Morten Hjorth-Jensen

Computational Physics

Lecture Notes Fall 2015

August 2015

Department of Physics, University of Oslo

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'être 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.

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 2008[1]. However, a high-level and fully object-oriented language like Python is now emerging as a good alternative although C++ and Fortran still outperform Python when it comes to computational speed. In this text we offer an approach where one can write all programs in 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 are CPU intensive. Such an approach allows you to structure the flow of data in a high-level language like Python while tasks of a mere repetitive and CPU intensive nature are left to low-level languages like C++ or Fortran. Python allows you also to smoothly interface your program with other software, such as plotting programs or operating system instructions.

[1] Throughout this text we refer to Fortran 2008 as Fortran, implying the latest standard.

A typical Python program you may end up writing contains everything from compiling and running your codes to preparing the body of a file for writing up your report.

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[2]. Moreover, the ability "to compute" forms part of the essential repertoire of research scientists. Several new fields 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.

More than a decade 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 a Bose-Einstein condensate, you need however a program which is fairly fast when computational speed matters. In this case you would most

[2] 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.

likely write a high-performance computing program using Monte Carlo methods in languages which are tailored 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 or 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++ and Fortran 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++ or Fortran being the fastest.

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[3], 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

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

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.

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 Hans Petter Langtangen, Anders Malthe-Sørenssen, Knut Mørken and Øyvind Ryan, whose input during the last fifteen years has considerably improved these lecture notes. Furthermore, the time we have spent and keep spending together on the Computing in Science Education project at the University, is just marvelous. Thanks so much. Concerning the Computing in Science Education initiative, you can read more at `http://www.mn.uio.no/english/about/collaboration/cse/`.

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, *Computational Physics*, University of Oslo (2013). The weblink to the course should also be included. Hope it is not too much to ask for. Enjoy!

Contents

Part I Introduction to programming and numerical methods

Part II Linear Algebra and Eigenvalues

Part III Differential Equations

Part IV Monte Carlo Methods

Part V Advanced topics

Part I

Introduction to programming and numerical methods

The first part of this text aims at giving an introduction to basic C++ and Fortran programming, including numerical methods for computing integrals, finding roots of functions and numerical interpolation and extrapolation. It serves also the aim of introducing the first examples on parallelization of codes for numerical integration.

Chapter 1
Introduction

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 11-14.
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 7.
4. The Householder matrix decomposition, developed by Alston Householder and discussed in chapter 7.
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 7
7. The Quicksort algorithm, developed by Anthony Hoare.
8. Fast Fourier Transform, developed by James Cooley and John Tukey.
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 (with digressions to Python as well) 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 how to program classes and the auxiliary library Blitz++, which contains several useful classes for numerical operations on vectors and matrices. This chapter contains also sections on numerical interpolation and extrapolation. Chapter 4 deals with the solution of non-linear equations and the finding of roots of polynomials. The link to Blitz++, matrices and selected algorithms for linear algebra problems are dealt with in chapter 6.

Therafter we switch to numerical integration for integrals with few dimensions, typically less than three, in chapter 5. The numerical integration chapter serves also to justify the introduction of Monte-Carlo methods discussed in chapters 11 and 12. 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 13 continues this discussion by extending to studies of phase transitions in statistical physics. Chapter 14 deals with Monte-Carlo studies of quantal systems, with an emphasis on variational Monte Carlo methods and diffusion Monte Carlo methods. In chapter 7 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 8-10 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.

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 one-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] With Fortran we will consistently mean Fortran 2008. There are no programming examples in Fortran 77 in this text.

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, see Refs. [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.
- 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.

1.1.0.1 Future plans

Since our undergraduate curriculum has changed considerably from the beginning of the fall semester of 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. Tools like Cython (or SWIG) are highly recommended, see for example the Cython link at `http://cython.org`.

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).

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 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 only a 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

[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.

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 favorite approaches to code writing.

- 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 and profiling tools is something we highly recommend during the development of a program.

Chapter 2
Introduction to C++ and Fortran

Abstract 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 tailored to these aims.

2.1 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 2.1 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. 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?

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.

[1] For more detailed texts on C++ programming in engineering and science are the books by Flowers [18] and Barton and Nackman [19]. The classic text on C++ programming is the book of Bjarne Stoustrup [20]. 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 [21, 22].

[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.

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.

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 $10e^{+308}$

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 2.1 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 chapter 3.

2.1.1 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://folk.uio.no/mhjensen/compphys/programs/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[])
{
```

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.

```
 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`. The name of the program enters `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>
#include <cstdlib>
#include <cmath>
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>
#include <cstdlib>
#include <cmath>
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 <cstdlib>
#include <cmath>
#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. Observe also that we have included the mathematics library via the `#include <cmath>` declaration.

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://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program1.f90

```
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://folk.uio.no/mhjensen/compphys/programs/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. In Python we do not need to declare variables. Mathematical functions like the $\sin$ function are imported from the *math* module. For further references to Python and its syntax, we recommend the text of Hans Petter Langtangen [22]. The corresponding codes in Python are available at the webpage of the course. All programs are listed as a directory tree beginning with programs/chapterxx. Each chapter has in turn three directories, one for C++, one for Fortran and finally one for Python codes. The Fortran codes in this chapter can be found in the directory programs/chapter02/Fortran.

2.2 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

$$417=a_n2^n+a_{n-1}2^{n-1}+a_{n-2}2^{n-2}+\cdots+a_02^0,$$

where the coefficients a_k with $k=0,\dots,n$ 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

$$a_n2^n+a_{n-1}2^{n-1}+a_{n-2}2^{n-2}+\cdots+a_02^0.$$

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://folk.uio.no/mhjensen/compphys/programs/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://folk.uio.no/mhjensen/compphys/programs/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 options which flag if an overflow or underflow is reached.

2.2.1 *Fortran codes*

The corresponding Fortran code is

http://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/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://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/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 `MOD(number,2)` in case of a division by 2. The exponentation of a number is given by for example `2**N` instead of the call to the `pow` function in C++.

2.3 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 $x = 0.006$ (in radians) our choice of precision results in

$$\sin(0.007) \approx 0.59999 \times 10^{-2},$$

and

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

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

$$f(x) = \frac{1-0.99998}{0.59999\times 10^{-2}} = \frac{0.2\times 10^{-4}}{0.59999\times 10^{-2}} = 0.33334\times 10^{-2},$$

while the second expression results in

$$f(x) = \frac{0.59999\times 10^{-2}}{1+0.99998} = \frac{0.59999\times 10^{-2}}{1.99998} = 0.30000\times 10^{-2},$$

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.

2.3.1 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 $1/10 \le r < 1$. 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 $1/2 \le q < 1$. 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

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

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 $1 \le q < 2$. 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 $-128 \le m \le 127$, 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

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

with x the exact number and the error $|\varepsilon_x| \le |\varepsilon_M|$, 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 $\varepsilon_M \sim 10^{-7}$ and for double precision (64 bits) we have $\varepsilon_M \sim 10^{-16}$, or in terms of a binary base as 2^{-23} and 2^{-52} for single and double precision, respectively.

2.3.2 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 [23] 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

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

we can write the previous inequality

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

where $|\varepsilon_x| \leq \varepsilon_M = 2^{-24}$ 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

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

or

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

meaning that

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

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 $b \approx c$)

$$\varepsilon_a \approx \frac{b}{a}(\varepsilon_b - \varepsilon_c),$$

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

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

The above subraction is thus

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

yielding

$$\frac{|fl(a)-a|}{a} = \frac{|b\varepsilon_b - c\varepsilon_c|}{a}.$$

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 [23]: *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. [23].

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 contained 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 $x \rightarrow 0$. Such problems may also need the restructuring of the algorithm.

2.4 Programming Examples on Loss of Precision and Round-off Errors

2.4.1 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
$$\exp x = \sum_{n=0}^{\infty} s_n$$
and thereafter taking the inverse
$$e^{-x} = \frac{1}{\exp x}$$

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$\{\dots\}$ statement.

http://folk.uio.no/mhjensen/compphys/programs/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}.

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

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 $\exp(-x)$.

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

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

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://folk.uio.no/mhjensen/compphys/programs/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()
```

[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 5.

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 $\exp(-x)$.

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

2.4.2 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://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program4.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

! 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://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/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
```

2.4.3 Further examples

2.4.3.1 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.

2.4.3.2 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 $\bar{x}$. 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 $\bar{x}$. 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

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

while

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

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 $\bar{x}$.

The standard algorithm computes the two contributions to σ separately, that is we sum $\sum_i x(i)^2$ and subtract thereafter $\bar{x}\sum_i x(i)$. 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 $x(i) - \bar{x}$ 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://folk.uio.no/mhjensen/compphys/programs/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://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/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
   sumsq2=sumsq2+(x(i)-xbar)**2
 ENDDO
 sigma2=SQRT(sumsq2/126.)
 WRITE(*,*) xbar, sigma1, sigma2

END PROGRAM standard_deviation
```

2.5 Additional Features of C++ and Fortran

2.5.1 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.

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 .

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.

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

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.

