no longer all independent. It is possible at least in principle to use each constraint to eliminate one variable and to proceed with a new and smaller set of independent varables.

The use of so-called Lagrangian multipliers is an alternative technique when the elimination of of variables is incovenient or undesirable. Assume that we have an equation of constraint on the variables x, y, z

$$\phi(x, y, z) = 0,$$

resulting in

$$d\phi = \frac{\partial \phi}{\partial x}dx + \frac{\partial \phi}{\partial y}dy + \frac{\partial \phi}{\partial z}dz = 0.$$

Now we cannot set anymore

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} = \frac{\partial f}{\partial z} = 0,$$

if $df = 0$ is wanted because there are now only two independent variables! Assume x and y are the independent variables. Then dz is no longer arbitrary.

However, we can add to

$$df = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy + \frac{\partial f}{\partial z}dz,$$

a multiplum of $d\phi$, viz. $\lambda d\phi$, resulting in

$$df + \lambda d\phi = (\frac{\partial f}{\partial z} + \lambda\frac{\partial \phi}{\partial x})dx + (\frac{\partial f}{\partial y} + \lambda\frac{\partial \phi}{\partial y})dy + (\frac{\partial f}{\partial z} + \lambda\frac{\partial \phi}{\partial z})dz = 0.$$

Our multiplier is chosen so that

$$\frac{\partial f}{\partial z} + \lambda\frac{\partial \phi}{\partial z} = 0.$$

However, we took dx and dy as to be arbitrary and thus we must have

$$\frac{\partial f}{\partial x} + \lambda \frac{\partial \phi}{\partial x} = 0,$$

and

$$\frac{\partial f}{\partial y} + \lambda \frac{\partial \phi}{\partial y} = 0.$$

When all these equations are satisfied, $df = 0$. We have four unknowns, x, y, z and λ. Actually we want only x, y, z, λ need not to be determined, it is therefore often called Lagrange's undetermined multiplier. If we have a set of constraints ϕ_k we have the equations

$$\frac{\partial f}{\partial x_i} + \sum_k \lambda_k \frac{\partial \phi_k}{\partial x_i} = 0.$$

Let us specialize to the expectation value of the energy for one particle in three-dimensions. This expectation value reads

$$E = \int dxdydz \psi^*(x, y, z) \hat{H} \psi(x, y, z),$$

with the constraint

$$\int dxdydz \psi^*(x, y, z) \psi(x, y, z) = 1,$$

and a Hamiltonian

$$\hat{H} = -\frac{1}{2}\nabla^2 + V(x, y, z).$$

I will skip the variables x, y, z below, and write for example $V(x, y, z) = V$.

The integral involving the kinetic energy can be written as, if we assume periodic boundary conditions or that the function ψ vanishes strongly for large values of x, y, z,

$$\int dxdydz \psi^* \left(-\frac{1}{2}\nabla^2\right) \psi dxdydz = \psi^* \nabla \psi | + \int dxdydz \frac{1}{2} \nabla \psi^* \nabla \psi.$$

Inserting this expression into the expectation value for the energy and taking the variational minimum we obtain

$$\delta E = \delta \left\{ \int dxdydz \left(\frac{1}{2} \nabla \psi^* \nabla \psi + V \psi^* \psi \right) \right\} = 0.$$

The constraint appears in integral form as

$$\int dxdydz \psi^* \psi = \text{constant},$$

and multiplying with a Lagrangian multiplier λ and taking the variational minimum we obtain the final variational equation

$$\delta \left\{ \int dxdydz \left(\frac{1}{2} \nabla \psi^* \nabla \psi + V \psi^* \psi - \lambda \psi^* \psi \right) \right\} = 0.$$

Introducing the function f

$$f = \frac{1}{2} \nabla \psi^* \nabla \psi + V \psi^* \psi - \lambda \psi^* \psi = \frac{1}{2}(\psi_x^* \psi_x + \psi_y^* \psi_y + \psi_z^* \psi_z) + V \psi^* \psi - \lambda \psi^* \psi,$$

where we have skipped the dependence on x, y, z and introduced the shorthand ψ_x, ψ_y and ψ_z for the various derivatives.

For ψ^* the Euler equation results in

$$\frac{\partial f}{\partial \psi^*} - \frac{\partial}{\partial x}\frac{\partial f}{\partial \psi_x^*} - \frac{\partial}{\partial y}\frac{\partial f}{\partial \psi_y^*} - \frac{\partial}{\partial z}\frac{\partial f}{\partial \psi_z^*} = 0,$$

which yields

$$-\frac{1}{2}(\psi_{xx} + \psi_{yy} + \psi_{zz}) + V\psi = \lambda\psi.$$

We can then identify the Lagrangian multiplier as the energy of the system. Then the last equation is nothing but the standard Schrödinger equation and the variational approach discussed here provides a powerful method for obtaining approximate solutions of the wave function.

Obtaining the Hartree-Fock equations

The calculus of variations involves problems where the quantity to be minimized or maximized is an integral. In our case the functional of Eq. (16.22) is stationary with respect to variations of the single-particle wave functions, subject to the N^2 conditions imposed by the orthogonality requirements,

$$\int \psi_\mu(\mathbf{r})\psi_\nu(\mathbf{r})d\mathbf{r} = \delta_{\mu,\nu}$$

In the general case we have an integral of the type

$$E = \int_a^b f(y(x), \frac{\partial y}{\partial x}, x)dx,$$

where E is the quantity which is sought minimized or maximized. The problem is that although f is a function of the variables y, $\partial y/\partial x$ and x, the exact dependence of y on x is not known. This means again that even though the integral has fixed limits a and b, the path of integration is not known. In our case the unknown quantities are the single-particle wave functions and we wish to choose an integration path which makes the functional $E[\Phi]$ stationary. This means that we want to find minima, or maxima or saddle points. In physics we search normally for minima. Our task is therefore to find the minimum of $E[\Phi]$ so that its variation δE is zero subject to specific constraints. In our case the constraints appear as the integral which expresses the orthogonality of the single-particle wave functions. The constraints can be treated via the technique of Lagrangian multipliers

Let us specialize to the expectation value of the energy for one particle in three-dimensions. This expectation value reads

$$E = \int dxdydz\psi^*(x, y, z)\hat{H}\psi(x, y, z),$$

with the constraint

$$\int dxdydz\psi^*(x, y, z)\psi(x, y, z) = 1,$$

and a Hamiltonian

$$\hat{H} = -\frac{1}{2}\nabla^2 + V(x, y, z).$$

The integral involving the kinetic energy can be written as, if we assume periodic boundary conditions or that the function ψ vanishes strongly for large valuesof x, y, z,

$$\int dxdydz\psi^*(x, y, z)\left(-\frac{1}{2}\nabla^2\right)\psi(x, y, z)dxdydz = \int dxdydz\frac{1}{2}\nabla\psi^*\nabla\psi.$$

Inserting this expression into the expectation value for the energy and taking the variational minimum we obtain

$$\delta E = \delta \left\{ \int dxdydz \left(\frac{1}{2}\nabla\psi^*\nabla\psi + V(x,y,z)\psi^*(x,y,z)\psi(x,y,z) \right) \right\} = 0.$$

The constraint appears in integral form as

$$\int dxdydz\psi^*(x,y,z)\psi(x,y,z) = \text{constant},$$

and multiplying with a Lagrangian multiplier λ and taking the variational minimum we obtain the final variational equation

$$\delta \left\{ \int dxdydz \left(\frac{1}{2}\nabla\psi^*\nabla\psi + V(x,y,z)\psi^*(x,y,z)\psi(x,y,z) - \lambda\psi^*(x,y,z)\psi(x,y,z) \right) \right\} = 0. \tag{16.23}$$

Introducing the function f

$$f = \frac{1}{2}\nabla\psi^*\nabla\psi + V(x,y,z)\psi^*(x,y,z)\psi(x,y,z) - \lambda\psi^*(x,y,z)\psi(x,y,z) = \frac{1}{2}(\psi_x^*\psi_x + \psi_y^*\psi_y + \psi_z^*\psi_z) + V\psi^*\psi - \lambda\psi^*\psi,$$

where we have skipped the dependence on x, y, z and introduced the shorthand ψ_x, ψ_y and ψ_z for the various derivatives. The Euler-Lagrange equations, see for example Ref. [78], result in

$$\frac{\partial f}{\partial\psi^*} - \frac{\partial}{\partial x}\frac{\partial f}{\partial\psi_x^*} - \frac{\partial}{\partial y}\frac{\partial f}{\partial\psi_y^*} - \frac{\partial}{\partial z}\frac{\partial f}{\partial\psi_z^*} = 0,$$

which yields

$$-\frac{1}{2}(\psi_{xx} + \psi_{yy} + \psi_{zz}) + V\psi = \lambda\psi. \tag{16.24}$$

We can then identify the Lagrangian multiplier as the energy of the system. Then the last equation is nothing but the standard Schrödinger equation and the variational approach discussed here provides a powerful method for obtaining approximate solutions of the wave function.

If we generalize the Euler-Lagrange equations to more variables and introduce N^2 Lagrange multipliers which we denote by $\epsilon_{\mu\nu}$, we can write the variational equation for the functional of Eq. (16.22) as

$$\delta E - \sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\epsilon_{\mu\nu}\delta\int\psi_\mu^*\psi_\nu = 0. \tag{16.25}$$

For the orthogonal wave functions ψ_μ this reduces to

$$\delta E - \sum_{\mu=1}^{N}\epsilon_\mu\delta\int\psi_\mu^*\psi_\mu = 0. \tag{16.26}$$

Variation with respect to the single-particle wave functions ψ_μ yields then

$$\sum_{\mu=1}^{N}\int \delta\psi_{\mu}^{*}\hat{h}_i\psi_{\mu}d\mathbf{x_i}+\frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\left[\int \delta\psi_{\mu}^{*}\psi_{\nu}^{*}\frac{1}{r_{ij}}\psi_{\mu}\psi_{\nu}d(\mathbf{x_i}\mathbf{x_j})-\int \delta\psi_{\mu}^{*}\psi_{\nu}^{*}\frac{1}{r_{ij}}\psi_{\nu}\psi_{\mu}d\mathbf{r}_i d\mathbf{r}_j\right]$$
$$+\sum_{\mu=1}^{N}\int \psi_{\mu}^{*}\hat{h}_i\delta\psi_{\mu}d\mathbf{r}_i+\frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\left[\int \psi_{\mu}^{*}\psi_{\nu}^{*}\frac{1}{r_{ij}}\delta\psi_{\mu}\psi_{\nu}d\mathbf{r}_i d\mathbf{r}_j-\int \psi_{\mu}^{*}\psi_{\nu}^{*}\frac{1}{r_{ij}}\psi_{\nu}\delta\psi_{\mu}d\mathbf{r}_i d\mathbf{r}_j\right]$$
$$-\sum_{\mu=1}^{N}E_{\mu}\int \delta\psi_{\mu}^{*}\psi_{\mu}d\mathbf{x_i}-\sum_{\mu=1}^{N}E_{\mu}\int \psi_{\mu}^{*}\delta\psi_{\mu}d\mathbf{r}_i=0. \tag{16.27}$$

Although the variations $\delta\psi$ and $\delta\psi^*$ are not independent, they may in fact be treated as such, so that the terms dependent on either $\delta\psi$ and $\delta\psi^*$ individually may be set equal to zero. To see this, simply replace the arbitrary variation $\delta\psi$ by $i\delta\psi$, so that $\delta\psi^*$ is replaced by $-i\delta\psi^*$, and combine the two equations. We thus arrive at the Hartree-Fock equations

$$\left[-\frac{1}{2}\nabla_i^2-\frac{Z}{r_i}+\sum_{\nu=1}^{N}\int \psi_{\nu}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_{\nu}(\mathbf{r}_j)d\mathbf{r}_j\right]\psi_{\mu}(\mathbf{x_i})$$
$$-\left[\sum_{\nu=1}^{N}\int \psi_{\nu}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_{\mu}(\mathbf{r}_j)d\mathbf{r}_j\right]\psi_{\nu}(\mathbf{r}_i)=\epsilon_{\mu}\psi_{\mu}(\mathbf{r}_i). \tag{16.28}$$

Notice that the integration $\int d\mathbf{r}_j$ implies an integration over the spatial coordinates $\mathbf{r_j}$ and a summation over the spin-coordinate of electron j.