2.5.2 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 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://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program7.cpp

```
using namespace std;
main()
 {
   int var;
   int *pointer;

   pointer = &var;
   var = 421;
   printf("Address of the integer variable var : %p\n",&var);
   printf("Value of var : %d\n", var);
   printf("Value of the integer pointer variable: %p\n",pointer);
   printf("Value which pointer is pointing at : %d\n",*pointer);
   printf("Address of the pointer variable : %p\n",&pointer);
 }
```

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://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program8.cpp

```cpp
 1  using namespace std;
 2  #included <iostream>
 3  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
```

2.5.3 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.

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

In C++ we would replace such function definition by employing so-called `inline` functions. 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 6, 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.

2.5.4 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 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 6.

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_states;
        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. Note that we reserve the naming orbit for the generic labelling $1s$, $2s$ and $2p$ while we use the naming states when we include all possible quantum numbers. In total there are 10 possible single-particle states 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 this structure.

The following program segment illustrates how we access these elements To access these elements we could for example read from a given device the various quantum numbers:

```
   for ( int i = 0; i < electrons.total_states; 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_states
   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.

2.6 Exercises

2.1. 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.

2.2. Make a program which sums

1.

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

   and

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

   The program should read N from screen and write the final output to screen.
2. 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 $n = 1,2,\dots,10$. This means you need to compute $log_{10}(|(s_{\mathrm{up}}(N) - s_{\mathrm{down}}(N))/s_{\mathrm{down}}(N)|)$ as function of $log_{10}(N)$.

2.3. 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 $\sin x$

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

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.

2.4. Assume that you do not have access to the intrinsic function for $\exp x$. Write your own algorithm for $\exp(-x)$ 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.

2.5. 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.

2.6. 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.

2.7. 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.

2.8. 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 $b_{N+1}=b_{N+2}+..\dots=0$ 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.

2.9. 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 $x_3 = 0.236111$.

2.10. 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.

1. 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 5.
2. Make thereafter a program which calculates the real part of the spherical harmonics
3. 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 3.5.
4. 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.

2.11. Other well-known polynomials are the Laguerre and the Hermite polynomials, both being solutions to famous differential equations. The Laguerre 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 for example from the solution of the radial Schrödinger equation with a centrally symmetric potential such as the Coulomb potential. The first 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 orthogonality relation

$$\int_{-\infty}^{\infty} e^{-x} \mathscr{L}_n(x)^2 dx = 1,$$

and the recursion relation

$$(n+1)\mathscr{L}_{n+1}(x) = (2n+1-x)\mathscr{L}_n(x) - n\mathscr{L}_{n-1}(x).$$

Similalry, the Hermite polynomials are solutions of the differential equation

$$\frac{d^2H(x)}{dx^2} - 2x\frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0,$$

which arises for example by solving Schrödinger’s equation for a particle confined to move in 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 orthogonality relation

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x)^2 dx = 2^n n! \sqrt{\pi},$$

and the recursion relation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x).$$

Write a program which computes the above Laguerre and Hermite polynomials for different values of n using the pertinent recursion relations. Check your results agains some selected closed-form expressions.

Chapter 3
Numerical differentiation and interpolation

Abstract 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 the derivative 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. We discuss also selected methods for numerical interpolation. This chapter 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. We will also discuss several object-oriented features of C++, ending the chapter with an analogous discussion of Fortran features.

3.1 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 obtain an expression for the first derivative as

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

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 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 second 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

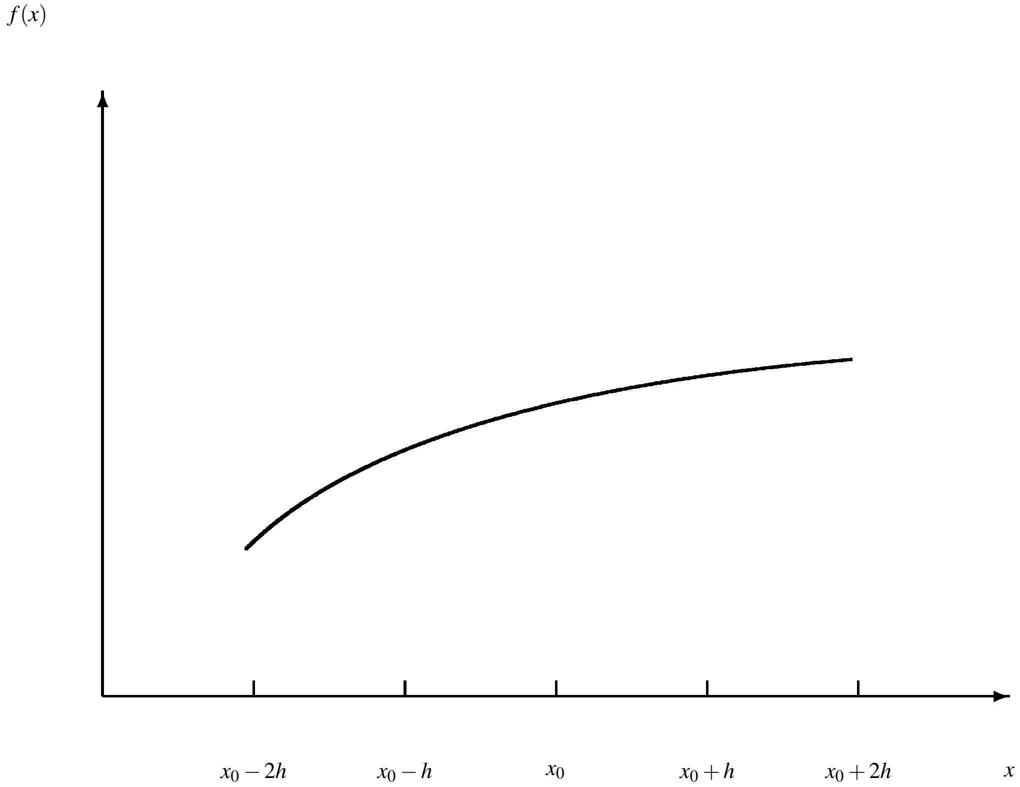


Fig. 3.1 Demonstration of the subdivision of the x-axis into small steps h. Each point corresponds to a set of values $x, f(x)$. The value of x is incremented by the step length h. If we use the points x_0 and x_0+h we can draw a straight line and use the slope at this point to determine an approximation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

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

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

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

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

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

3.1.1 The second derivative of $\exp(x)$

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

```
double speed_of_sound;
.....
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

```
using namespace std;
# include <iostream>
// begin main function
int main(int argc, char argv[])
{
  int a;
  int *b;
  a = 10;
  b = new int[10];
  for( int i = 0; i < 10; i++){
    b[i] = i;
  }
  func(a,b);
  return 0;
} // end of main function
// definition of the function func
void func(int x, int *y)
{
  x += 7;
  *y += 10;
  y[6] += 10;
21   return;
22 } // end function func
```

There are several features to be noted.

- Lines 5 and 6: 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 7: The value of a is now 10.
- Line 8: 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 10: All 10 elements of b are given values: b[0] = 0, b[1] = 1, ..., b[9] = 9;
- Line 12: 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 16: 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 18: The local variable x stored in the stack memory is changed to 17. Nothing happens with the value a in main().
- Line 19: 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 20: 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 21: 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 we 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.

3.1.1.1 Initializations 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 tasks 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://folk.uio.no/mhjensen/compphys/programs/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 initialize (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
      initialize (&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 `initialize`. 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];
 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;
 delete [] computed_derivative;
```

3.1.1.2 The function initialize

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

void initialize (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 initialize
```

This function receives the addresses of the three variables

```
void initialize (double *initial_step, double *x, int *number_of_steps)
```

and returns updated values by reading from screen.

3.1.1.3 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
     //   initialize 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`.

3.1.1.4 The output function

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

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 the 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 C++ program. This program reads from screen the names of the input and output files.

http://folk.uio.no/mhjensen/compphys/programs/chapter03/cpp/program2.cpp

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int col:
4
5 int main(int argc, char *argv[])
6 {
7   FILE *inn, *out;
8   int c;
9   if( argc < 3) {
10  printf("You have to read in :\n");
11  printf("in_file and out_file \n");
12  exit(1);
13  inn = 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( out == 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(inn);
24  fclose(out);
25  return 0;
}
```

This program has several interesting features.

Line	Program comments
5	• The function `main()` takes three arguments, given by `argc`. The variable `argv` points to the following: the name of the program, the first and second arguments, in this case the file names to be read from screen.
7	• C++ has a data type called `FILE`. The pointers `inn` 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	• This applies for the output file as well, 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://folk.uio.no/mhjensen/compphys/programs/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 initialize (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
   initialize (&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://folk.uio.no/mhjensen/compphys/programs/chapter03/Fortran/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
    !    initialize 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 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 initialize 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 chapter 6, where we deal with allocation of matrices, we will introduce the numerical libraries Armadillo and Blitz++ which allow for similar treatments of arrays in C++. By default however, these features are not included in the ANSI C++ standard.

3.1.1.5 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 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.

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.

3.1.2 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

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

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.

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

$$\varepsilon_{\text{tot}} = \varepsilon_{\text{approx}} + \varepsilon_{\text{ro}}.$$

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

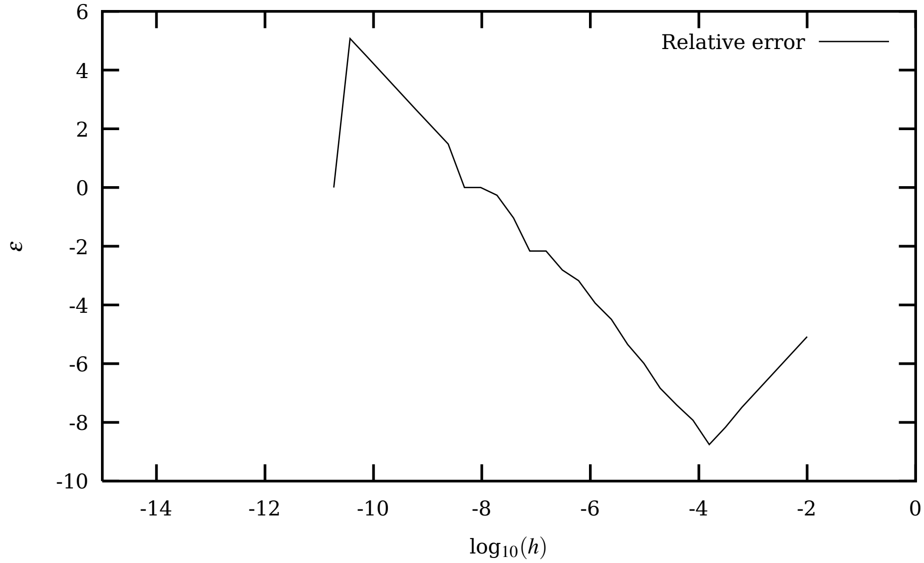


Fig. 3.2 Log-log plot of the relative error of the second derivative of $\exp(x)$ as function of decreasing step lengths h. The second derivative was computed for $x = 10$ in the program discussed above. See text for further details

$$\varepsilon_{\text{approx}} \approx \frac{f_0^{(4)}}{12} h^2.$$

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 $(f_{\pm h} - f_0) \approx \varepsilon_M$, where $|\varepsilon_M| \leq 10^{-7}$ for single and $|\varepsilon_M| \leq 10^{-15}$ for double precision, respectively.

We have then

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

Our total error becomes

$$|\varepsilon_{\text{tot}}| \leq \frac{2\varepsilon_M}{h^2} + \frac{f_0^{(4)}}{12} h^2. \tag{3.6}$$

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

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

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 $h \approx 10^{-4}$ 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

$$(f_h - f_0) + (f_{-h} - f_0) = (\exp(x+h) - \exp x) + (\exp(x-h) - \exp x),$$

as

$$(f_h - f_0) + (f_{-h} - f_0) = \exp(x)(\exp(h) + \exp(-h) - 2),$$

since it is the difference $(\exp(h) + \exp(-h) - 2)$ 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 digits.

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

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.

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 which is approximately equal to -2. This agrees again with our simple expression in Eq. (3.6).

3.2 Numerical Interpolation and Extrapolation

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. The cubic spline interpolation approach is discussed in chapter 6.

3.2.1 Interpolation

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

$$P_N(x_i) = f(x_i) = y_i, \qquad i = 0,1,\dots,N \tag{3.7}$$

for our data points. If we then write P_N on the form

$$P_N(x) = a_0 + a_1(x-x_0) + a_2(x-x_0)(x-x_1) + \dots + a_N(x-x_0)\dots(x-x_{N-1}), \tag{3.8}$$

then Eq. (3.7) results in a triangular system of equations

$$\begin{array}{lllll} a_0 & = f(x_0) & & & \\ a_0 + & a_1(x_1-x_0) & & = f(x_1) & \\ a_0 + & a_1(x_2-x_0) + & a_2(x_2-x_0)(x_2-x_1) & = f(x_2) & . \\ \dots & \dots & \dots & \dots & \end{array}$$

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

$$P_N(x) = \sum_{i=0}^{N} \prod_{k \neq i} \frac{x - x_k}{x_i - x_k} y_i. \tag{3.9}$$

If we have just two points (a straight line) we get

$$P_1(x) = \frac{x-x_0}{x_1-x_0} y_1 + \frac{x-x_1}{x_0-x_1} y_0,$$

and with three points (a parabolic approximation) we have

$$P_3(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} y_2 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} y_1 + \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} y_0$$

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

$$f(x) = P_N(x) + \frac{\omega_{N+1}(x) f^{(N+1)}(\xi)}{(N+1)!}. \tag{3.10}$$

The function $\omega_{N+1}(x)$ is given by

$$\omega_{N+1}(x) = a_N(x-x_0)\dots(x-x_N),$$

and $\xi = \xi(x)$ is a point in the smallest interval containing all interpolation points x_j and x. The program we provide below is however based on divided differences. The recipe is quite simple. If we take $x = x_0$ in Eq. (3.8), 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

$$\frac{f(x) - f(x_0)}{x - x_0} = a_1 + a_2(x-x_1) + \dots + a_N(x-x_1)(x-x_2)\dots(x-x_{N-1}),$$

where we hereafter omit the rest term

$$\frac{f^{(N+1)}(\xi)}{(N+1)!}(x-x_1)(x-x_2)\dots(x-x_N).$$

The quantity

$$f_{0x} = \frac{f(x)-f(x_0)}{x-x_0},$$

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

$$\frac{f_{0x}-f_{01}}{x-x_1} = a_2 + \dots + a_N(x-x_2)\dots(x-x_{N-1}).$$

and the quantity

$$f_{01x} = \frac{f_{0x}-f_{01}}{x-x_1},$$

is a divided difference of second order. We note that the coefficient

$$a_1 = f_{01},$$

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

$$f_{01\dots kx} = \frac{f_{01\dots(k-1)x} - f_{01\dots(k-1)k}}{x-x_k}, \tag{3.11}$$

meaning that the corresponding coefficient a_k is given by

$$a_k = f_{01\dots(k-1)k}.$$

With these definitions we see that Eq. (3.10) can be rewritten as

$$f(x) = a_0 + \sum_{k=1} N f_{01\dots k}(x-x_0)\dots(x-x_{k-1}) + \frac{\omega_{N+1}(x)f^{(N+1)}(\xi)}{(N+1)!}.$$

If we replace $x_0, x_1, \dots, x_k$ in Eq. (3.11) 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

$$f_{x_i x_{i+1}\dots x_k} = \frac{f_{x_{i+1}\dots x_k} - f_{x_i x_{i+1}\dots x_{k-1}}}{x_k - x_i}.$$

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 $N = 3$.

$$\begin{array}{llllll}
x_0 & y_0 & & & \\
 & & f_{x_0x_1} & & \\
x_1 & y_1 & & f_{x_0x_1x_2} & \\
 & & f_{x_1x_2} & & f_{x_0x_1x_2x_3} \\
x_2 & y_2 & & f_{x_1x_2x_3} & \\
 & & f_{x2x_3} & & \\
x_3 & y_3 & & &
\end{array}.$$

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_0y_0 and x_1y_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 we provide below included is based on an extension of this algorithm, knowns as Neville's algorithm. The error provided by Neville's algorithm is based on the truncation error in Eq. (3.10).

http://folk.uio.no/mhjensen/compphys/programs/chapter03/cpp/program4.cpp

```
  /*
  ** The function
  **        polint()
  ** takes as input xa[0,..,n-1] and ya[0,..,n-1] together with a given value
  ** of x and returns a value y and an error estimate dy. If P(x) is a polynomial
  ** of degree N - 1 such that P(xa_i) = ya_i, i = 0,..,n-1, then the returned
  ** value is y = P(x).
  */
void polint(double xa[], double ya[], int n, double x, double *y, double *dy)
{
 int    i, m, ns = 1;
 double den,dif,dift,ho,hp,w;
 double *c,*d;