The two first terms are the one-body kinetic energy and the electron-nucleus potential. The third or *direct* term is the averaged electronic repulsion of the other electrons. This term is identical to the Coulomb integral introduced in the simple perturbative approach to the helium atom. As written, the term includes the 'self-interaction' of electrons when $i=j$. The self-interaction is cancelled in the fourth term, or the *exchange* term. The exchange term results from our inclusion of the Pauli principle and the assumed determinantal form of the wave-function. The effect of exchange is for electrons of like-spin to avoid each other. A theoretically convenient form of the Hartree-Fock equation is to regard the direct and exchange operator defined through

$$V_{\mu}^{d}(\mathbf{r}_i)=\int \psi_{\mu}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_{\mu}(\mathbf{r}_j)d\mathbf{r}_j \tag{16.29}$$

and

$$V_{\mu}^{ex}(\mathbf{r}_i)g(\mathbf{r}_i)=\left(\int \psi_{\mu}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}g(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_{\mu}(\mathbf{r}_i), \tag{16.30}$$

respectively. The function $g(\mathbf{r}_i)$ is an arbitrary function, and by the substitution $g(\mathbf{r}_i)=\psi_{\nu}(\mathbf{r}_i)$ we get

$$V_{\mu}^{ex}(\mathbf{r}_i)\psi_{\nu}(\mathbf{r}_i)=\left(\int \psi_{\mu}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_{\nu}(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_{\mu}(\mathbf{r}_i). \tag{16.31}$$

We may then rewrite the Hartree-Fock equations as

$$H_i^{HF}\psi_{\nu}(\mathbf{r}_i)=\epsilon_{\nu}\psi_{\nu}(\mathbf{r}_i), \tag{16.32}$$

with

$$H_i^{HF} = h_i + \sum_{\mu=1}^{N} V_\mu^d(\mathbf{r}_i) - \sum_{\mu=1}^{N} V_\mu^{ex}(\mathbf{r}_i), \tag{16.33}$$

and where h_i is defined by equation (16.7).

16.2.2 Solving the Hartree-Fock Equations

Planned finished spring 2009.

16.3 Density functional theory

Hohenberg and Kohn proved that the total energy of a system including that of the many-body effects of electrons (exchange and correlation) in the presence of static external potential (for example, the atomic nuclei) is a unique functional of the charge density. The minimum value of the total energy functional is the ground state energy of the system. The electronic charge density which yields this minimum is then the exact single particle ground state energy.

It was then shown by Kohn and Sham that it is possible to replace the many electron problem by an exactly equivalent set of self consistent one electron equations. The total energy functional can be written as a sum of several terms:

for a fixed set of atomic nuclei. The first two terms are the classical Coulomb interaction between the electrons and ions and between electrons and other electrons respectively, both of which are simply functions of the electronic charge density. This equation is analogous to the Hartree method, but the term contains the effects of exchange and correlation and also the single particle kinetic energy:

In the different HF methods one works with large basis sets. This poses a problem for large systems. An alternative to the HF methods is *density functional theory* (DFT) [105]. DFT takes into account electron correlations but is less demanding computationally than for example CI and MP2.

The electronic energy E is said to be a *functional* of the electronic density, $E[\rho]$, in the sense that for a given function $\rho(r)$, there is a single corresponding energy. The *Hohenberg-Kohn theorem* [106] confirms that such a functional exists, but does not tell us the form of the functional. As shown by Kohn and Sham, the exact ground-state energy E of an N-electron system can be written as

$$E[\rho] = -\frac{1}{2}\sum_{i=1}^{N}\int \Psi_i^*(\mathbf{r_1})\nabla_1^2\Psi_i(\mathbf{r_1})d\mathbf{r_1} - \int \frac{Z}{r_1}\rho(\mathbf{r_1})d\mathbf{r_1} + \frac{1}{2}\int \frac{\rho(\mathbf{r_1})\rho(\mathbf{r_2})}{r_{12}}d\mathbf{r_1}d\mathbf{r_2} + E_{XC}[\rho] \tag{16.34}$$

with Ψ_i the *Kohn-Sham* (KS) *orbitals*. The ground-state charge density is given by

$$\rho(\mathbf{r}) = \sum_{i=1}^{N} |\Psi_i(\mathbf{r})|^2, \tag{16.35}$$

where the sum is over the occupied Kohn-Sham orbitals. The last term, $E_{EX}[\rho]$, is the *exchange-correlation energy* which in theory takes into account all non-classical electron-electron interaction. However, we do not know how to obtain this term exactly, and are forced to approximate it. The KS orbitals are found by solving the *Kohn-Sham equations*, which can be found by applying a variational principle to the electronic energy $E[\rho]$. This approach is similar to the one used for obtaining the HF equation in the previous section. The KS equations read

$$\left\{-\frac{1}{2}\nabla_1^2 - \frac{Z}{r_1} + \int \frac{\rho(\mathbf{r_2})}{r_{12}} d\mathbf{r_2} + V_{XC}(\mathbf{r_1})\right\} \Psi_i(\mathbf{r_1}) = \epsilon_i \Psi_i(\mathbf{r_1}) \tag{16.36}$$

where ϵ_i are the KS orbital energies, and where the *exchange-correlation potential* is given by

$$V_{XC}[\rho] = \frac{\delta E_{XC}[\rho]}{\delta \rho}. \tag{16.37}$$

The KS equations are solved in a self-consistent fashion. A variety of basis set functions can be used, and the experience gained in HF calculations are often useful. The computational time needed for a DFT calculation formally scales as the third power of the number of basis functions.

The main source of error in DFT usually arises from the approximate nature of E_{XC}. In the *local density approximation* (LDA) it is approximated as

$$E_{XC} = \int \rho(\mathbf{r}) \epsilon_{XC}[\rho(\mathbf{r})] d\mathbf{r}, \tag{16.38}$$

where $\epsilon_{XC}[\rho(\mathbf{r})]$ is the exchange-correlation energy per electron in a homogeneous electron gas of constant density. The LDA approach is clearly an approximation as the charge is not continuously distributed. To account for the inhomogeneity of the electron density, a nonlocal correction involving the gradient of ρ is often added to the exchange-correlation energy.

16.4 Variational Monte Carlo studies with importance sampling

Finished Spring 2009.

16.5 Exercises and projects

Project 16.1: Hartree-Fock and variational Monte Carlo

The aim of this project is to use the Variational Monte Carlo (VMC) method and evaluate the ground state energy of the atoms Helium, Beryllium and Neon.

We label r_1 the distance from electron 1 to the nucleus and similarly r_2 the distance between electron 2 and the nucleus. The contribution to the potential energy from the interactions between the electrons and the nucleus is

$$-\frac{2}{r_1} - \frac{2}{r_2}, \tag{16.39}$$

and if we add the electron-electron repulsion with $r_{12} = |\mathbf{r}_1 - \mathbf{r}_2|$, the total potential energy $V(r_1, r_2)$ is

$$V(r_1, r_2) = -\frac{2}{r_1} - \frac{2}{r_2} + \frac{1}{r_{12}}, \tag{16.40}$$

yielding the total Hamiltonian

$$\widehat{\mathbf{H}} = -\frac{\nabla_1^2}{2} - \frac{\nabla_2^2}{2} - \frac{2}{r_1} - \frac{2}{r_2} + \frac{1}{r_{12}}, \tag{16.41}$$

and Schrödinger's equation reads

$$\widehat{\mathbf{H}}\psi = E\psi. \tag{16.42}$$

All equations are in so-called atomic units. The distances r_i and r_{12} are dimensionless. To have energies in electronvolt you need to multiply all results with $2 \times E_0$, where $E_0 = 13.6$ eV. The experimental binding energy for helium in atomic units a.u. is $E_{\mathrm{He}} = -2.9037$ a.u..

1a) Set up the Hartree-Fock equations for the ground state of the Helium atom with two electrons occupying the hydrogen-like orbitals with quantum numbers $n = 1$, $s = 1/2$ and $l = 0$. There is no spin-orbit part in the two-body Hamiltonian. **Make sure to write these equations using atomic units**.

b) Write a program which solves the Hartree-Fock equations for the Helium atom. Use as input for the first iteration the hydrogen-like single-particle wave function, with analytical shape $\sim \exp(-\alpha r_i)$ where r_i represents the coordinates of electron i. The details of all equations which you need to program will be discussed during the lectures. Compare the results with those obtained using the hydrogen-like wave functions only.

c) Our next step is to perform a Variational Monte Carlo calculation of the ground state of the helium atom. In our first attempt we will use a brute force Metropolis sampling with a trial wave function which has the following form

$$\psi_T(\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_{12}}) = \exp(-\alpha(r_1 + r_2)) \exp\left(\frac{r_{12}}{2(1+\beta r_{12})}\right), \tag{16.43}$$

with α and β as variational parameters.

Your task is to perform a Variational Monte Carlo calculation using the Metropolis algorithm to compute the integral

$$\langle E \rangle = \frac{\int d\mathbf{r_1} d\mathbf{r_2} \psi_T^*(\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_{12}}) \widehat{\mathbf{H}}(\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_{12}}) \psi_T(\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_{12}})}{\int d\mathbf{r_1} d\mathbf{r_2} \psi_T^*(\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_{12}}) \psi_T(\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_{12}})}. \tag{16.44}$$

In performing the Monte Carlo analysis you should use blocking as a technique to make the statistical analysis of the numerical data. The code has to run in parallel. A code for doing a VMC calculation for the Helium atom can be found on the webpage of the course, see under programs.

d) Repeat the last step but use now importance sampling. Study the dependence of the results as function of the time step δt.

(e) Our final step is to replace the hydrogen-like orbits in Eq. (16.43) with those obtained from b) by solving the Hartree-Fock equations. This leads us to only one variational parameter, β. The calculations should include parallelization, blocking and importance sampling. There is no need to do brute force Metropolis sampling.

Compare the results with those from c) and the Hartree-Fock results from b). How important is the correlation part?

Here we will focus on the Neon and Beryllium atoms. It is convenient to make modules or classes of trial wave functions, both many-body wave functions and single-particle wave functions and the quantum numbers involved,such as spin, orbital momentum and principal quantum numbers.

The new item you need to pay attention to is the calculation of the Slater Determinant. This is an additional complication to your VMC calculations. If we stick to hydrogen-like wave functions, the trial wave function for Beryllium can be written as

$$\psi_T(\mathbf{r_1},\mathbf{r_2},\mathbf{r_3},\mathbf{r_4}) = Det\left(\phi_1(\mathbf{r_1}),\phi_2(\mathbf{r_2}),\phi_3(\mathbf{r_3}),\phi_4(\mathbf{r_4})\right)\prod_{i<j}^{4}\exp\left(\frac{r_{ij}}{2(1+\beta r_{ij})}\right), \quad (16.45)$$

where the Det is a Slater determinant and the single-particle wave functions are the hydrogen wave functions for the $1s$ and $2s$ orbitals. Their form within the variational ansatz are given by

$$\phi_{1s}(\mathbf{r_i}) = e^{-\alpha r_i}, \quad (16.46)$$

and

$$\phi_{2s}(\mathbf{r_i}) = (1-\alpha r_i/2)\, e^{-\alpha r_i/2}. \quad (16.47)$$

For Neon , the trial wave function can take the form

$$\psi_T(\mathbf{r_1},\mathbf{r_2},\dots,\mathbf{r_{10}}) = Det\left(\phi_1(\mathbf{r_1}),\phi_2(\mathbf{r_2}),\dots,\phi_{10}(\mathbf{r_{10}})\right)\prod_{i<j}^{10}\exp\left(\frac{r_{ij}}{2(1+\beta r_{ij})}\right), \quad (16.48)$$

In this case you need to include the $2p$ wave function as well. It is given as

$$\phi_{2p}(\mathbf{r_i}) = \alpha \mathbf{r_i} e^{-\alpha r_i/2}. \quad (16.49)$$

Observe that $r_i = \sqrt{r_{i_x}^2 + r_{i_y}^2 + r_{i_z}^2}$.

f) Set up the Hartree-Fock equations for the ground state of the Beryllium and Neon atoms with four and ten electrons, respectively, occupying the respective hydrogen-like orbitals. There is no spin-orbit part in the two-body Hamiltonian. Find also the experimental ground state energies using atomic units.

g) Solve the Hartree-Fock equations for the Beryllium and Neon atoms. Use again as input for the first iteration the hydrogen-like single-particle wave function. Compare the results with those obtained using the hydrogen-like wave functions only (first iteration).

(h) Write a function which sets up the Slater determinant for Beryllium. Neon is optional. Use the Hartree-Fock single-particle wave functions to set up the Slater determinant. You have only one variational parameter, β. Compute the ground state energies of Neon and Beryllium as you did for the Helium atom in e). The calculations should include parallelization, blocking and importance sampling. Compare the results with the Hartree-Fock results from g). How important is the correlation part? Is there a difference compared with Helium? Comment your results.

Chapter 17

Many-body methods for Bose-Einstein condensation

17.1 Bose-Einstein condensation in atoms

The spectacular demonstration of Bose-Einstein condensation (BEC) in gases of alkali atoms ^{87}Rb, ^{23}Na, ^{7}Li confined in magnetic traps [107–109] has led to an explosion of interest in confined Bose systems. Of interest is the fraction of condensed atoms, the nature of the condensate, the excitations above the condensate, the atomic density in the trap as a function of Temperature and the critical temperature of BEC, T_c. The extensive progress made up to early 1999 is reviewed by Dalfovo et al. [110].

A key feature of the trapped alkali and atomic hydrogen systems is that they are dilute. The characteristic dimensions of a typical trap for ^{87}Rb is $a_{h0} = (\hbar/m\omega_\perp)^{\frac{1}{2}} = 1 - 2 \times 10^4$ Å (Ref. [107]). The interaction between ^{87}Rb atoms can be well represented by its s-wave scattering length, a_{Rb}. This scattering length lies in the range $85 < a_{Rb} < 140a_0$ where $a_0 = 0.5292$ Å is the Bohr radius. The definite value $a_{Rb} = 100a_0$ is usually selected and for calculations the definite ratio of atom size to trap size $a_{Rb}/a_{h0} = 4.33 \times 10^{-3}$ is usually chosen [110]. A typical ^{87}Rb atom density in the trap is $n \simeq 10^{12} - 10^{14}$ atoms/cm^3 giving an inter-atom spacing $\ell \simeq 10^4$ Å. Thus the effective atom size is small compared to both the trap size and the inter-atom spacing, the condition for diluteness ($na^3_{Rb} \simeq 10^{-6}$ where $n = N/V$ is the number density). In this limit, although the interaction is important, dilute gas approximations such as the Bogoliubov theory [111], valid for small na^3 and large condensate fraction $n_0 = N_0/N$, describe the system well. Also, since most of the atoms are in the condensate (except near T_c), the Gross-Pitaevskii equation [112, 113] for the condensate describes the whole gas well. Effects of atoms excited above the condensate have been incorporated within the Popov approximation [114].