 dif = fabs(x - xa[0]);
 c = new double [n];
 d = new double [n];
 for(i = 0; i < n; i++) {
    if((dift = fabs(x - xa[i])) < dif) {
      ns = i;
  dif = dift;
    }
    c[i] = ya[i];
    d[i] = ya[i];
 }
 *y = ya[ns--];
 for(m = 0; m < (n - 1); m++) {
   for(i = 0; i < n - m; i++) {
      ho = xa[i] - x;
      hp = xa[i + m] - x;
      w = c[i + 1] - d[i];
      if((den = ho - hp) < ZERO) {
        printf("\n\n Error in function polint(): ");
        printf("\nden = ho - hp = %4.1E -- too small\n",den);
        exit(1);
  }
      den = w/den;
      d[i] = hp * den;
      c[i] = ho * den;
    }
    *y += (*dy = (2 * ns < (n - m) ? c[ns + 1] : d[ns--]));
  }
  delete [] d;
  delete [] c;
} // End: function polint()
```

When using this function, you need obviously to declare the function itself.

3.2.2 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

$$D(h) = D(0) + a_1h^2 + a_2h^4 + a_3h^6 + \dots,$$

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

$$D(h) = D(0) + a_1h^2 + a_2h^4 + a_3h^6 + \dots,$$

and

$$D(h/2) = D(0) + \frac{a_1h^2}{4} + \frac{a_2h^4}{16} + \frac{a_3h^6}{64} + \dots,$$

and we can eliminate the term with a_1 by combining

$$D(h/2) + \frac{D(h/2) - D(h)}{3} = D(0) - \frac{a_2h^4}{4} - \frac{5a_3h^6}{16}. \tag{3.12}$$

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

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

$$\frac{f_{h/2} - f_{-h/2}}{h} = f_0' + O(h^2/4),$$

which can be combined, using Eq. (3.12) to yield

$$\frac{-f_h + 8f_{h/2} - 8f_{-h/2} + f_{-h}}{6h} = f_0' - \frac{h^4}{480}f^{(5)}.$$

In practice, what happens is that our approximations to $D(0)$ goes through a series of steps

$$\begin{array}{llll}
D_0^{(0)} & & & \\
D_0^{(1)} & D_1^{(0)} & & \\
D_0^{(2)} & D_1^{(1)} & D_2^{(0)} & \\
D_0^{(3)} & D_1^{(2)} & D_2^{(1)} & D_3^{(0)} \\
\dots & \dots & \dots & \dots
\end{array},$$

where the elements in the first column represent the given approximations

$$D_0^{(k)} = D(h/2^k).$$

This means that $D_1^{(0)}$ in the second column and row is the result of the extrapolation based on $D_0^{(0)}$ and $D_0^{(1)}$. An element $D_m^{(k)}$ in the table is then given by

$$D_m^{(k)} = D_{m-1}^{(k)} + \frac{D_{m-1}^{(k+1)} - D_{m-1}^{(k)}}{4^m - 1} \tag{3.13}$$

with $m > 0$.

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 $h = 0.000001$. 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 3.3 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. (3.13). An alternative recipe is to use our function for the polynomial extrapolation discussed in the previous subsection and calculate the derivatives for several values of h and then extrapolate to $h = 0$. We will use this method to obtain improved eigenvalues in chapter 7.

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 3.3 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.

Other methods to interpolate a function $f(x)$ such as spline methods will be discussed in chapter 6.

3.3 Classes in C++

In Fortran a vector (this applies to matrices as well) starts with 1, but it is easy to change the declaration of 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)`. Fortran allows us to write a vector addition $\mathbf{a} = \mathbf{b} + \mathbf{c}$ as `a = b + c`. This means that we have overloaded the addition operator in order to translate this operation into two loops and an addition of two vector elements $a_i = b_i + c_i$.

The way the vector 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 vectors we need to explicitely write out a loop as

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

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 vector or matrix addition in exactly the same way as we would do in Fortran. We could also change the way we declare a C++ vector (or matrix) element a_i, from $a[i]$ to say $a(i)$, 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 demonstrate most of the features of classes. We develop first a class called `Complex` which allows us to perform various operations on complex variables. We extend thereafter our discussion of classes to define a class `Vector` 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. Later, in our discussion on linear algebra, we will also present our final matrix and vector class.

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 libraries like Armadillo or Blitz++, discussed in our linear algebra chapter. These libraries hide 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 at the end of this section.

3.3.1 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 variables as `COMPLEX(DPC) :: x(6.1,8.2), y(0.5,1.3)`.

The libraries Armadillo and Blitz++ include an extension of the complex class to operations on vectors, matrices and higher-dimensional arrays. We recommend the usage of such libraries 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 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 chosen 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 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 are 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 \verb?const}. What does it mean?
This declaration means that a variable cannot be changed within a called function.
If we define a variable as
\verb?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); }
```

3.3.1.1 Overloading operators

C++ (and Fortran) allows 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);
```

The compiler translates `c = a + b;` into `c = operator+ (a, b);`. Since this implies the call to a 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 setup.

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;}
```

3.3.1.2 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 could 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 the next subsection 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++ or Armadillo.

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. From a computational point of view, a library like Armadillo `http://arma.sourceforge.net/`, which contains much of the array functionality included in Blitz++, is preferred. Armadillo is a C++ linear algebra library that aims towards a good balance between speed and ease of use. It includes optional integration possibilities with popular linear algebra packages like LAPACK and BLAS, see chapter 6 for further discussions.

3.3.2 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 libraries like Blitz++ or Armadillo discussed above. Furthermore, as was the case for the complex class, C++ contains also its own class for one-dimensional arrays, that is a vector class. At the end however, we recommend that you use libraries like Armadillo.

Our class Vector 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\ 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 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`.
- Create a vector as a copy of another vector by simply writing `Vector 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 `a = inner(w,v);`
- To write out the content of a vector could be done by via `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
{
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 ();           // Constructor, use as Vector v;
  Vector (int n);      // use as Vector v(n);
  Vector (const Vector& w); // us as Vector v(w);
 ~Vector ();           // destructor to clean up dynamic memory

  bool redim (int n);          // change length, us as v.redim(m);
  Vector& operator= (const Vector& 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& w) const; // a = v.inner(w);
  int size () const { return length; } // n = v.size();
};
```

The class is defined via the statement `class Vector`. 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` and how this variable is initialized

```
Vector v; // declare a vector of length 0

// this actually means calling the function

Vector::Vector ()
{ 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::Vector ()`. 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 v(n); // declare a vector of length n
// means calling the function
Vector::Vector (int n)
{ allocate(n); }
void Vector::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` 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::~Vector ()
{
  deallocate();
}

// free dynamic memory:
void Vector::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 to 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 objects
v = w;
// means calling
Vector& Vector::operator= (const Vector& 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&, 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::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 v(w); // take a copy of w

Vector::Vector (const Vector& 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::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::inner (const Vector& 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 v
cout << v;
```

```
ostream& operator<< (ostream& o, const Vector& 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
    {
      ...
      // compute (*this) * w
      double operator* (const Vector& w) const;
      ...
    };

    double Vector::operator* (const Vector& w) const
    {
      return inner(w);
    }
```

```
 // have some Vector u, v, w; double a;
 u = v + a*w;
 // global function operator+
 Vector operator+ (const Vector& a, const Vector& b)
 {
  Vector tmp(a.size());
  for (int i=1; i<=a.size(); i++)
    tmp(i) = a(i) + b(i);
  return tmp;
 }
 // global function operator*
 Vector operator* (const Vector& a, double r)
 {
  Vector tmp(a.size());
  for (int i=1; i<=a.size(); i++)
    tmp(i) = a(i)*r;
  return tmp;
 }
 // symmetric operator: r*a
 Vector operator* (double r, const Vector& a)
 { return operator*(a,r); }
```

3.3.2.1 Classes and templates in C++

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
```

```
{
  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<double> a(10);
Vector<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` 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`. One ends often up with using ready-made classes in C++ libraries such as Blitz++ or Armadillo 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++ or Armadillo or the matrix class discussed in the linear algebra chapter, see chapter 6.

We end this section by listing the final vector class, with both header file and the definitions of the various functions. The major part of the listing below is obvious and is not commented. The usage of the class could be as follows:

```
Vector v1;

// Redimension the vector to have length n:
int n1 = 3;
v1.redim(n1);
cout << "v1.getlength: " << v1.getLength() << endl;

// Extract the length of the vector:
const int length = v1.getLength();

// Create a vector of a specific length:
int n2 = 5;
Vector v2(n2);
cout << "v2.getlength: " << v2.getLength() << endl;

// Create a vector from an existing array:
int n3 = 3;
double* array = new double[n3];
Vector v4(n3, array);
cout << "v4.getlength: " << v4.getLength() << endl;

// Create a vector as a copy of another one:
Vector v5(v1);
cout << "v5.getlength: " << v5.getLength() << endl;

// Assign the entries in a vector:
v5(0) = 3.0; // or alternatively v5[0] = 3.0;
```

```
v5(1) = 2.5; // or alternatively v5[1] = 2.5;
v5(2) = 1.0; // or alternatively v5[2] = 1.0;

// Extract the ith component of a vector:
int i = 2;
double value = v5(1);
cout << "value: " << value << endl;

// Set a vector equal another one:
Vector v6 = v5;

cout << "try redim.v6: " << v6.redim(1) << endl;
cout << "v6.getLength: " << v6.getLength() << endl;

// Take the inner product between two vectors:
double dot = v6.inner(v5); // alternatively: double dot = inner(v6,v5);
cout << "dot(v6,v5): " << dot << endl;

// Get the euclidean norm to a vector:
double norm = v6.l2norm();
cout << "norm of v6: " << norm << endl;

// Normalize a vector:
v5.normalize();

// Dump a vector to the screen:
v5.print(std::cout << "v5: " << endl);

// Arithmetic operations with vectors using a
// syntax close to the mathematical language
Vector w = v1 + a*v2;
```

We list here the header file first.

http://folk.uio.no/mhjensen/compphys/programs/chapter03/cpp/Vector.h

```
#ifndef VECTOR_H
#define VECTOR_H

#include <cmath>
#include <iostream>

/******************************************************************************/
/*                     VECTOR CLASS                        */
/******************************************************************************/