Most theoretical studies of Bose-Einstein condensates (BEC) in gases of alkali atoms confined in magnetic or optical traps have been conducted in the framework of the Gross-Pitaevskii (GP) equation [112, 113]. The key point for the validity of this description is the dilute condition of these systems, i.e., the average distance between the atoms is much larger than the range of the inter-atomic interaction. In this situation the physics is dominated by two-body collisions, well described in terms of the s-wave scattering length a. The crucial parameter defining the condition for diluteness is the gas parameter $x(\mathbf{r}) = n(\mathbf{r})a^3$, where $n(\mathbf{r})$ is the local density of the system. For low values of the average gas parameter $x_{av} \leq 10^{-3}$, the mean field Gross-Pitaevskii equation does an excellent job (see for example Ref. [110] for a review). However, in recent experiments, the local gas parameter may well exceed this value due to the possibility of tuning the scattering length in the presence of a Feshbach resonance [115].

Under such circumstances it is unavoidable to test the accuracy of the GP equation by performing microscopic calculations. If we consider cases where the gas parameter has been driven to a region were one can still have a universal regime, i.e., that the specific shape of the potential is unimportant, we may attempt to describe the system as dilute hard spheres whose diameter coincides with the scattering length. However, the value of x is such that the calculation of the energy of the uniform hard-sphere Bose gas would require to take into account the second term in the low-density expansion [116] of the energy density

$$\frac{E}{V} = \frac{2\pi n^2 a\hbar^2}{m}\left[1+\frac{128}{15}\left(\frac{na^3}{\pi}\right)^{1/2}+\cdots\right], \tag{17.1}$$

where m is the mass of the atoms treated as hard spheres. For the case of uniform systems, the validity of this expansion has been carefully studied using Diffusion Monte Carlo [117] and Hyper-Netted-Chain techniques [118].

The energy functional associated with the GP theory is obtained within the framework of the local-density approximation (LDA) by keeping only the first term in the low-density expansion of Eq. (17.1)

$$E_{\rm GP}[\Psi] = \int d\mathbf{r}\left[\frac{\hbar^2}{2m} \mid \nabla\Psi(\mathbf{r}) \mid^2 + V_{\rm trap}(\mathbf{r}) \mid \Psi \mid^2 + \frac{2\pi\hbar^2 a}{m} \mid \Psi \mid^4\right], \tag{17.2}$$

where

$$V_{\rm trap}(\mathbf{r}) = \frac{1}{2}m(\omega_\perp^2 x^2 + \omega_\perp^2 y^2 + \omega_z^2 z^2) \tag{17.3}$$

is the confining potential defined by the two angular frequencies $\omega_\perp$ and ω_z. The condensate wave function Ψ is normalized to the total number of particles.

By performing a functional variation of $E_{\rm GP}[\Psi]$ with respect to Ψ^* one finds the corresponding Euler-Lagrange equation, known as the Gross-Pitaevskii (GP) equation

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V_{\rm trap}(\mathbf{r}) + \frac{4\pi\hbar^2 a}{m} \mid \Psi \mid^2\right]\Psi = \mu\Psi, \tag{17.4}$$

where μ is the chemical potential, which accounts for the conservation of the number of particles. Within the LDA framework, the next step is to include into the energy functional of Eq. (17.2) the next term of the low density expansion of Eq. (17.1). The functional variation gives then rise to the so-called modified GP equation (MGP) [119]

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V_{\rm trap}(\mathbf{r}) + \frac{4\pi\hbar^2 a}{m} \mid \Psi \mid^2 \left(1+\frac{32a^{3/2}}{3\pi^{1/2}} \mid \Psi \mid\right)\right]\Psi = \mu\Psi. \tag{17.5}$$

The MGP corrections have been estimated in Ref. [119] in a cylindrical condensate in the range of the scattering lengths and trap parameters from the first JILA experiments with Feshbach resonances. These experiments took advantage of the presence of a Feshbach resonance in the collision of two ^{85}Rb atoms to tune their scattering length [115]. Fully microscopic calculations using a hard-spheres interaction have also been performed in the framework of Variational and Diffusion Monte Carlo methods [120–123].

17.2 Exercises and projects

Project 17.1: Bose-Einstein condensation of atoms

The aim of this project is to use the Variational Monte Carlo (VMC) method and evaluate the ground state energy of a trapped, hard sphere Bose gas for different numbers of particles with a specific trial wave function. See Ref. [64] for a discussion of VMC.

This wave function is used to study the sensitivity of condensate and non-condensate properties to the hard sphere radius and the number of particles. The trap we will use is a spherical (S) or an elliptical (E) harmonic trap in three dimensions given by

$$V_{ext}(\mathbf{r}) = \begin{cases} \frac{1}{2}m\omega_{ho}^2 r^2 & (S) \\ \frac{1}{2}m[\omega_{ho}^2(x^2+y^2)+\omega_z^2 z^2] & (E) \end{cases} \tag{17.6}$$

where (S) stands for symmetric and

$$H = \sum_i^N \left(\frac{-\hbar^2}{2m}\nabla_i^2 + V_{ext}(\mathbf{r}_i)\right) + \sum_{i<j}^N V_{int}(\mathbf{r}_i,\mathbf{r}_j), \tag{17.7}$$

as the two-body Hamiltonian of the system. Here ω_{ho}^2 defines the trap potential strength. In the case of the elliptical trap, $V_{ext}(x,y,z)$, $\omega_{ho} = \omega_\perp$ is the trap frequency in the perpendicular or xy plane and ω_z the frequency in the z direction. The mean square vibrational amplitude of a single boson at $T = 0K$ in the trap (17.6) is $< x^2 >= (\hbar/2m\omega_{ho})$ so that $a_{ho} \equiv (\hbar/m\omega_{ho})^{\frac{1}{2}}$ defines the characteristic length of the trap. The ratio of the frequencies is denoted $\lambda = \omega_z/\omega_\perp$ leading to a ratio of the trap lengths $(a_\perp/a_z) = (\omega_z/\omega_\perp)^{\frac{1}{2}} = \sqrt{\lambda}$.

We represent the inter boson interaction by a pairwise, hard core potential

$$V_{int}(|\mathbf{r}_i - \mathbf{r}_j|) = \begin{cases} \infty & |\mathbf{r}_i - \mathbf{r}_j| \le a \\ 0 & |\mathbf{r}_i - \mathbf{r}_j| > a \end{cases} \tag{17.8}$$

where a is the hard core diameter of the bosons. Clearly, $V_{int}(|\mathbf{r}_i - \mathbf{r}_j|)$ is zero if the bosons are separated by a distance $|\mathbf{r}_i - \mathbf{r}_j|$ greater than a but infinite if they attempt to come within a distance $|\mathbf{r}_i - \mathbf{r}_j| \le a$.

Our trial wave function for the ground state with N atoms is given by

$$\Psi_T(\mathbf{R}) = \Psi_T(\mathbf{r}_1, \mathbf{r}_2, \ldots \mathbf{r}_N, \alpha, \beta) = \prod_i g(\alpha,\beta,\mathbf{r}_i) \prod_{i<j} f(a, |\mathbf{r}_i - \mathbf{r}_j|), \tag{17.9}$$

where α and β are variational parameters. The single-particle wave function is proportional to the harmonic oscillator function for the ground state, i.e.,

$$g(\alpha,\beta,\mathbf{r}_i) = \exp\left[-\alpha(x_i^2 + y_i^2 + \beta z_i^2)\right]. \tag{17.10}$$

For spherical traps we have $\beta = 1$ and for non-interacting bosons ($a = 0$) we have $\alpha = 1/2a_{ho}^2$. The correlation wave function is

$$f(a, |\mathbf{r}_i - \mathbf{r}_j|) = \begin{cases} 0 & |\mathbf{r}_i - \mathbf{r}_j| \le a \\ (1 - \frac{a}{|\mathbf{r}_i - \mathbf{r}_j|}) & |\mathbf{r}_i - \mathbf{r}_j| > a. \end{cases} \tag{17.11}$$

a) Find analytic expressions for the local energy

$$E_L(\mathbf{R}) = \frac{1}{\Psi_T(\mathbf{R})} H \Psi_T(\mathbf{R}), \tag{17.12}$$

for the above trial wave function of Eq. (17.9). Compute also the analytic expression for the drift force to be used in importance sampling

$$F = \frac{2\nabla\Psi_T}{\Psi_T}. \tag{17.13}$$

The tricky part is to find an analytic expressions for the derivative of the trial wave function

$$\frac{1}{\Psi_T(\mathbf{R})}\sum_i^N \nabla_i^2 \Psi_T(\mathbf{R}),$$

for the above trial wave function of Eq. (17.9). We rewrite

$$\Psi_T(\mathbf{R}) = \Psi_T(\mathbf{r}_1, \mathbf{r}_2, \dots \mathbf{r}_N, \alpha, \beta) = \prod_i g(\alpha, \beta, \mathbf{r}_i) \prod_{i<j} f(a, |\mathbf{r}_i - \mathbf{r}_j|),$$

as

$$\Psi_T(\mathbf{R}) = \prod_i g(\alpha, \beta, \mathbf{r}_i) e^{\sum_{i<j} u(r_{ij})}$$

where we have defined $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ and

$$f(r_{ij}) = e^{\sum_{i<j} u(r_{ij})},$$

and in our case

$$g(\alpha, \beta, \mathbf{r}_i) = e^{-\alpha(x_i^2 + y_i^2 + z_i^2)} = \phi(\mathbf{r}_i).$$

The first derivative becomes

$$\nabla_k \Psi_T(\mathbf{R}) = \nabla_k \phi(\mathbf{r}_k) \left[\prod_{i \neq k} \phi(\mathbf{r}_i)\right] e^{\sum_{i<j} u(r_{ij})} + \prod_i \phi(\mathbf{r}_i) e^{\sum_{i<j} u(r_{ij})} \sum_{j \neq k} \nabla_k u(r_{ij})$$

We leave it as an exercise for the reader to find the expression for the sceond derivative. The final expression is

$$\frac{1}{\Psi_T(\mathbf{R})} \nabla_k^2 \Psi_T(\mathbf{R}) = \frac{\nabla_k^2 \phi(\mathbf{r}_k)}{\phi(\mathbf{r}_k)} + \frac{\nabla_k \phi(\mathbf{r}_k)}{\phi(\mathbf{r}_k)} \left(\sum_{j \neq k} \frac{\mathbf{r}_k}{r_k} u'(r_{ij})\right) +$$

$$\sum_{ij \neq k} \frac{(\mathbf{r}_k - \mathbf{r}_i)(\mathbf{r}_k - \mathbf{r}_j)}{r_{ki} r_{kj}} u'(r_{ki}) u'(r_{kj}) + \sum_{j \neq k} \left(u''(r_{kj}) + \frac{2}{r_{kj}} u'(r_{kj})\right)$$

You need to get the analytic expression for this expression using the harmonic oscillator wave functions and the correlation term defined in the project.

b) Write a Variational Monte Carlo program which uses standard Metropolis sampling and compute the ground state energy of a spherical harmonic oscillator ($\beta = 1$) with no interaction. Use natural units and make an analysis of your calculations using both the analytic expression for the local energy and a numerical calculation of the kinetic energy using numerical derivation. Compare the CPU time difference. You should also parallelize your code. The only variational parameter is α. Perform these calculations for $N = 10$, 100 and 500 atoms. Compare your results with the exact answer.

c) We turn now to the elliptic trap with a hard core interaction. We fix, as in Refs. [120, 124] $a/a_{ho} = 0.0043$. Introduce lengths in units of a_{ho}, $r \to r/a_{ho}$ and energy in units of $\hbar\omega_{ho}$. Show then that the original Hamiltonian can be rewritten as

$$H = \sum_{i=1}^N \frac{1}{2}\left(-\nabla_i^2 + x_i^2 + y_i^2 + \gamma^2 z_i^2\right) + \sum_{i<j} V_{int}(|\mathbf{r}_i - \mathbf{r}_j|). \qquad (17.14)$$

What is the expression for γ? Choose the initial value for $\beta = \gamma = 2.82843$ and set up a VMC program which computes the ground state energy using the trial wave function of Eq. (17.9). using only α as variational parameter. Use standard Metropolis sampling and vary the parameter α in order to find a minimum. Perform the calculations for $N = 10, 50$ and $N = 100$ and compare your results to those from the ideal case in the previous exercise. In actual calculations employing e.g., the Metropolis algorithm, all moves are recast into the chosen simulation cell with periodic boundary conditions. To carry out consistently the Metropolis moves, it has to be assumed that the correlation function has a range shorter than $L/2$. Then, to decide if a move of a single particle is accepted or not, only the set of particles contained in a sphere of radius $L/2$ centered at the referred particle have to be considered.

d) We repeat exercise c), but now we replace the brute force Metropolis algorithm with importance sampling based on the Fokker-Planck and the Langevin equations. Discuss your results and comment on eventual differences between importance sampling and brute force sampling.