/**
* @file Vector.h
* @class Vector
* @brief Class used for manipulating one-dimensional arrays.
*
* Contains user-defined operators to do computations with arrays in a style
* close to mathematical equations.
*
**/

class Vector{
  private:
    int length; // Number of entries.
    double *vec; // Entries.

  public:
```

```
/**
* @brief Constructor. Creates a vector initializing its elements to zero
* @param int _length. The number of entries in the array.
**/
// Default constructor
Vector();



/**
* @brief Constructor. Creates a vector initializing its elements to zero
* @param int length. The number of entries in the array.
**/
// Constructor
Vector(int _length);


/**
* Constructor. Creates a vector to hold a given array.
* @param int _length. Number of entreis in the array.
* @param const double* a. Constant pointer to a double array.
**/
// Constructor
Vector(int _length, const double *array);

/**
* Copy constructor.
*
**/
// copy constructor
Vector(const Vector&);

/**
* Destructor.
**/
// Destructor
~Vector();

/** Get the number of elements in an array.
* @return the length of the array.
**/
// Get the length of the array.
int getLength() const;

// Return pointers to the data: Useful for sending data
// to Fortran and C
const double* getPtr() const;
double* getPtr();

double inner(const Vector&) const;

//Normalize a vector, i.e., create an unit vector
// Normalize a vector
void normalize();

void print(std::ostream&) const;

/**
* Change the length of a vector
**/
```

```
bool redim(int n1);

/****************************************************/
/*   (USER-DEFINED) OVERLOADED OPERATORS */
/****************************************************/

// Member arithmetic operators (unary operators)
// Vector quantities: u, v, w. Scalar: a

// Copy-assignment (assignment by copy) operator
Vector& operator =(const Vector&); // v = w

// Add-assignment (assigment by addition) operator
Vector& operator+=(const Vector&); // v += w

// Substraction-assignment (assignment by substraction) operator
Vector& operator-=(const Vector&); // v -= w

// Multiplication-assignment (assignment by multiplication) operator
Vector& operator*=(double); // v *= a

// Division-assignment (assignment by division) operator
Vector& operator/=(double); // v /= a

const double& operator[](int i) const;
double& operator[](int i);
const double& operator()(int i) const;
double& operator()(int i);
bool indexOk(int i) const;
// Get the euclidian norm (l2norm)
double l2norm() const;
// Unary operator +
friend Vector operator+(const Vector&);   // u = + v
// Unary operator -
friend Vector operator-(const Vector&);   // u = - v
/**
* Addition of two vectors:
**/
friend Vector operator+(const Vector&, const Vector&); // u = v + w
/**
* Substraction of two vectors:
**/
friend Vector operator-(const Vector&, const Vector&); // u = v - w
/**
* Product between two vectors:
**/
friend Vector operator*(const Vector&, const Vector&); // u = v * w
 /**
* Premultiplication by a floating point number:
**/
friend Vector operator*(double, const Vector&); // u = a*v
/**
* Postmultiplication by a floating point number:
**/
friend Vector operator*(const Vector&, double); // u = v*a

/**
* Matrix-vector product:
**/
friend Vector operator*(const Matrix&, const Vector&); // u = A*v
```

```
   /**
   * Division of the entries of a vector by a scalar.
   **/
   friend Vector operator/(const Vector&, double); // u = v/a
   // dot product
   friend double inner(const Vector&, const Vector&);

   /**
   * print the entries of a vector to screen
   **/
   friend std::ostream& operator<<(std::ostream&, const Vector&); // cout << v
   // Note: This function does not need access to the data
   // member. Therefore, it could have been declared as a not friend.
};

/*******************************************************************/
/*             INLINE FUNCTIONS                    */
/*******************************************************************/

// Destructor
inline Vector::~Vector(){delete[] vec;}

// Get the number of entries in a vector
inline int Vector::getLength() const {return length;}

/**
* @return a constant pointer to the array of data.
* This function can be used to interface C++ with Fortran/C.
**/
inline const double* Vector::getPtr() const {return vec;}

/**
* @return a pointer to the array of data.
* This function can be used to interface C++ with Fortran/C.
**/
inline double* Vector::getPtr(){return vec; }

// Subscript. If v is an object of type Vector, the ith
// component of v can be accessed as v[i] closer to the
// ordinary mathematical notation instead of v.vec[i].
// The return value "const double&" is equivalent to
// "double", with the difference that the first approach
// is preferible when the returned object is big.
inline const double& Vector::operator[](int i) const{
  #ifdef CHECKBOUNDS_ON
  indexOk(i);
  #endif
  return vec[i];
} // read-only the ith component of the vector.
// const at the end of the function declaration means
// that the caller code can just read, not modify

// Subscript. (DANGEROUS)
inline double& Vector::operator[](int i){
  #ifdef CHECKBOUNDS_ON
  indexOk(i);
  #endif
  return vec[i];
} // read-write the ith coordinate
```

```
// Alternative to operator[]
inline const double& Vector::operator()(int i) const{
 #ifdef CHECKBOUNDS_ON
 indexOk(i);
 #endif
 return vec[i];
} // read-only the ith component of vec

// Subscript (DANGEROUS). If v is an object of type Vector, the ith
// component of v can be accessed as v(i) closer to the
// ordinary mathematical notation instead of v.vec(i).
inline double& Vector::operator()(int i){
 #ifdef CHECKBOUNDS_ON
 indexOk(i);
 #endif
 return vec[i];
} // read-write the ith component of vec

/******************************************************************/
/*         (Arithmetic) Unary operators          */
/******************************************************************/
// Unary operator +
inline Vector operator+(const Vector& v){ // u = + v
return v;
}

// Unary operator -
inline Vector operator-(const Vector& v){ // u = - v
return Vector(v.length) -v;
}

#endif
```

Finally, we list the source codes not included in the header file (all function which are not inlined)

http://folk.uio.no/mhjensen/compphys/programs/chapter03/cpp/Vector.cpp

```
#include "Vector.h"

/**
* @file Vector.cpp
* @class Vector
* @brief Implementation of class used for manipulating one-dimensional arrays.
**/

// default constructor
Vector::Vector(){
 length = 0;
 vec = NULL;
}

// constructor
Vector::Vector(int _length){
 length = _length;
 vec = new double[_length];
 for(int i=0; i<_length; i++)
  vec[i] = 0.0;
}

// Declare the array to be constant because it is passed
// as a pointer. Hence, it could be modified by the calling code.
```

```
Vector::Vector(int _length, // length of the array
       const double *array){ // one-dimensioal array
 length = _length;
 vec = new double[length];
 for(int i=0; i<length; i++)
  vec[i] = array[i];
}

// copy constructor
Vector::Vector(const Vector& w){
 vec = new double[length = w.length];
 for(int i=0; i<length; i++)
  vec[i] = w[i]; // This possible because we have overloaded the operator[]

 // A more straigforward way of implementing this constructor is:
 // vec = new double[length=w.length];
 // *this = w; // Here we use the assignment operator=
}

// normalize a vector
void Vector::normalize(){
 double tmp = 1.0/l2norm();
 for(int i=0;i<length; i++)
  vec[i] = vec[i]*tmp;
}

void Vector::print(std::ostream& os) const{
 int i;
 for(i=0; i<length; i++){
  os << "(" << i << ") = " << vec[i] << "\n";
 }
}

// change the length of a vector
bool Vector::redim(int _length){
 if(length == _length)
  return false;
 else{
  if(vec != NULL){
   delete[] vec;
  }
  length = _length;
  vec = new double[length];
  return true;
 }
}

bool Vector::indexOk(int i) const{
 if(i<0 || i>=length){
  std::cerr << "vector index check; index i=" << i
  << " out of bounds 0:" << length-1
  << std::endl;
  return false;
 }
 else
  return true; // valid index!
}

/**********************************************************/
/*      DEFINITION OF OPERATORS            */
/**********************************************************/
```

```
Vector& Vector::operator=(const Vector& w){ // v = w
 if(this != &w){    // beware of self-assignment v=v
  if(length != w.length)
   std::cout << "Bad vector sizes" << std::endl;
  for(int i=0; i<length; i++)
   vec[i] = w[i];  // closer to the mathematical notation than w.vec[i]
 }
 return *this;
} // assignment operator

Vector& Vector::operator+=(const Vector& w){ // v += w
 if(length != w.length) std::cout << "Bad vector sizes" << std::endl;
 for(int i=0; i<length; i++)
  vec[i] += w[i]; // This is possible because we have overloaded the operator[]
  return *this;
} // add a vector to the current one

Vector& Vector::operator-=(const Vector& w){ // v -= w
 if(length != w.length) std::cout << "Bad vector sizes" << std::endl;
 for(int i=0; i<length; i++)
  vec[i] -= w[i];// This possible because we have overloaded the operator[]
  return *this;
}

Vector& Vector::operator*=(double scalar){ // v *= a
 for(int i=0; i<length; i++)
  vec[i] *= scalar;
 return *this;
}

Vector& Vector::operator/=(double scalar){ // v /= a
 for(int i=0; i<length; i++)
  vec[i] /= scalar;
 return *this;
}

/*****************************************************************/
/*         (Arithmetic) Binary operators         */
/*****************************************************************/

// Sum of two vectors
Vector operator+(const Vector& v, const Vector& w){ // u = v + w
 // The copy constructor checks the lengths
 return Vector(v) += w;
} // vector plus vector

// Substraction of two vectors
Vector operator-(const Vector& v, const Vector& w){ // u = v - w
 // The copy constructor checks the lengths
 return Vector(v) -= w;
} // vector minus vector

// Multiplication between two vectors
Vector operator*(const Vector& v, const Vector& w){ // u = v * w
 if(v.length != w.length) std::cout << "Bad vector sizes!" << std::endl;
 int n = v.length;
 Vector tmp(n);
 for(int i=0; i<n; i++)
  tmp[i] = v[i]*w[i];
 return tmp;
} // vector times vector
```

```
// Postmultiplication operator
Vector operator*(const Vector& v, double scalar){ // u = v*a
 return Vector(v) *= scalar;
}

// Premultiplication operator.
Vector operator*(double scalar, const Vector& v){ // u = a*v
 return v*scalar; // Note the call to postmultiplication operator defined above
}

// Multiplication (product) operator: Matrix times vector
Vector operator*(const Matrix& A, const Vector& v){ // u = A*v
 int m = A.getRows();
 int n = A.getColumns();

 if(A.getColumns() != v.getLength()){
  std::cerr << "Bad sizes in: Vector operator*(const Matrix& A, const Vector& v)";
 }

 Vector u(m);
 for(int i=0; i<m; i++){
  for(int j=0; j<n; j++){
   u[i] += A[i][j]*v[j];
  }
 }
 return u;
}

// Division of the entries in a vector by a scalar
Vector operator/(const Vector& v, double scalar){
 if(!scalar) std::cout << "Division by zero!" << std::endl;
 return (1.0/scalar)*v;
}

// compute the dot product between two vectors
double inner(const Vector& u, const Vector& v){ // dot product
 if(u.length != v.length){
  std::cout << "Bad vector sizes in: double inner(const Vector& u, const Vector& v)" <<
      std::endl;
 }
 double sum = 0.0;
 for(int i=0; i<u.length; i++)
  sum += u[i]*v[i];
 return sum;
}

double Vector::inner(const Vector& v) const{ // dot product double a = u.inner(v)
 if(length != v.length)
  std::cout << "Bad vector sizes in: double Vector::inner(const Vector& v) const" <<
      std::endl;
 double sum = 0.0;
 for(int i=0; i<v.length; i++)
  sum += vec[i]*v.vec[i];
 return sum;
}

// compute the eucledian norm
double Vector::l2norm() const{
 double norm = fabs(vec[0]);
 for(int i=1; i<length; i++){
    double vi = fabs(vec[i]);
    if(norm < 100 && vi < 100){
      norm = sqrt(norm*norm + vi*vi);
    }else if(norm > vi){
      norm *= sqrt(1.0 + pow(vi/norm,2));
    }else{
      norm = vi*sqrt(1.0 + pow(norm/vi,2));
    }
  }
  return norm;
}

// dump the components of a vector to screen
std::ostream& operator<<(std::ostream& s, const Vector& v){ // output operator
  v.print(s);
  return s;
}
```

3.4 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.

3.5 How to make Figures with Gnuplot

We end this chapter 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 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)
```

3.6 Exercises

3.1. 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

$$f'_{2c}(x) = \frac{f(x+h) - f(x)}{h} + O(h), \qquad (3.14)$$

and

$$f'_{3c} = \frac{f_h - f_{-h}}{2h} + O(h^2), \qquad (3.15)$$

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

1. 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.
2. Make thereafter a program which computes the first derivative using Eqs. (3.14) and (3.15) 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. (3.14) and (3.15) 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.
3. Compute thereafter
   $$\varepsilon = log_{10}\left(\left|\frac{f'_{\text{computed}} - f'_{\text{exact}}}{f'_{\text{exact}}}\right|\right),$$
   as function of $log_{10}(h)$ for Eqs. (3.14) and (3.15) 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.

3.2. Modify your program from the previous exercise in order to include both Richardson's deferred extrapolation algorithm from Eq. (3.13) and Neville's interpolation algorithm discussed in program4.cpp in this chapter. You will need to write a program for Richardson's algorithm. Discuss and comment your results.

3.3. Use the results from your program for the calculation of derivatives to make a table of the derivatives as a function of the step length h. Write thereafter a program which reads these results and performs a numerical interpolation using Lagrange's formula from Eq. (3.9) up to a polynomial of degree five. Compare the tabulated values with those obtained using Lagrange's formula. Compare also these results with those obtained using Neville's algorithm and comment your results.

3.4. Write your own C++ class which allows for operations on complex variables, such as addition, subtraction, multiplication and division.

3.5. Write a C++ class which allows for treating one-dimensional arrays for integer, real and complex variables. Use your complex class from the previous exercise. Use this class to perform simple vector addition and vector multiplication operations.

3.6. Write a C++ class which sets up various approximations to the derivatives and repeat exercise 3.1 using this class.

3.7. Write a C++ class which sets up the position for a given particle in arbitrary dimensions. Write thereafter a program which uses this class in order to set up the electron coordinates for the ten electrons in the neutral neon atom. This is a three-dimensional system. Calculate also the distance $|\mathbf{r}_i| = \sqrt{x_i^2 + y_i^2 + z_i^2}$ (modulus of the position from the mass center, where the mass center is defined as the the atomic nucleus) of a given electron i to the atomic nucleus. Extend the class so that it can be used to calculate the modulus of the relative distance between two electrons

$$|\mathbf{r}_i - \mathbf{r}_j| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}.$$

3.8. Use the class from the previous exercise to write a program which reads in the position of all planets in the solar system, using the sun as the center of mass of the system. Let this program calculate the distance from the sun to all planets, and the relative distance between all planets.

3.9. Use and extend the vector class discussed in this chapter to compute the 1 and 2 vector norms given by

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

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

Add to the vector class the possibility to calculate an arbitrary norm p

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

where $p \geq 1$.

Write thereafter a program which checks numerically the the so-called Cauchy-Schwartz. For any $\mathbf{x}$ and $\mathbf{y}$ being real-valued or complex-valued quantities, the inner product space satisfies

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

and the equality is obeyed only if $\mathbf{x}$ and $\mathbf{y}$ are linearly dependent. Your program should be able to read from file two tabulated vectors, or, alternatively let the program set them up.

Chapter 4
Non-linear Equations

Abstract 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.

In chapter 6 we will discuss methods to find for example zeros and roots of equations. In particular, we will discuss the conjugate gradient method.

4.1 Particle in a Box Potential

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

$$-\frac{\hbar^2}{2m}\frac{d^2u}{dx^2}+V(x)u(x)=Eu(x), \tag{4.1}$$

and our potential is defined as

$$V(r)=\begin{cases}-V_0 & 0\leq x<a\\ 0 & x>a\end{cases} \tag{4.2}$$

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)

$$\frac{d^2u(x)}{dx^2}+\frac{2m}{\hbar^2}\left(V_0+E\right)u(x)=0 \qquad x<a, \tag{4.3}$$

and

$$\frac{d^2u(x)}{dx^2}+\frac{2m}{\hbar^2}Eu(x)=0 \qquad x>a. \tag{4.4}$$

If we specialize to bound states $E<0$ and implement the boundary conditions on the wave function we obtain

$$u(r)=Asin(\sqrt{2m(V_0-|E|)}r/\hbar) \qquad r<a, \tag{4.5}$$

and

$$u(r) = B\exp\left(-\sqrt{2m|E|}r/\hbar\right) \qquad r > a, \tag{4.6}$$

where A and B are constants. Using the continuity requirement on the wave function at $r = a$ one obtains the transcendental equation

$$\sqrt{2m(V_0 - |E|)}\cot\left(\sqrt{2ma^2(V_0 - |E|)}/\hbar\right) = -\sqrt{2m|E|}. \tag{4.7}$$

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 and Newton-Raphson's method.

In order to find the solution for Eq. (4.7), a simple procedure is to define a function

$$f(E) = \sqrt{2m(V_0 - |E|)}\cot\left(\sqrt{2ma^2(V_0 - |E|)}/\hbar\right) + \sqrt{2m|E|}. \tag{4.8}$$

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. 4.1 for $V_0 = 20$ MeV, $a = 2$ fm and $m = 938$ MeV. Fig. 4.1 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.