Your code should reproduce the results of Refs. [120, 124].

Project 17.2: Liquid ^{4}He

Liquid ^{4}He is an example of a so-called extended system, with an infinite number of particles. The density of the system varies from dilute to extremely dense. It is fairly obvious that we cannot attempt a simulation with such degrees of freedom. There are however ways to circumvent this problem. The usual way of dealing with such systems, using concepts from statistical Physics, consists in representing the system in a simulation cell with e.g., periodic boundary conditions, as we did for the Ising model. If the cell has length L, the density of the system is determined by putting a given number of particles N in a simulation cell with volume L^3. The density becomes then $\rho = N/L^3$.

In general, when dealing with such systems of many interacting particles, the interaction itself is not known analytically. Rather, we will have to rely on parametrizations based on e.g., scattering experiments in order to determine a parametrization of the potential energy. The interaction between atoms and/or molecules can be either repulsive or attractive, depending on the distance R between two atoms or molecules. One can approximate this interaction as

$$V(R) = -\frac{A}{R^m} - \frac{B}{R^n}, \tag{17.15}$$

where m, n are some integers and A, B constans with dimension energy and length, and with units in e.g., eVnm. The constants A, B and the integers m, n are determined by the constraints that we wish to reproduce both scattering data and the binding energy of say a given molecule. It is thus an example of a parametrized interaction, and does not enjoy the status of being a fundamental interaction such as the Coulomb interaction does.

A well-known parametrization is the so-called Lennard-Jones potential

$$V_{LJ}(R) = 4\epsilon \left\{ \left(\frac{\sigma}{R}\right)^{12} - \left(\frac{\sigma}{R}\right)^{6} \right\}, \tag{17.16}$$

where $\epsilon = 8.79 \times 10^{-4}$ eV and $\sigma = 0.256$ nm for helium atoms. Fig. 17.1 displays this interaction model. The interaction is both attractive and repulsive and exhibits a minimum at r_0. The reason why we have repulsion at small distances is that the electrons in two different helium atoms start repelling each other. In addition, the Pauli exclusion principle forbids two electrons to have the same set of quantum numbers.

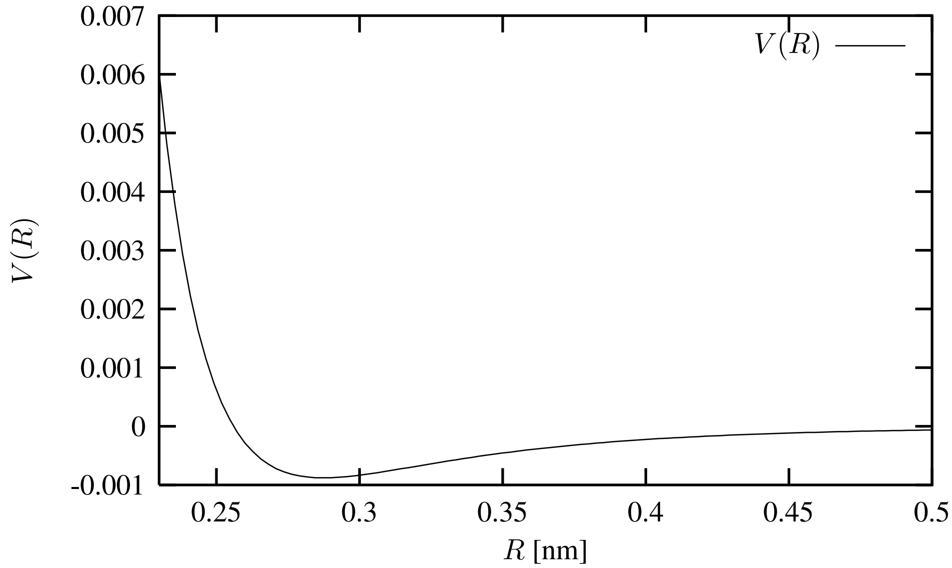


Figure 17.1: Plot for the Van der Waals interaction between helium atoms. The equilibrium position is $r_0 = 0.287$ nm.

Let us now assume that we have a simple trial wave function of the form

$$\psi_T(\mathbf{R}) = \prod_{i<j}^{N} f(r_{ij}), \tag{17.17}$$

where we assume that the correlation function $f(r_{ij})$ can be written as

$$f(r_{ij}) = e^{-\frac{1}{2}(b/r_{ij})^n}, \tag{17.18}$$

with b being the only variational parameter. Can we fix the value of n using the 'cusp'-conditions discussed in connection with the helium atom? We see from the form of the potential, that it diverges at small interparticle distances. Since the energy is finite, it means that the kinetic energy term has to cancel this divergence at small r. Let us assume that electrons i and j are very close to each other. For the sake of convenience, we replace $r_{ij} = r$. At small r we require then that

$$-\frac{1}{f(r)}\nabla^2 f(r) + V(r) = 0. \tag{17.19}$$

In the limit $r \to 0$ we have

$$-\frac{n^2 b^n}{4r^{2n+2}} + \left(\frac{\sigma}{r}\right)^{12} = 0, \tag{17.20}$$

resulting in $n = 5$ and thus

$$f(r_{ij}) = e^{-\frac{1}{2}(b/r_{ij})^5}, \tag{17.21}$$

with

$$\psi_T(\mathbf{R}) = \prod_{i<j}^{N} e^{-\frac{1}{2}(b/r_{ij})^5}, \tag{17.22}$$

as trial wave function. We can rewrite the above equation as

$$\psi_T(\mathbf{R}) = e^{-\frac{1}{2}\sum_{i<j}^{N}(b/r_{ij})^5} = e^{-\frac{1}{2}\sum_{i<j}^{N}u(r_{ij})}, \tag{17.23}$$

with

$$u(r_{ij}) = (b/r_{ij})^5 \,.$$

For this variational wave function, the analytical expression for the local energy is rather simple. The tricky part comes again from the kinetic energy given by

$$-\frac{1}{\psi_T(\mathbf{R})}\nabla^2\psi_T(\mathbf{R}). \tag{17.24}$$

It is possible to show, after some tedious algebra, that

$$-\frac{1}{\psi_T(\mathbf{R})}\nabla^2\psi_T(\mathbf{R}) = -\sum_{k=1}^{N}\frac{1}{\psi_T(\mathbf{R})}\nabla_k^2\psi_T(\mathbf{R}) = -10b^5\sum_{i<k}^{N}\frac{1}{r_{ik}^7}. \tag{17.25}$$

Chapter 18

Modelling Phase Transitions in Statistical Physics

Chapter 19

Quantum Information Theory and Quantum Algorithms

Part III

Programs and additional notes on C++, MPI and Fortran 90/95

Appendix A

Additional C++, Python and Fortran programming features

A.1 Classes, templates and Blitz++

In Fortran a vector or matrix start with 1, but it is easy to change a vector so that it starts with zero or even a negative number. If we have a double precision Fortran vector which starts at -10 and ends at 10, we could declare it as **REAL**(**KIND**=8):: vector(-10:10). Similarly, if we want to start at zero and end at 10 we could write **REAL**(**KIND**=8):: vector(0:10). We have also seen that Fortran allows us to write a matrix addition $\mathbf{A} = \mathbf{B} + \mathbf{C}$ as A = B + C. This means that we have overloaded the addition operator so that it translates this operation into two loops and an addition of two matrix elements $a_{ij} = b_{ij} + c_{ij}$.

The way the matrix addition is written is very close to the way we express this relation mathematically. The benefit for the programmer is that our code is easier to read. Furthermore, such a way of coding makes it more likely to spot eventual errors as well.

In Ansi C and C++ arrays start by default from $i = 0$. Moreover, if we wish to add two matrices we need to explicitely write out the two loops as

```
for(i=0 ; i < n ; i++) {
   for(j=0 ; j < n ; j++) {
      a[i][j]=b[i][j]+c[i][j]
   }
}
```

However, the strength of C++ over programming languages like C and Fortran 77 is the possibility to define new data types, tailored to some particular problem. Via new data types and overloading of operations such as addition and subtraction, we can easily define sets of operations and data types which allow us to write a matrix addition in exactly the same way as we would do in Fortran. We could also change the way we declare a C++ matrix elements a_{ij}, from $a[i][j]$ to say $a(i,j)$, as we would do in Fortran. Similarly, we could also change the default range from $0 : n-1$ to $1 : n$.

To achieve this we need to introduce two important entities in C++ programming, classes and templates.

The function and class declarations are fundamental concepts within C++. Functions are abstractions which encapsulate an algorithm or parts of it and perform specific tasks in a program. We have already met several examples on how to use functions. Classes can be defined as abstractions which encapsulate data and operations on these data. The data can be very complex data structures and the class can contain particular functions which operate on these data. Classes allow therefore for a higher level of abstraction in computing. The elements (or components) of the data type are the class data members, and the procedures are the class member functions.

Classes are user-defined tools used to create multi-purpose software which can be reused by other classes or functions. These user-defined data types contain data (variables) and functions operating on the data.

A simple example is that of a point in two dimensions. The data could be the x and y coordinates of a given point. The functions we define could be simple read and write functions or the possibility to compute the distance between two points.

The two examples we elaborate on below[1] demonstrate most of the features of classes. We develop first a class called Complex which allows us to perform various operations on complex variables. In appendix A we extend our discussion of classes to define a class vector_operations which allows us to perform various operations on a user-specified one-dimesional array, from declarations of a vector to mathematical operations such as additions of vectors.

The classes we define are easy to use in other codes and/or other classes and many of the details which would be present in C (or Fortran 77) codes are hidden inside the class. The reuse of a well-written and functional class is normally rather simple. However, to write a given class is often complicated, especially if we deal with complicated matrix operations. In this text we will rely on ready-made classes in C++ for dealing with matrix operations. We have chosen to use the Blitz++ library, discussed below. This library hides for us many low-level operations with matrices and vectors, such as matrix-vector multiplications or allocation and deallocation of memory. Such libraries make it then easier to build our own high-level classes out of well-tested lower-level classes.

The way we use classes in this text is close to the MODULE data type in Fortran and we provide some simple demonstrations of the latter as well in appendix A.

In this text we will mainly use classes to encapsulate specific operations, but will not use the full power such as inheritance and other object-oriented programming concepts. For examples of the latter see Refs. [19–21]

A.1.1 The Complex class

As remarked in chapter 2, C++ has a class complex in its standard template library (STL). The standard usage in a given function could then look like

```
// Program to calculate addition and multiplication of two complex numbers
using namespace std;
#include <iostream>
#include <cmath>
#include <complex>
int main()
{
  complex<double> x(6.1,8.2), y(0.5,1.3);
  // write out x+y
  cout << x + y << x*y  << endl;
  return 0;
}
```

where we add and multiply two complex numbers $x = 6.1 + \imath 8.2$ and $y = 0.5 + \imath 1.3$ with the obvious results $z = x + y = 6.6 + \imath 9.5$ and $z = x \cdot y = -7.61 + \imath 12.03$. In Fortran we would declare the above

[1] These examples are taken from the course INF-VERK3830, see `http://heim.ifi.uio.no/~hpl/INF-VERK4830/` for more information.

variables as COMPLEX(DPC):: x(6.1,8.2), y (0.5,1.3) .

The library Blitz++ includes an extension of the complex class to operations on vectors, matrices and higher-dimensional arrays. We recommend the use of Blitz++ when you develop your own codes. However, writing a complex class yourself is a good pedagogical exercise.

We proceed by splitting our task in three files.

– We define first a header file complex.h which contains the declarations of the class. The header file contains the class declaration (data and functions), declaration of stand-alone functions, and all inlined functions, starting as follows

```
#ifndef Complex_H
#define Complex_H
//   various include statements and definitions
#include <iostream>          // Standard ANSI-C++ include files
#include <new>
#include ....

class Complex
{...
definition of variables and their character
};
//   declarations of various functions used by the class
...
#endif
```

– Next we provide a file complex.cpp where the code and algorithms of different functions (except inlined functions) declared within the class are written. The files complex.h and complex.cpp are normally placed in a directory with other classes and libraries we have defined.

– Finally,we discuss here an example of a main program which uses this particular class. An example of a program which uses our complex class is given below. In particular we would like our class to perform tasks like declaring complex variables, writing out the real and imaginary part and performing algebraic operations such as adding or multiplying two complex numbers.

```
#include "Complex.h"
...  other include and declarations
int main ()
{
  Complex a(0.1,1.3);    // we declare a complex variable a
  Complex b(3.0), c(5.0,-2.3);  // we declare  complex variables b and
      c
  Complex d = b;          //  we declare  a new complex variable d
  cout << "d=" << d << ", a=" << a << ", b=" << b << endl;
  d = a*c + b/a;  //   we add, multiply and divide two complex numbers
  cout << "Re(d)=" << d.Re() << ", Im(d)=" << d.Im() << endl;  // write
       out of the real and imaginary parts
}
```

We include the header file complex.h and define four different complex variables. These are $a = 0.1 + \imath 1.3$, $b = 3.0 + \imath 0$ (note that if you don't define a value for the imaginary part this is set to zero), $c = 5.0 - \imath 2.3$ and $d = b$. Thereafter we have defined standard algebraic operations and the member functions of the class which allows us to print out the real and imaginary part of a given variable.