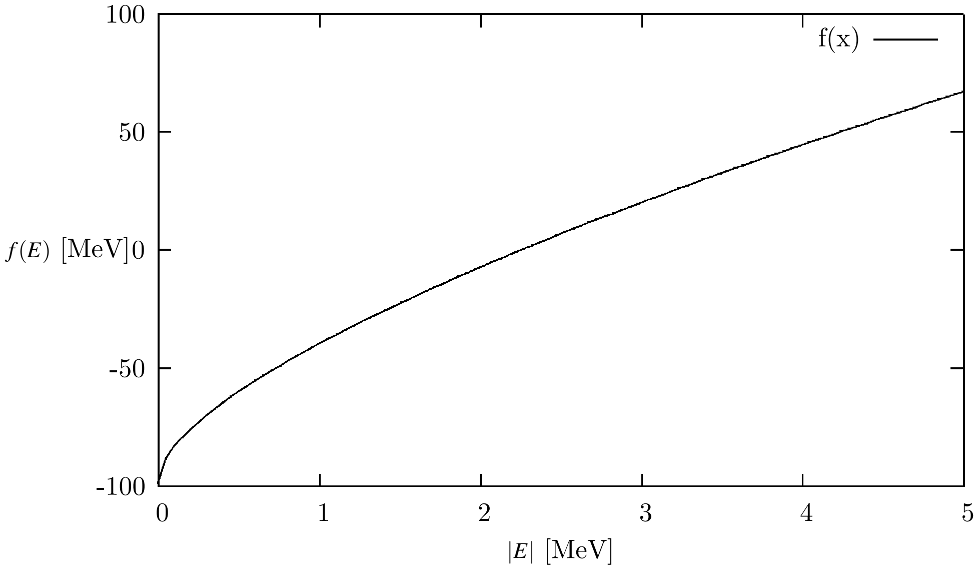


Fig. 4.1 Plot of $f(E)$ in Eq. (4.8) as function of energy |E| in MeV. Te function $f(E)$ is in units of megaelectronvolts MeV. Note well that the energy E is for bound states.

4.2 Iterative 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

[1] In the following discussion, the variable s is reserved for the value of x where we have a solution.

numerics. Through an 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

$$|x_0 - s| < \varepsilon, \tag{4.9}$$

and $f(s)=0$. We could use other criteria as well like

$$\left|\frac{x_0 - s}{s}\right| < \varepsilon, \tag{4.10}$$

and $|f(x_0)| < \varepsilon$ 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

$$|f(x_1) - f(x_2)| \le k\,|x_1 - x_2|, \tag{4.11}$$

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

$$f(x_1) - f(x_2) = f'(\xi)(x_1 - x_2), \tag{4.12}$$

with x_1,x_2 within $[a,b]$ and ξ within $[x_1,x_2]$. We have then

$$|f(x_1) - f(x_2)| \le |f'(\xi)|\,|x_1 - x_2|. \tag{4.13}$$

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 converges 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

$$|s - x_n| \le \frac{k}{1-k}\,|x_1 - x_2|. \tag{4.14}$$

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. $$|x_n - s| < \varepsilon, \tag{4.15}$$
   and
2. $$|f(s)| < \delta, \tag{4.16}$$
3. and a maximum number of iterations N_{maxiter} in actual calculations.

4.3 Bisection

This is an extremely simple method to code. The philosophy can best be explained by choosing a region in e.g., Fig. 4.1 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

$$c = \frac{a+b}{2}, \tag{4.17}$$

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 fulfills $f(c) = 0$ 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

$$\frac{1}{2^n}|b-a|, \tag{4.18}$$

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. (4.14) results in

$$|s - x_n| \leq \frac{1}{2^{n+1}}|b-a|, \tag{4.19}$$

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

$$\frac{|s-x_n|}{|s|} \le 10^{-12}. \tag{4.20}$$

It suffices in our case to study $s \ge 50$, which results in

$$\frac{|s-x_n|}{50} \le 10^{-12}, \tag{4.21}$$

and with Eq. (4.19) we obtain

$$\frac{13}{2^{n+1}50} \le 10^{-12}, \tag{4.22}$$

meaning $n \ge 37$. 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 bisection():" << 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;
}
// 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)`.

4.4 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. (4.8) 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

$$f(s) = 0 = f(x) + (s-x)f'(x) + \frac{(s-x)^2}{2}f''(x) + \dots. \tag{4.23}$$

For small enough values of the function and for well-behaved functions, the terms beyond linear are unimportant, hence we obtain

$$f(x) + (s-x)f'(x) \approx 0, \tag{4.24}$$

yielding

$$s \approx x - \frac{f(x)}{f'(x)}. \tag{4.25}$$

Having in mind an iterative procedure, it is natural to start iterating with

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{4.26}$$

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. 4.2

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

$$e_{n+1} = x_{n+1} - s = x_n - \frac{f(x_n)}{f'(x_n)} - s, \tag{4.27}$$

and using Eq. (4.23) we have

$$e_{n+1} = e_n + \frac{-e_n f'(x_n) + e_n^2/2f''(\xi)}{f'(x_n)} = \frac{e_n^2/2f''(\xi)}{f'(x_n)}. \tag{4.28}$$

This gives

$$\frac{|e_{n+1}|}{|e_n|^2} = \frac{1}{2}\frac{|f''(\xi)|}{|f'(x_n)|^2} = \frac{1}{2}\frac{|f''(s)|}{|f'(s)|^2} \tag{4.29}$$

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 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

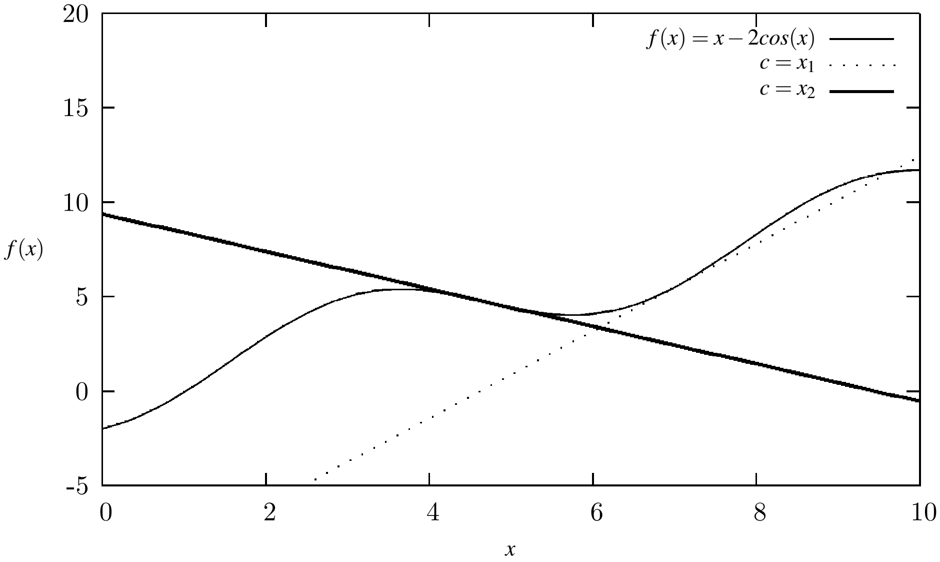


Fig. 4.2 Example of a case where Newton-Raphson's method does not converge. For the function $f(x) = x - 2cos(x)$, we see that if we start at $x = 7$, the first iteration gives us that the first point where we cross the $x-$axis is given by x_1. However, using x_1 as a starting point for the next iteration 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 $f(x) = 0$.

$$|e_{n+1}| = C|e_n|^2, \tag{4.30}$$

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

$$|s - x_{n+1}| \leq C|s - x_n|^2,$$

with $n \geq 0$. 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

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{R}{x_n}\right), \tag{4.31}$$

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

$$\begin{array}{c} f_1(x_1,x_2) = 0 \\ f_2(x_1,x_2) = 0 \end{array}, \tag{4.32}$$

which we Taylor expand to obtain

$$\begin{array}{c} 0 = f_1(x_1+h_1,x_2+h_2) = f_1(x_1,x_2)+h_1\partial f_1/\partial x_1+h_2\partial f_1/\partial x_2+\ldots \\ 0 = f_2(x_1+h_1,x_2+h_2) = f_2(x_1,x_2)+h_1\partial f_2/\partial x_1+h_2\partial f_2/\partial x_2+\ldots \end{array}. \tag{4.33}$$

Defining the Jacobian matrix $\hat{\mathbf{J}}$ we have

$$\hat{\mathbf{J}} = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{pmatrix}, \tag{4.34}$$

we can rephrase Newton's method as

$$\begin{pmatrix} x_1^{n+1} \\ x_2^{n+1} \end{pmatrix} = \begin{pmatrix} x_1^n \\ x_2^n \end{pmatrix} + \begin{pmatrix} h_1^n \\ h_2^n \end{pmatrix}, \tag{4.35}$$

where we have defined

$$\begin{pmatrix} h_1^n \\ h_2^n \end{pmatrix} = -\hat{\mathbf{J}}^{-1} \begin{pmatrix} f_1(x_1^n,x_2^n) \\ f_2(x_1^n,x_2^n) \end{pmatrix}. \tag{4.36}$$

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;
     }
     if (fabs(dx) < xacc) return rtn;
   }
  cout << "Error in function newtonraphson:" << endl;
  cout << "Too many iterations!" << endl;
}
// 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)`.

4.5 The Secant Method

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

$$x_{n+1} = x_n - f(x_n)\left(\frac{x_n - x_{n-1}}{f(x_n)-f(x_{n-1})}\right), \tag{4.37}$$

which we rewrite to

$$x_{n+1} = \frac{f(x_n)x_{n-1} - f(x_{n-1})x_n}{f(x_n)-f(x_{n-1})}. \tag{4.38}$$

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. 4.3.