To achieve these features, let us see how we could define the complex class. In C++ we could define a complex class as follows

```
class Complex
{
private:
   double re, im; // real and imaginary part
public:
   Complex ();                              // Complex c;
   Complex (double re, double im = 0.0); // Definition of a complex variable
       ;
   Complex (const Complex& c);               // Usage: Complex c(a);    //
       equate two complex variables
   Complex& operator= (const Complex& c); // c = a;    // equate two complex
        variables, same as previous
  ~Complex () {}                             // destructor
   double    Re () const;          // double real_part = a.Re();
   double    Im () const;          // double imag_part = a.Im();
   double    abs () const;         // double m = a.abs(); // modulus
   friend Complex operator+ (const Complex&  a, const Complex& b);
   friend Complex operator- (const Complex&  a, const Complex& b);
   friend Complex operator* (const Complex&  a, const Complex& b);
   friend Complex operator/ (const Complex&  a, const Complex& b);
};
```

The class is defined via the statement **class** Complex. We must first use the key word **class**, which in turn is followed by the user-defined variable name Complex. The body of the class, data and functions, is encapsulated within the parentheses {...};.

Data and specific functions can be private, which means that they cannot be accessed from outside the class. This means also that access cannot be inherited by other functions outside the class. If we use **protected** instead of **private**, then data and functions can be inherited outside the class. The key word **public** means that data and functions can be accessed from outside the class. Here we have defined several functions which can be accessed by functions outside the class. The declaration **friend** means that stand-alone functions can work on privately declared variables of the type (re, im). Data members of a class should be declared as private variables.

The first public function we encounter is a so-called constructor, which tells how we declare a variable of type Complex and how this variable is initialized. We have chose three possibilities in the example above:

1. A declaration like Complex c; calls the member function Complex() which can have the following implementation

   ```
   Complex:: Complex ()   { re = im = 0.0; }
   ```

   meaning that it sets the real and imaginary parts to zero. Note the way a member function is defined. The constructor is the first function that is called when an object is instantiated.

2. Another possibility is

   ```
   Complex:: Complex ()   {}
   ```

which means that there is no initialization of the real and imaginary parts. The drawback is that a given compiler can then assign random values to a given variable.

3. A call like Complex a (0.1,1.3) ; means that we could call the member function Complex(**double**, **double**)as

```
Complex:: Complex (double re_a, double im_a)
{ re = re_a; im = im_a; }
```

The simplest member function are those we defined to extract the real and imaginary part of a variable. Here you have to recall that these are private data, that is they invisible for users of the class. We obtain a copy of these variables by defining the functions

```
double Complex:: Re () const { return re; }} //  getting the real part
double Complex:: Im () const { return im; }  //   and the imaginary part
\end{lstlistingline}
Note that we have introduced  the declaration  \lstinline{const}.  What
    does it mean?
This declaration means that a varibale cannot be changed within  a called
    function.
If we define a variable as
\lstinline{const double p = 3;} and then try to change its value, we will
    get an error when we
compile our program. This means that constant arguments in functions cannot
    be changed.
\begin{lstlisting}
// const arguments (in functions) cannot be changed:
void myfunc (const Complex& c)
{ c.re = 0.2; /* ILLEGAL!! compiler error... */  }
```

If we declare the function and try to change the value to 0.2, the compiler will complain by sending an error message. If we define a function to compute the absolute value of complex variable like

```
double Complex:: abs ()  { return sqrt(re*re + im*im);}
```

without the constant declaration and define thereafter a function myabs as

```
double myabs (const Complex& c)
{ return c.abs(); }   // Not ok because c.abs() is not a const func.
```

the compiler would not allow the c.abs() call in myabs since Complex::abs is not a constant member function. Constant functions cannot change the object's state. To avoid this we declare the function abs as

```
double Complex:: abs () const { return sqrt(re*re + im*im); }
```

Overloading operators

C++ (and Fortran) allow for overloading of operators. That means we can define algebraic operations on for example vectors or any arbitrary object. As an example, a vector addition of the type $\mathbf{c} = \mathbf{a} + \mathbf{b}$ means that we need to write a small part of code with a for-loop over the dimension of the array. We would rather like to write this statement as c = a+b; as this makes the code much more readable and close to eventual equations we want to code. To achieve this we need to extend the definition of operators.

Let us study the declarations in our complex class. In our main function we have a statement like d = b;, which means that we call d.**operator**= (b) and we have defined a so-called assignment operator as a part of the class defined as

```
Complex& Complex:: operator= (const Complex& c)
{
   re = c.re;
   im = c.im;
   return *this;
}
```

With this function, statements like Complex d = b; or Complex d(b); make a new object d, which becomes a copy of b. We can make simple implementations in terms of the assignment

```
Complex:: Complex (const Complex& c)
{ *this = c; }
```

which is a pointer to "this object", $*$**this** is the present object, so $*$**this** = c; means setting the present object equal to c, that is **this**$-$>**operator**= (c);.

The meaning of the addition operator $+$ for Complex objects is defined in the function Complex **operator**+ (**const** Complex& a, **const** Complex& b); // *a+b* The compiler translates c = a + b; into c = **operator** + (a, b);. Since this implies the call to function, it brings in an additional overhead. If speed is crucial and this function call is performed inside a loop, then it is more difficult for a given compiler to perform optimizations of a loop. The solution to this is to inline functions. We discussed inlining in chapter 2. Inlining means that the function body is copied directly into the calling code, thus avoiding calling the function. Inlining is enabled by the inline keyword

```
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex (a.re + b.re, a.im + b.im); }
```

Inline functions, with complete bodies must be written in the header file complex.h. Consider the case c = a + b; that is, c.**operator**= (**operator**+ (a,b)); If **operator**+, **operator**= and the constructor Complex(r,i) all are inline functions, this transforms to

```
c.re = a.re + b.re;
c.im = a.im + b.im;
```

by the compiler, i.e., no function calls

The stand-alone function **operator**+ is a friend of the Complex class

```
class Complex
{
   ...
   friend Complex operator+ (const Complex& a, const Complex& b);
   ...
};
```

so it can read (and manipulate) the private data parts re and im via

```
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex (a.re + b.re, a.im + b.im); }
```

Since we do not need to alter the re and im variables, we can get the values by Re() and Im(), and there is no need to be a friend function

```
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex (a.Re() + b.Re(), a.Im() + b.Im()); }
```

The multiplication functionality can now be extended to imaginary numbers by the following code

```
inline Complex operator* (const Complex& a, const Complex& b)
{
  return Complex(a.re*b.re - a.im*b.im, a.im*b.re + a.re*b.im);
}
```

It will be convenient to inline all functions used by this operator. To inline the complete expression a*b;, the constructors and **operator**= must also be inlined. This can be achieved via the following piece of code

```
inline Complex:: Complex () { re = im = 0.0; }
inline Complex:: Complex (double re_, double im_)
{ ... }
inline Complex:: Complex (const Complex& c)
{ ... }
inline Complex:: operator= (const Complex& c)
{ ... }
// e, c, d are complex
e = c*d;
// first compiler translation:
e.operator= (operator* (c,d));
// result of nested inline functions
// operator=, operator*, Complex(double,double=0):
e.re = c.re*d.re - c.im*d.im;
e.im = c.im*d.re + c.re*d.im;
```

The definitions **operator**– and **operator**/ follow the same set up.

Finally, if we wish to write to file or another device a complex number using the simple syntax cout << c;, we obtain this by defining the effect of << for a Complex object as

```
ostream& operator<< (ostream& o, const Complex& c)
{ o << "(" << c.Re() << "," << c.Im() << ") "; return o;}
```

Templates

The reader may have noted that all variables and some of the functions defined in our class are declared as doubles. What if we wanted to make a class which takes integers or floating point numbers with single precision? A simple way to achieve this is copy and paste our class and replace **double** with for example **int**.

C++ allows us to do this automatically via the usage of templates, which are the C++ constructs for parameterizing parts of classes. Class templates is a template for producing classes. The declaration consists of the keyword **template** followed by a list of template arguments enclosed in brackets. We can therefore make a more general class by rewriting our original example as

```
template<class T>
class Complex
{
private:
   T re, im; // real and imaginary part
```

```
public:
   Complex ();                               // Complex c;
   Complex (T re, T im = 0); // Definition of a complex variable;
   Complex (const Complex& c);               // Usage: Complex c(a);    //
      equate two complex variables
   Complex& operator= (const Complex& c); // c = a;   //  equate two complex
       variables, same as previous
  ~Complex () {}                         // destructor
   T    Re () const;          // T real_part = a.Re();
   T    Im () const;          // T imag_part = a.Im();
   T    abs () const;         // T m = a.abs(); // modulus
   friend Complex operator+ (const Complex&  a, const Complex& b);
   friend Complex operator- (const Complex&  a, const Complex& b);
   friend Complex operator* (const Complex&  a, const Complex& b);
   friend Complex operator/ (const Complex&  a, const Complex& b);
};
```

What it says is that Complex is a parameterized type with T as a parameter and T has to be a type such as double or float. The class complex is now a class template and we would define variables in a code as

```
Complex<double> a(10.0,5.1);
Complex<int> b(1,0);
```

Member functions of our class are defined by preceding the name of the function with the **template** keyword. Consider the function we defined as Complex:: Complex (**double** re_a, **double** im_a). We would rewrite this function as

```
template<class T>
Complex<T>:: Complex (T re_a, T im_a)
{ re = re_a; im = im_a; }
```

The member functions are otherwise defined following ordinary member function definitions.

To write a class like the above is rather straightforward. The class for handling one-dimensional arrays, presented in appendix A shows some of the additional possibilities which C++ offers. However, it can be rather difficult to write good classes for handling matrices or more complex objects. For such applications we recommend therefore the usage of ready-made libraries like Blitz++

Blitz++ `http://www.oonumerics.org/blitz/` is a C++ library whose two main goals are to improve the numerical efficiency of C++ and to extend the conventional dense array model to incorporate new and useful features. Some examples of such extensions are flexible storage formats, tensor notation and index placeholders. It allows you also to write several operations involving vectors and matrices in a simple and clear (from a mathematical point of view) way. The way you would code the addition of two matrices looks very similar to the way it is done in Fortran. The C++ programming language offers many features useful for tackling complex scientific computing problems: inheritance, polymorphism, generic programming, and operator overloading are some of the most important. Unfortunately, these advanced features came with a hefty performance pricetag: until recently, C++ lagged behind Fortran's performance by anywhere from 20% to a factor of ten. It was not uncommon to read in textbooks on high-performance computing that if performance matters, then one should resort to Fortran, preferentially Fortran 77. As a result, untill very recently, the adoption of C++ for scientific computing has been slow. This has changed quite a lot in the last years and modern C++ compilers with numerical libraries have improved the situation considerably. Recent benchmarks show C++ encroaching steadily on Fortran's high-performance monopoly, and for some benchmarks, C++ is even faster than Fortran! These results are being obtained not through better optimizing compilers, preprocessors, or language extensions, but through the use of template techniques. By using templates cleverly, optimizations such as loop fusion, unrolling, tiling, and algorithm specialization can be performed automatically at compile time.

The features of Blitz++ which are useful for our studies are the dynamical allocation of vectors and matrices and algebraic operations on these objects. In particular, if you access the Blitz++ webpage at `http://www.oonumerics.org/blitz/`, we recommend that you study chapters two and three.

In this section we discuss several applications of the Blitz++ library and demonstrate the benefits when handling arrays and mathematical expressions involving arrays.

At `http://folk.uio.no/mhjensen/fys3150/2005/programs/blitz` you will find examples of makefiles, simple examples like those discussed here and the C++ library which contains the algorithms discussed in this text. You can choose whether you want to employ Blitz++ fully or just use the more old-fashioned C++ codes.

The example included here shows some of the versatility of Blitz++ when handling matrices. Note that you need to define the path where you have store Blitz++. We recommend that you study the examples available at the Blitz++ web page and the examples which follow this text.

As an example, a float matrix is defined simply as `Array<float,2> A(r,r);`. As the example shows we can even change the range of the matrix from the standard which starts at 0 and ends at $n-1$ to one which starts at 1 and ends at n. This can be useful if you deal with matrices from a Fortran code or if you wish to code your matrix operation following the way you index the matrix elements.

You can also easily initialise to zero your matrix by simply writing A=0.;. Note also the way you can fill in matrix elements and print out elements using one single statement, instead of for example two for loops. The following example illustrates some of these features.

`http://folk.uio.no/mhjensen/fys3150/2005/blitz/examples/example1.cpp`

```cpp
//      Simple test case of matrix operations
//      using Blitz++
#include <blitz/array.h>
#include <iostream>
using namespace std;
using namespace blitz;

int main()
{
  // Create two 4x4 arrays.  We want them to look like matrices, so
  // we'll make the valid index range 1..4 (rather than 0..3 which is
  // the default).

  Range r(1,4);
  Array<float,2> A(r,r), B(r,r);

  // The first will be a Hilbert matrix:
  //
  // a    =   1
  //  ij    -----
  //        i+j-1
  //
  // Blitz++ provides a set of types { firstIndex, secondIndex, ... }
  // which act as placeholders for indices.  These can be used directly
  // in expressions.  For example, we can fill out the A matrix like this:
```

```
  firstIndex i;     // Placeholder for the first index
  secondIndex j;    // Placeholder for the second index

  A = 1.0 / (i+j-1);
 cout << "A = " << A << endl;
  // Now the A matrix has each element equal to a_ij = 1/(i+j-1).
  //
  // The matrix B will be the permutation matrix
  //
  // [ 0 0 0 1 ]
  // [ 0 0 1 0 ]
  // [ 0 1 0 0 ]
  // [ 1 0 0 0 ]
  //
  // Here are two ways of filling out B:

  B = (i == (5-j));           // Using an equation -- a bit cryptic

  cout << "B = " << B << endl;

  B = 0, 0, 0, 1,             // Using an initializer list
    0, 0, 1, 0,
    0, 1, 0, 0,
    1, 0, 0, 0;

  cout << "B = " << B << endl;
}
```

More examples are discussed in appendix A.

A.2 The vector class

Our next next example is a very simple class to handle one-dimensional arrays. It demonstrates again many aspects of C++ programming. However, most likely you will end up using a ready-made array class from a library like Blitz++ discussed above.

Our class vector_operations has as data a plain one-dimensional array. We define several functions which operate on these data, from subscripting, change of the length of the array, assignment to another vector, inner product with another vector etc etc. To be more specific, we define the following usage of our class,that is the way the class is used in another part of the program:

- Create vectors of a specified length defining a vector as vector_operations v(n); Via this statement we allocate space in memory for a vector with n elements.

- Create a vector with zero length by writing the statement vector_operations v;

- Change the dimension of a vector v to a given length n by declaring v.redim(n);. Note here the way we use a function defined within a class. The function here is redim define in our class.

- Create a vector as a copy of another vector by simply writing vector_operations v(w);

- To extract the length of the vector by writing **const int** n = v. size () ;
- To find particular value of the vector **double** e = v(i);
- or assign a number to an entry via v(j) = e;
- We would also like to set two vectors equal to each other by simply writing w = v;
- or take the inner product of two vectors as **double** a = w.inner(v); or alternatively lstinlinea = inner(w,v);
- To write a vector to screen could be done by writing v. print (cout);

This list can be made longer by adding features like vector algebra, operator overloading etc.

We present now the declaration of the class, with our comments on the various declarations.

```
class vector_operations
{
private:
  double* A;                    // vector entries
  int     length;               // the length ofthe vector
  void    allocate (int n);     // allocate memory, length=n
  void    deallocate();         // free memory
public:
  vector_operations ();                  // Constructor, use as
     vector_operations v;
  vector_operations (int n);             // use as vector_operations v(n);
  vector_operations (const vector_operations& w);  //  us as
     vector_operations v(w);
 ~vector_operations ();                  // destructor to clean up dynamic
    memory

  bool redim (int n);                    // change length, us as v.redim(m)
     ;
  vector_operations& operator= (const vector_operations& w);// set two
     vectors equal v = w;
  double  operator() (int i) const;      // a = v(i);
  double& operator() (int i);            // v(i) = a;

  void print (std::ostream& o) const;    // v.print(cout);
  double inner (const vector_operations& w) const; // a = v.inner(w);
  int size () const { return length; }   // n = v.size();
};
```

The class is defined via the statement **class** vector_operations . We must first use the key word **class**, which in turn is followed by the user-defined variable name. The body of the class, data and functions, is encapsulated within the parentheses ...;.

Data and specific functions can be private, which means that they cannot be accessed from outside the class. This means also that access cannot be inherited by other functions outside the class. If we use **protected** instead of **private**, then data and functions can be inherited outside the class. The key word **public** means that data and functions can be accessed from outside the class. Here we have defined several functions which can be accessed by functions outside the class.

The first public function we encounter is a so-called constructor, which tells how we declare a variable of type vector_operations and how this variable is initialized

```
vector_operations v;    // declare a vector of length 0

// this actually means calling the function

vector_operations::vector_operations ()
{ A = NULL; length = 0; }
```

The constructor is the first function that is called when an object is instantiated. The variable A is the vector entry which defined as a private entity. Here the length is set to zero. Note also the way we define a method within the class by writing `vector_operations :: vector_operations ()`. The general form is `< return type> name of class :: name of method(<list of arguments>)`.

To give our vector v a dimensionality n we would write

```
vector_operations v(n);  // declare a vector of length n

// means calling the function

vector_operations::vector_operations (int n)
{ allocate(n); }

void vector_operations::allocate (int n)
{
  length = n;
  A = new double[n];  // create n doubles in memory
}
```

Note that we defined a Fortran-like function for allocating memory. This is one of nice features of C++ for Fortran programmers, one can always define a Fortran-like world if one wishes. Moreover,the private function allocate operates on the private variables length and A. A vector_operations object is created (dynamically) at run time, but must also be destroyed when it is no longer in use. The destructor specifies how to destroy the object via the tilde symbol shown here

```
vector_operations::~vector_operations ()
{
  deallocate();
}

// free dynamic memory:
void vector_operations::deallocate ()
{
  delete [] A;
}
```

Again we have define a deallocation statement which mimicks the Fortran way of removing an object from memory. The observant reader may also have discovered that we have sneaked in the word 'object'. What do we mean by that? A clarification is needed. We will always refer a class as user defined and declared variable which encapsulates various data (of a given type) and operations on these data. An object on the other hand is an instance of a variable of a given type. We refer to every variable we create and use as an object of a given type. The variable A above is an object of type **int**.

The function where we set two vectors to have the same length and have the same values can be written as

```
// v and w are vector_operations objects
v = w;
// means calling
vector_operations& vector_operations::operator= (const
    vector_operations& w)
// for setting v = w;
{
  redim (w.size()); // make v as long as w
  int i;
  for (i = 0; i < length; i++) { // (C++ arrays start at 0)
    A[i] = w.A[i];    // fill in teh vector w
  }
  return *this;
}
// return of *this, i.e. a vector_operations&, allows nested
    operations
u = v = u_vec = v_vec;
```

where we have used the redim function

```
v.redim(n);  // make a vector v of length n

bool vector_operations::redim (int n)
{
  if (length == n)
    return false;  // no need to allocate anything
  else {
    if (A != NULL) {
      // "this" object has already allocated memory
      deallocate();
    }
    allocate(n);
    return true;    // the length was changed
  }
}
```

and the copy action is defined as

```
vector_operations v(w);  // take a copy of w

vector_operations::vector_operations (const vector_operations& w)
{
  allocate (w.size());  // "this" object gets w's length
  *this = w;            // call operator =
}
```

Here we have defined `this` to be a pointer to the current ("this") object, in other words `*this` is the object itself.

```
void vector_operations::print (std::ostream& o) const
{
  int i;
  for (i = 1; i <= length; i++)
    o << "(" << i << ")=" << (*this)(i) << '\n';
```

```
}
```

```
double a = v.inner(w);

double vector_operations::inner (const vector_operations& w) const
{
  int i; double sum = 0;
  for (i = 0; i < length; i++)
    sum += A[i]*w.A[i];
  // alternative:
  // for (i = 1; i <= length; i++) sum += (*this)(i)*w(i);
  return sum;
}
```

```
// vector_operations v
cout << v;

ostream& operator<< (ostream& o, const vector_operations& v)
{ v.print(o); return o; }

// must return ostream& for nested output operators:
cout << "some text..." << w;

// this is realized by these calls:
operator<< (cout, "some text...");
operator<< (cout, w);
```

We can redefine the multiplication operator to mean the inner product of two vectors:

```
double a = v*w;  // example on attractive syntax

class vector_operations
{
  ...
  // compute (*this) * w
  double operator* (const vector_operations& w) const;
  ...
};

double vector_operations::operator* (const vector_operations& w) const
{
  return inner(w);
}
```

```
  // have some vector_operations u, v, w; double a;
  u = v + a*w;
  // global function operator+
  vector_operations operator+ (const vector_operations& a, const
      vector_operations& b)
  {
    vector_operations tmp(a.size());
    for (int i=1; i<=a.size(); i++)
      tmp(i) = a(i) + b(i);
```

```
    return tmp;
  }
  // global function operator*
  vector_operations operator* (const vector_operations& a, double r)
  {
    vector_operations tmp(a.size());
    for (int i=1; i<=a.size(); i++)
      tmp(i) = a(i)*r;
    return tmp;
  }
  // symmetric operator: r*a
  vector_operations operator* (double r, const vector_operations& a)
  { return operator*(a,r); }
```

Classes and templates in C++ Blitz++

We can again use templates to generalize our class to accept other types than just doubles. To achieve that we use templates, which are the native C++constructs for parameterizing parts of classes, using statements like

```
template<class T>
class vector_operations
{
  T* A;
  int length;
public:
  ...
  T& operator() (int i) { return A[i-1]; }
  ...
};
```

In a code which uses this class we could declare various vectors as Declarations in user code:

```
vector_operations<double> a(10);
vector_operations<int> i(5);
```

where the first variable is double vector with ten elements while the second is an integer vector with five elements.

Summarizing, it is easy to use the class vector_operations and we can hide in the class many details which are visible in C and Fortran 77 codes. However, as you may have noted it is not easy to write class vector_operations. One ends often up with using ready-made classes in C++ libraries such as Blitz++ unless you really need to develop your own code. Furthermore, our vector class has served mainly a pedagogical scope, since C++ has a Standard Template Library (STL) with vector types, including a vector for doing numerics that can be declared as

```
std::valarray<double> x(n);  // vector with n entries
```

However, there is no STL for a matrix type. We end therefore with recommending the use of ready-made libraries like Blitz++.

A.3 Modules in Fortran

In the previous section we discussed classes and templates in C++. Classes offer several advantages, such as

- Allows us to place classes into structures
- Pass arguments to methods
- Allocate storage for objects
- Implement associations
- Encapsulate internal details into classes
- Implement inheritance in data structures

Classes contain a new data type and the procedures that can be performed by the class. The elements (or components) of the data type are the class data members, and the procedures are the class member functions. In Fortran a class is defined as a MODULE which contains an abstract data TYPE definition. The example we elaborate on here is a Fortran class for defining operations on single-particle quantum numbers such as the total angular momentum, the orbital momentum, the energy, spin etc.

We present the MODULE single_particle_orbits here and discuss several of its feature with links to C++ programming.

```
!     Definition of single particle data

MODULE single_particle_orbits
  TYPE, PUBLIC :: single_particle_descript
     INTEGER :: total_orbits
     INTEGER, DIMENSION(:), POINTER :: nn, ll, jj, spin
     CHARACTER*10, DIMENSION(:), POINTER :: orbit_status, &
                                            model_space
     REAL(KIND=8), DIMENSION(:), POINTER :: e
  END TYPE single_particle_descript

  TYPE (single_particle_descript), PUBLIC :: all_orbit, &
       neutron_data, proton_data
  CONTAINS

! various member functions here

  SUBROUTINE allocate_sp_array(this_array,n)
  TYPE (single_particle_descript), INTENT(INOUT) :: this_array
  INTEGER , INTENT(IN) :: n
  IF (ASSOCIATED (this_array%nn) ) &
     DEALLOCATE(this_array%nn)
  ALLOCATE(this_array%nn(n))
  IF (ASSOCIATED (this_array%ll) ) &
     DEALLOCATE(this_array%ll)
  ALLOCATE(this_array%ll(n))
  IF (ASSOCIATED (this_array%jj) ) &
     DEALLOCATE(this_array%jj)
```

```
  ALLOCATE( this_array%jj(n))
  IF (ASSOCIATED (this_array%spin) ) &
     DEALLOCATE( this_array%spin)
  ALLOCATE( this_array%spin(n))
  IF (ASSOCIATED (this_array%e) ) &
      DEALLOCATE( this_array%e)
  ALLOCATE( this_array%e(n))
  IF (ASSOCIATED (this_array%orbit_status) ) &
     DEALLOCATE( this_array%orbit_status)
     ALLOCATE( this_array%orbit_status(n))
  IF (ASSOCIATED (this_array%model_space) ) &
     DEALLOCATE( this_array%model_space)
     ALLOCATE( this_array%model_space(n))
! blank all characters and zero all other values
  DO i= 1, n
     this_array%model_space(i)= ' '
     this_array%orbit_status(i)= ' '
     this_array%e(i)=0.
     this_array%nn(i)=0
     this_array%ll(i)=0
     this_array%jj(i)=0
     this_array%nshell(i)=0
     this_array%itzp(i)=0
  ENDDO

  SUBROUTINE deallocate_sp_array(this_array)

   TYPE (single_particle_descript), INTENT(INOUT) :: this_array
   DEALLOCATE( this_array%nn)
   DEALLOCATE( this_array%ll)
   DEALLOCATE( this_array%jj)
   DEALLOCATE( this_array%spin)
   DEALLOCATE( this_array%e)
   DEALLOCATE( this_array%orbit_status); &
   DEALLOCATE( this_array%model_space)

   END SUBROUTINE deallocate_sp_array
!
!     Read in all relevant single-particle data
!
  SUBROUTINE single_particle_data
    IMPLICIT NONE
    CHARACTER*100 ::  particle_species

    READ(5,*) particle_species
    WRITE(6,*) ' Particle species: '
    WRITE(6,*) particle_species
    SELECT CASE (particle_species)
       CASE ('electron')
          CALL read_electron_sp_data
       CASE ('proton&neutron')
          CALL read_nuclear_sp_data
    END SELECT
```

```
    END SUBROUTINE single_particle_data

END MODULE single_particle_orbits
```

The module ends with the END MODULE single_particle_orbits statement. We have defined a public variable TYPE, PUBLIC :: single_particle_descript which plays the same role as the **struct** type in C++. In addition we have defined several member functions which operate on various arrays and variables.

An example of a function which uses this module is given below and the module is accessed via the USE single_particle_orbits statement.

```
!
  PROGRAM main
   ....
  USE single_particle_orbits
  IMPLICIT NONE
  INTEGER :: i

  READ(5,*) all_orbit%total_orbits
  IF( all_orbit%total_orbits <= 0 ) THEN
     WRITE(6,*) 'WARNING, NO ELECTRON ORBITALS' ; STOP
  ENDIF
!     Setup all possible orbit information
!     Allocate space in heap for all single-particle data
  CALL allocate_sp_array(all_orbit,all_orbit%total_orbits)
!     Read electron single-particle data

  DO i=1, all_orbit%total_orbits
     READ(5,*) all_orbit%nn(i),all_orbit%ll , &
               all_orbit%jj(i),all_orbit%spin(i), &
               all_orbit%orbit_status(i), &
               all_orbit%model_space(i), all_orbit%e(i)
  ENDDO

! further instructions

   .......

! deallocate all arrays

  CALL deallocate_sp_array(all_orbit)


  END PROGRAM main
```

Inheritance allows one to create a hierarchy of classes in which the base class contains the common properties of the hierarchy and the derived classes can modify and specialize these properties. Specifically, a derived class contains all the class member functions of the base class and can add new ones. Further, a derived class contains all the class member functions of the base class and can modify them or add new ones. The value in using inheritance is to avoid duplicating code when creating classes which are similar to one another. Fortran does not support inheritance, but several features can be faked in Fortran! Consider the following declarations:

```
TYPE proton_sp_orbit
     TYPE (single_particle_orbits), PUBLIC :: &
           proton_particle_descript
     INTEGER, DIMENSION(:), POINTER, PUBLIC :: itzp
END TYPE proton_sp_orbit
```

To initialize the proton_sp_orbit TYPE, we could now define a new function

```
SUBROUTINE allocate_proton_array(this_array,n)

TYPE (single_particle_descript), INTENT(INOUT) :: this_array
INTEGER , INTENT(IN) :: n
IF (ASSOCIATED (this_array%itzp) ) &
   DEALLOCATE(this_array%itzp)
CALL allocate_sp_array(this_array,n)
this_array%itzp(i)=0

END SUBROUTINE allocate_proton_array
```

and

```
SUBROUTINE dellocate_proton_array(this_array)

TYPE (single_particle_descript), INTENT(INOUT) :: this_array
DEALLOCATE(this_array%itzp)
CALL deallocate_sp_array(this_array)

END SUBROUTINE deallocate_proton_array
```

and we could define a MODULE

```
  MODULE proton_class
     USE single_particle_orbits
     TYPE proton_sp_orbit
          TYPE (single_particle_orbits), PUBLIC :: &
                proton_particle_descript
          INTEGER, DIMENSION(:), POINTER, PUBLIC :: itzp
     END TYPE proton_sp_orbit
     INTERFACE allocate_proton
        MODULE PROCEDURE  allocate_proton_array, read_proton_array
     END INTERFACE
     INTERFACE deallocate_proton
        MODULE PROCEDURE  deallocate_proton_array
     END INTERFACE
     .....
     CONTAINS
     ....
!    various procedure

  END MODULE proton_class
```

```
PROGRAM with_just_protons
USE proton_class
....
```

```
TYPE (proton_sp_orbit ) :: proton_data
CALL allocate_proton(proton_data)
....
CALL deallocate_proton_array(prton_data)
```

We have a written a new class which contains the data of the base class and all the procedures of the base class have been extended to work with the new derived class. Interface statements have to be used to give the procedure uniform names.

We can now derive further classes for other particle types such as neutrons, hyperons etc etc.

A.4 Debugging of codes

Debugging is very useful in connection with segmentation fault. A text editor like Emacs has an in-built debugger. In the present section we take a C++ program and analyze the code using the LINUX debugging program **GDB**.

The steps are the following:

– Compile the program with debugging option on

c++ -g program_name.cpp -o program_name

– Start **emacs**

– Start the debugging program **GDB** in **emacs**

ESC x – gdb return – program_name

– Split the **emacs** window in two

CTRL x 2

– Read your source code into the second window. Then you are ready to start the debugging session. This means that you can execute your program in steps and have full control over execution of each individuel program statements.

– We start the program and stop at the first statement in the code.

  Put your pointer in the source code window and place it at the first executable statements. Then

CTRL x – space

  In the **GDB** window you get information about our first **breakpoint**

– Start the program in the **GDB window**

run (or r)

The program starts and stops at the first **breakpoint**. Now you have a large number of commands at your disposal. You can as an example execute statement by statement using the command

- Start the program in the **GDB window**

next (or n)

- **continue (or c)**

continue the execution from the current statement to the next **breakpoint**.

- If a statement is a call to a function you have two possibilities
- Start the program in the **GDB window**

next (or n) or step s

which takes you into the chosen subfunction and stops at the first statement. Then you can step through the subfunction using the command **next** as before till the end of the subfunction. Then you return to the previous calling function.

After having full control over the program execution we can obtain information of what is going on by using some of the following commands

- **print variable**
  print *vector@10
  display variable
  display *varable@10

- If you want to start the program again, just type **run**.
- Clean up command:
  delete (or d)
  delete display
  quit (or q)

A.5 How to make figures with Gnuplot

We end this appendix with a practical guide on making figures to be included in an eventual report file. **Gnuplot** is a simple plotting program which follows the Linux/Unix operating system. It is easy to use and allows also to generate figure files which can be included in a LaTeX document. Here we show how to make simple plots online and how to make postscript versions of the plot or even a figure file which can be included in a LaTeX document. There are other plotting programs such as **xmgrace** as well which follow Linux or Unix as operating systems. An excellent alternative which many of you are familiar with is to use Matlab to read in the data of a calculation and vizualize the results.

In order to check if gnuplot is present type

```
which gnuplot
```

If gnuplot is available, simply write

```
gnuplot
```

to start the program. You will then see the following prompt

```
gnuplot>
```

and type help for a list of various commands and help options. Suppose you wish to plot data points stored in the file **mydata.dat**. This file contains two columns of data points, where the first column refers to the argument x while the second one refers to a computed function value $f(x)$.

If we wish to plot these sets of points with gnuplot we just to need to write

```
gnuplot>plot 'mydata.dat' using 1:2 w l
```

or

```
gnuplot>plot 'mydata.dat' w l
```

since gnuplot assigns as default the first column as the x-axis. The abbreviations **w l** stand for 'with lines'. If you prefer to plot the data points only, write

```
gnuplot>plot 'mydata.dat' w p
```

For more plotting options, how to make axis labels etc, type help and choose **plot** as topic.

Gnuplot will typically display a graph on the screen. If we wish to save this graph as a postscript file, we can proceed as follows

```
gnuplot>set terminal postscript
gnuplot>set output 'mydata.ps'
gnuplot>plot 'mydata.dat' w l
```

and you will be the owner of a postscript file called **mydata.ps**, which you can display with **ghostview** through the call

```
gv mydata.ps
```

The other alternative is to generate a figure file for the document handling program **LaTeX**. The advantage here is that the text of your figure now has the same fonts as the remaining **LaTeX** document. Fig. 3.2 was generated following the steps below. You need to edit a file which ends with **.gnu**. The file used to generate Fig. 3.2 is called **derivative.gnu** and contains the following statements, which are a mix of **LaTeX** and **Gnuplot** statements. It generates a file **derivative.tex** which can be included in a **LaTeX** document. Writing the following

```
set terminal pslatex
set output "derivative.tex"
set xrange [-15:0]
set yrange [-10:8]
set xlabel "log$_{10}(h)$"
set ylabel "$\epsilon$"
plot "out.dat"  title "Relative error" w l
```

generates a LaTeX file **derivative.tex**. Alternatively, you could write the above commands in a file **derivative.gnu** and use **Gnuplot** as follows

```
gnuplot>load 'derivative.gnu'
```

You can then include this file in a LaTeX document as shown here

```
\begin{figure}
   \begin{center}
      \input{derivative}
   \end{center}
   \caption{Log-log plot of the relative error of the second
            derivative of $e^x$ as function of decreasing step
            lengths $h$. The second derivative was computed for
            $x=10$ in the program discussed above. See text for
            further details\label{fig:lossofprecision}}
\end{figure}
```

Most figures included in this text have been generated using gnuplot.

Many of the above commands can all be baked in a Python code. The following example reads a file from screen with x and y data, and plots these data and saves the result as a postscript figure.

```
#!/usr/bin/env python

import sys
from Numeric import *
import Gnuplot

g = Gnuplot.Gnuplot(persist=1)

try:
    infilename = sys.argv[1]
except:
    print "Usage of this script", sys.argv[0], "infile", sys.argv[1]; sys.
        exit(1)
# Read file with data
ifile = open(infilename, 'r')
# Fill in x and y
x = [] ;  y = []
for line in ifile:
    pair = line.split()
    x = float(pair[0]); y = float(pair[1])
ifile.close()
# convert to a form that the gnuplot interface can deal with
d = Gnuplot.Data(x, y, title='data from output file', with='lp')
g.xlabel('log10(h)')   #  make x label
g.ylabel('log10(|Exact-Computed|)/|Exact|')
g.plot(d)                          # plot the data
g.hardcopy(filename="relerror.ps",terminal="postscript", enhanced=1, color
    =1)
```

Bibliography

[1] J. Dongarra and F. Sullivan. *Computing in Science and Engineering*, 2:22, 2000.

[2] B. Cipra. *SIAM News*, 33:1, 2000.

[3] J.M. Thijssen. *Computational Physics*. Springer Verlag, 1999.

[4] S.E. Koonin and D. Meredith. *Computational Physics*. Addison Wesley, 1990.

[5] J. Gibbs. *Computational Physics*. World Scientific, 1994.

[6] B. Giordano and H. Nakanishi. *Computational Physics*. Preston, 2005.

[7] R.H. Landau and M.J. Paez. *Computational Physics*. Wiley, 1997.

[8] R. Guardiola, E. Higon, and J. Ros. *Metodes Numèrics per a la Física*. Universitat de Valencia, 1997.

[9] E.W. Schmid, G. Spitz, and W. Lösch. *Theoretische Physik mit dem Personal Computer*. Springer Verlag, 1987.

[10] H. Gould and J. Tobochnik. *An Introduction to Computer Simulation Methods: Applications to Physical Systems*. Addison-Wesley, 1996.

[11] B.T. Smith, J.C. Adams, W.S. Brainerd, and J.L. Wagener. *Fortran 95 Handbook*. MIT press, 1997.

[12] M. Metcalf and J. Reid. *The F90 Programming Language*. Oxford University Press, 1996.

[13] A.C. Marshall. *Fortran 90 Programming*. University of Liverpool, 1995.

[14] J. Reid. The new features of fortran 2003. Technical report, ISO working group on Fortran, 2007.

[15] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1999.

[16] M. Snir, S. Otto, S. Huss-Ledermann, D. Walker, and J. Dongarra. *MPI, the Complete Reference, Vols I and II*. The MIT Press, 1998.

[17] G. E. Karniadakis and R. M. Kirby II. *Parallel scientific computing in C++ and MPI*. Cambridge, 2005.

[18] *Java Numerics*. `http://math.nist.gov/javanumerics/`.

[19] B.H. Flowers. *An Introduction to Numerical Methods in C++*. Oxford University Press, 2000.

[20] J.J. Barton and L.R. Nackman. *Scientific and Engineering C++*. Addison Wesley, 1994.

[21] B. Stoustrup. *The C++ Programming Language*. Pearson, 1997.

[22] H.P. Langtangen. *Python Scripting for Computational Science*. Springer, 2006.

[23] H.P. Langtangen. *Introduction to Computer Programming; A Python-based approach for Computational Science*. Springer Verlag, 2009.

[24] D. Kincaid and W. Cheney. *Numerical Analysis*. Brooks/Gole Publishing Company, 1996.

[25] *LAPACK – Linear Algebra PACKage*. `http://www.netlib.org/lapack/`.

[26] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. *ACM Trans. Math. Soft.*, 5:308, 1979.

[27] G.H. Golub and C.F. Van Loan. *Matrix Computations*. John Hopkins University Press, 1996.

[28] B.N. Datta. *Numerical Linear Algebra and Applications*. Brooks/Cole Publishing Company, 1995.

[29] L.N. Trefethen and D. Bau III. *Numerical Linear Algebra*. SIAM Publications, 1997.

[30] R. Kress. *Numerical Analysis*. Springer, 1998.

[31] J.W. Demmel. *Numerical Linear Algebra*. SIAM Publications, 1996.

[32] T. Veldhuizen. *Blitz++ User's Guide*. `http://www.oonumerics.org/blitz/`, 2003.

[33] K.A. Reek. *Pointers on C*. Addison Wesley, 1998.

[34] J.R. Berryhill. *C++ Scientific Programming*. Wiley-Interscience, 2001.

[35] F. Franek. *Memory as a Programming Concept in C and C++*. Cambridge University Press, 2004.

[36] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C++, The art of scientific Computing*. Cambridge University Press, 1999.

[37] L. H. Thomas. The calculation of atomic fields. *Proc. Camb. Phil. Soc.*, 23:542, 1927.

[38] E. Fermi. Un metodo statistico per la determinazione di alcune proprietà dell'atomo. *Rend. Accad. Naz. Lincei*, 6:602, 1927.

[39] Elliott H. Lieb. Thomas-fermi and related theories of atoms and molecules. *Rev. Mod. Phys.*, 53(4):603–641, Oct 1981.

[40] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer Verlag, 1983.

[41] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer Verlag, 2004.

[42] P. Jackel. *Monte Carlo Methods in Finance*. John Wiley and Sons, LTD, 2002.

[43] J. Voit. *The Statistical Mechanics of Financial Markets*. Springer Verlag, 2005.

[44] J. L. McCauley. *Dynamics of Markets, Econophysics and Finance*. Cambridge University Press, 2004.

[45] D. Sornette. *Why Stock Markets Crash*. Princeton University Press, 2002.

[46] C.P. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer Verlag, 2004.

[47] J.L. Johnson. *Probability and Statistics for Computer Science*. Wiley-Interscience, 2003.

[48] G.S. Fishman. *Monte Carlo, Concepts, Algorithms and Applications*. Springer, 1996.

[49] H.A. Marzaglia and B. Zaman. *Computers in Physics*, 8:117, 1994.

[50] I. Karatsas and S.E. Shreve. *Brownian Motion and Stochastic Calculus*. Springer, 1988.

[51] N.W Ashcroft and N.D. Mermin. *Solid State Physics*. Holt-Saunders, 1976.

[52] M. Pliscke and B. Bergersen. *Equilibrium Statistical Physics*. Prentice-Hall, 1989.

[53] Lars Onsager. Crystal statistics. i. a two-dimensional model with an order-disorder transition. *Phys. Rev.*, 65(3-4):117–149, Feb 1944.

[54] H.E. Stanley. *Introduction to Phase Transitions and Critical Phenomena*. Oxford University Press, 1971.

[55] J. Cardy. *Scaling and Renormalization in Statistical Physics*. Cambridge University Press, 1996.

[56] M.E.J. Newman and G.T. Barkema. *Monte Carlo Methods in Statistical Physics*. Clarendon Press, 1999.

[57] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[58] Alan M. Ferrenberg and Robert H. Swendsen. New monte carlo technique for studying phase transitions. *Phys. Rev. Lett.*, 61:2635, 1988.

[59] Alan M. Ferrenberg and Robert H. Swendsen. Optimized monte carlo data analysis. *Phys. Rev. Lett.*, 63:1195, 1989.

[60] D.P. Landau and K. Binder. *A Guide to Monte Carlo Simulations in Statistical Physics*. Cambridge University Press, 2000.

[61] M. P. Nightingale and H. W. J. Blöte. Dynamic exponent of the two-dimensional ising model and monte carlo computation of the subdominant eigenvalue of the stochastic matrix. *Phys. Rev. Lett.*, 76(24):4548–4551, Jun 1996.

[62] Ulli Wolff. Collective monte carlo updating for spin systems. *Phys. Rev. Lett.*, 62(4):361–364, Jan 1989.

[63] Robert H. Swendsen and Jian-Sheng Wang. Nonuniversal critical dynamics in monte carlo simulations. *Phys. Rev. Lett.*, 58:86–88, 1987.

[64] B. L. Hammond, W. A. Lester, and P.J. Reynolds. *Monte Carlo Methods in ab Initio Quantum Chemistry*. World Scientific, 1994.

[65] R.L. Liboff. *Introductory Quantum Mechanics*. Addison Wesley, 2003.

[66] T. Helgaker, P. Jørgensen, and J. Olsen. *Molecular Electronic Structure Theory. Energy and Wave Functions*. Wiley, Chichester, 2000.

[67] H. R. Pagels. *Cosmic Code: Quantum Physics as the Law of Nature*. Simon and Schuster, 1982.

[68] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Cambridge University Press, 1996.

[69] J.c. Butcher. *Numerical Methods for Ordinary Differential equations*. Wiley, 2008.

[70] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I*. Springer Verlag, Berlin, 1987.

[71] S. L. Shapiro and S. A. Teukolsky. *Black holes, white dwarfs, and neutron stars: the physics of compact objects*. Wiley, 1983.

[72] H. Heiselberg and M. Hjorth-Jensen. Phases of dense matter in neutron stars. *Phys. Rep.*, 328:237, 2000.

[73] N. K. Glendenning. *Compact Stars*. Springer, 2000.

[74] H.A. Bethe and M.B. Johnson. *Nucl. Phys. A*, 230:1, 1974.

[75] H.P. Langtangen. *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*. Springer, 1999.

[76] J.M. Ortega and W.C. Poole. *An Introduction to Numerical Methods for Differential Equations*. Pitman, 1981.

[77] A. Tveito and R. Winther. *Introduction to Partial Differential Equations*. Springer, 2002.

[78] G. Arfken. *Mathematical Methods for Physicists*. Academic Press, 1985.

[79] L.C. Evans. *Partial differential equations*, volume 19 of *Graduate Studies in Mathematics*. American Mathematical Society, 2002.

[80] L. Ramdas Ram-Mohan. *Finite Element and Boundary Element Applications in Quantm Mechanics*. Oxford University Press, 2002.

[81] G. Evans, J. Blackledge, and P. Yardley. *Numerical methods for partial differential equations*. Springer, 1999.

[82] H. Kümmel, K. H. Luhrmann, and J. G. Zabolitzky. *Phys. Rep.*, 36:1, 1978, and references therein.

[83] Rodney J. Bartlett and Monika Musiał. Coupled-cluster theory in quantum chemistry. *Reviews of Modern Physics*, 79(1):291, 2007.

[84] D. J. Dean and M. Hjorth-Jensen. Coupled-cluster approach to nuclear physics. *Physical Review C*, 69:054320, 2004.

[85] K. Kowalski, D. J. Dean, M. Hjorth-Jensen, T. Papenbrock, and P. Piecuch. *Phys. Rev. Lett.*, 92:132501, 2004.

[86] B. S. Pudliner, V. R. Pandharipande, J. Carlson, Steven C. Pieper, and R. B. Wiringa. Quantum monte carlo calculations of nuclei with $a \leq 7$. *Phys. Rev. C*, 56:1720, 1997.

[87] S.E. Koonin, D.J. Dean, and K. Langanke. Shell model monte carlo methods. *Phys. Rep.*, 278:1, 1997.

[88] D. M. Ceperley. Path integrals in the theory of condensed helium. *Rev. Mod. Phys.*, 67:279, 1995.

[89] P. J. Ellis and E. Osnes. An introductory guide to effective operators in nuclei. *Rev. Mod. Phys.*, 49:777, 1977.

[90] I. Lindgren and J. Morrison. *Atomic Many-Body Theory*. Springer, Berlin, 1985.

[91] M. Hjorth-Jensen, T. T. S. Kuo, and E. Osnes. Realistic effective interactions for nuclear systems. *Phys. Rep.*, 261:125, 1995.

[92] W. H. Dickhoff and D. Van Neck. *Many-Body Theory exposed!* World Scientific, 2005.

[93] J. P. Blaizot and G. Ripka. *Quantum theory of finite systems*. MIT press, Cambridge, USA, 1986.

[94] S. R. White. Density matrix formulation for quantum renormalization groups. *Phys. Rev. Lett.*, 69:2863–2866, 1992.

[95] U. Schollwock. The density-matrix renormalization group. *Rev. Mod. Phys.*, 77:259, 2005.

[96] R. O. Jones and O. Gunnarsson. The density functional formalism, its applications and prospects. *Rev. Mod. Phys.*, 61(3):689–746, Jul 1989.

[97] R. J. Bartlett, V. F. Lotrich, and I. V. Schweigert. Ab initio density functional theory: The best of both worlds? *J. Chem. Phys.*, 123:062205, 2005.

[98] K. Peirs, D. Van Neck, and M. Waroquier. Algorithm to derive exact exchange-correlation potentials from correlated densities in atoms. *Physical Review A (Atomic, Molecular, and Optical Physics)*, 67:012505, 2003.

[99] D. Van Neck, S. Verdonck, G. Bonny, P. W. Ayers, and M. Waroquier. Quasiparticle properties in a density-functional framework. *Physical Review A (Atomic, Molecular, and Optical Physics)*, 74:042501, 2006.

[100] R. R. Whitehead, A. Watt, B. J. Cole, and I. Morrison. Computational methods for shell-model calculations. *Adv.Nucl.Phys.*, 9:123, 1977.

[101] E. Caurier, G. Martinez-Pinedo, F. Nowacki, A. Poves, and A. P. Zuker. The shell model as a unified view of nuclear structure. *Rev. Mod. Phys.*, 77:427, 2005.

[102] M. Horoi, B. A. Brown, T. Otsuka, M. Honma, and T. Mizusaki. Shell model analysis of the ^{56}ni spectrum in the full pf model space. *Phys. Rev. C*, 73:061305, 2006.

[103] D. J. Dean and M Hjorth-Jensen. *Computational Quantum Mechanics for Nuclear Physics*. Chapman & Hall/CRC, 2009.

[104] B. H. Bransden and C. J. Joachain. *Physics of Atoms and Molecules*. Longman, 1983.

[105] P. W. Atkins and R. S. Freidman. *Molecular Quantum Mechanics*. (Oxford Univerity Press, 2003).

[106] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:864, 1964.

[107] M. H. Anderson, J. R. Ensher, M. R. Matthews, C. E. Wieman, and E. A. Cornell. *Science*, 269:198, 1995.

[108] K. B. Davis, M. O. Mewes, M. R. Andrews, N. J. van Druten, D. S. Durfee, D. M. Kurn, and W. Ketterle. Bose-einstein condensation in a gas of sodium atoms. *Phys. Rev. Lett.*, 75(22):3969–3973, Nov 1995.

[109] C. C. Bradley, C. A. Sackett, J. J. Tollett, and R. G. Hulet. Evidence of bose-einstein condensation in an atomic gas with attractive interactions. *Phys. Rev. Lett.*, 75(9):1687–1690, Aug 1995.

[110] Franco Dalfovo, Stefano Giorgini, Lev P. Pitaevskii, and Sandro Stringari. Theory of bose-einstein condensation in trapped gases. *Rev. Mod. Phys.*, 71:463–512, 1999.

[111] N.N. Bogolyubov. *Dolk. Akad. Nauk SSSR*, 119:244, 1959.

[112] E.P. Gross. *Nuovo Cimento*, 20:454, 1961.

[113] L.P. Pitaevskii. *Zh.EKSP.Teor.Fiz.*, 40:646, 1961.

[114] D. A. W. Hutchinson, E. Zaremba, and A. Griffin. Finite temperature excitations of a trapped bose gas. *Phys. Rev. Lett.*, 78(10):1842–1845, Mar 1997.

[115] S. L. Cornish, N. R. Claussen, J. L. Roberts, E. A. Cornell, and C. E. Wieman. Stable ^{85}rb bose-einstein condensates with widely tunable interactions. *Phys. Rev. Lett.*, 85(9):1795–1798, Aug 2000.

[116] A. L. Fetter and J. D. Walecka. *Quantum Theory of Many-Particle Systems*. McGraw-Hill, 1971.

[117] S. Giorgini, J. Boronat, and J. Casulleras. Ground state of a homogeneous bose gas: A diffusion monte carlo calculation. *Phys. Rev. A*, 60(6):5129–5132, Dec 1999.

[118] F. Mazzanti, A. Polls, and A. Fabrocini. Energy and structure of dilute hard- and soft-sphere gases. *Phys. Rev. A*, 67(6):063615, Jun 2003.

[119] A. Fabrocini and A. Polls. Beyond the gross-pitaevskii approximation: Local density versus correlated basis approach for trapped bosons. *Phys. Rev. A*, 60(3):2319–2323, Sep 1999.

[120] J. L. DuBois and H. R. Glyde. Bose-einstein condensation in trapped bosons: A variational monte carlo analysis. *Phys. Rev. A*, 63(2):023602, Jan 2001.

[121] A. R. Sakhel, J. L. DuBois, and H. R. Glyde. Bose-einstein condensates in ^{85}rb gases at higher densities. *Phys. Rev. A*, 66(6):063610, Dec 2002.

[122] J. L. DuBois and H. R. Glyde. Natural orbitals and bose-einstein condensates in traps: A diffusion monte carlo analysis. *Phys. Rev. A*, 68(3):033602, Sep 2003.

[123] D. Blume and Chris H. Greene. Quantum corrections to the ground-state energy of a trapped bose-einstein condensate: A diffusion monte carlo calculation. *Phys. Rev. A*, 63(6):063601, May 2001.

[124] J. K. Nilsen, J. Mur-Petit, M. Guilleumas, M. Hjorth-Jensen, and A. Polls. Vortices in atomic bose-einstein condensates in the large-gas-parameter region. *Physical Review A (Atomic, Molecular, and Optical Physics)*, 71:053610, 2005.