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 $(a, f(a))$ to $(b, f(b))$ crosses the $x-axis$, that is

$$c = \frac{f(b)a - f(a)b}{f(b)-f(a)}. \tag{4.39}$$

We then see clearly the difference between the bisection method and the secant method. The convergence criterion for the secant method is

$$|e_{n+1}| \approx A|e_n|^{\alpha}, \tag{4.40}$$

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

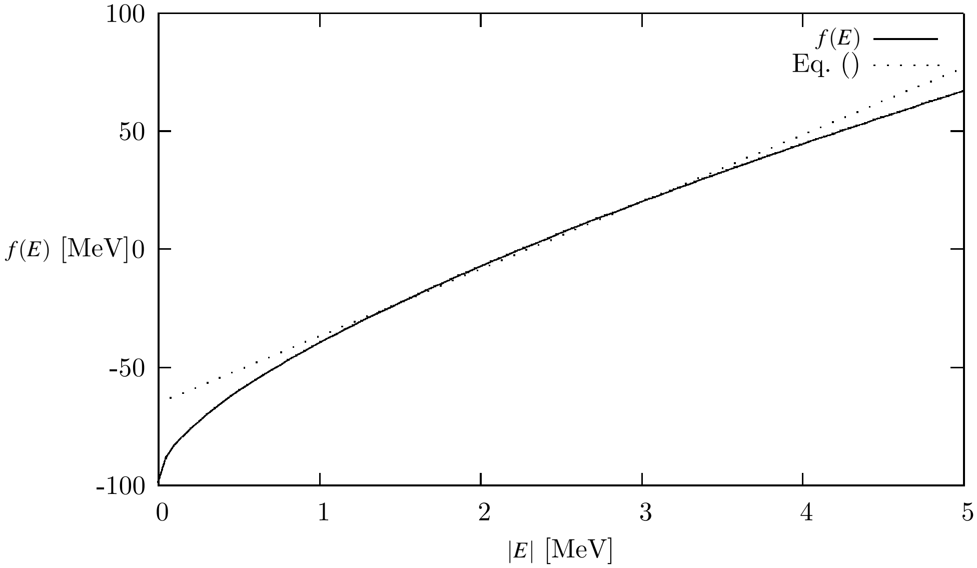


Fig. 4.3 Plot of $f(E)$ Eq. (4.8) as function of energy |E|. The point c is determined by where the straight line from $(a, f(a))$ to $(b, f(b))$ crosses the $x - axis$.

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. 4.4 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.

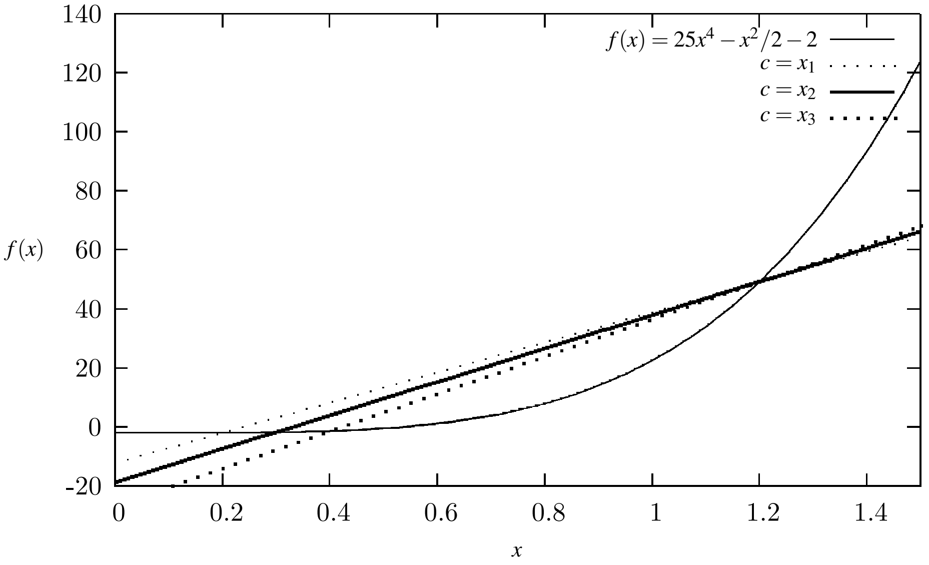


Fig. 4.4 Plot of $f(x) = 25x^4 - x^2/2 - 2$. The various straight lines correspond to the determination of the point c after each iteration. c is determined by where the straight line from $(a, f(a))$ to $(b, f(b))$ crosses the $x - axis$. Here we have chosen three values for c, x_1, x_2 and x_3 which refer to the first, second and third iterations respectively.

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 thenuse Lagrange's interpolation formula for a polynomial, see the discussion in the previous chapter.

4.5.1 Broyden's Method

Broyden's method is a quasi-Newton method for the numerical solution of nonlinear equations in k variables.

Newton's method for solving the equation $f(x) = 0$ uses the Jacobian matrix and determinant J, at every iteration. However, computing the Jacobian is a difficult and expensive operation. The idea behind Broyden's method is to compute the whole Jacobian only at the first iteration, and to do a so-called rank-one update at the other iterations.

The method is a generalization of the secant method to multiple dimensions. The secant method replaces the first derivative $f'(x_n)$ with the finite difference approximation

$$f'(x_n) \simeq \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}},$$

and proceeds using Newton's method

$$x_{n+1} = x_n - \frac{1}{f'(x_n)} f(x_n).$$

Broyden gives a generalization of this formula to a system of equations $F(x) = 0$, replacing the derivative f' with the Jacobian J. The Jacobian is determined using the secant equation (using the finite difference approximation):

$$J_n \cdot (x_n - x_{n-1}) \simeq F(x_n) - F(x_{n-1}).$$

However this equation is underdetermined in more than one dimension. Broyden suggested using the current estimate of the Jacobian J_{n-1} and improving upon it by taking the solution to the secant equation that is a minimal modification to J_{n-1} (minimal in the sense of minimizing the Frobenius norm $\|J_n - J_{n-1}\|_F$))

$$J_n = J_{n-1} + \frac{\Delta F_n - J_{n-1}\Delta x_n}{\|\Delta x_n\|^2} \Delta x_n^T,$$

and then apply Newton's method

$$x_{n+1} = x_n - J_n^{-1} F(x_n).$$

In the formula above $x_n = (x_1[n], ..., x_k[n])$ and $F_n(x) = (f_1(x_1[n], ..., x_k[n]), ..., f_k(x_1[n], ..., x_k[n]))$ are vector-columns with k elements for a system with k dimensions. We obtain then

$$\Delta x_n = \begin{bmatrix} x_1[n] - x_1[n-1] \\ ... \\ x_k[n] - x_k[n-1] \end{bmatrix} \quad \text{and} \quad \Delta F_n = \begin{bmatrix} f_1(x_1[n], ..., x_k[n]) - f_1(x_1[n-1], ..., x_k[n-1]) \\ ... \\ f_k(x_1[n], ..., x_k[n]) - f_k(x_1[n-1], ..., x_k[n-1]) \end{bmatrix}.$$

Broyden also suggested using the Sherman-Morrison formula to update directly the inverse of the Jacobian

$$J_n^{-1} = J_{n-1}^{-1} + \frac{\Delta x_n - J_{n-1}^{-1}\Delta F_n}{\Delta x_n^T J_{n-1}^{-1}\Delta F_n}(\Delta x_n^T J_{n-1}^{-1})$$

This method is commonly known as the "good Broyden's method". Many other quasi-Newton schemes have been suggested in optimization, where one seeks a maximum or minimum by finding the root of the first derivative (gradient in multi dimensions). The Jacobian of the gradient is called Hessian and is symmetric, adding further constraints to its upgrade.

4.6 Exercises

4.1. 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.

4.2. Make thereafter a class which includes the above three methods and test this class against selected problems.

4.3. 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

$$V(r) = \begin{cases} -V_0 & 0 \le r < a \\ 0 & r > a \end{cases}$$

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)

$$\frac{d^2u(r)}{dr^2} + \frac{m}{\hbar^2}(V_0 + E)u(r) = 0 \qquad r < a,$$

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

$$u(r) = B\exp{(-\beta r)} \qquad r > a,$$

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

$$kcot(ka) = -\beta. \tag{4.41}$$

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. (4.41) 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. (4.41) is satisfied, obtain a numerical value for E using the class you programmed in the previous exercise, including the 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 5
Numerical Integration

Abstract In this chapter we discuss some of the classical methods for integrating a function. The methods we discuss are the trapezoidal, rectangular and Simpson's rule for equally spaced abscissas and integration approaches 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 few-dimensional (typically up to four dimensions) integrals. In chapter 11 we show how Monte Carlo methods can be used to compute multi-dimensional integrals. We discuss also how to compute singular integrals. We end this chapter with an extensive discussion on MPI and parallel computing. The examples focus on parallelization of algorithms for computing integrals.

5.1 Newton-Cotes Quadrature

The integral

$$I = \int_a^b f(x)dx \tag{5.1}$$

has a very simple meaning. If we consider Fig. 5.1 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.

In considering equal step methods, our basic approach is that of approximating a function $f(x)$ with a polynomial of at most degree $N-1$, given N integration points. If our polynomial is of degree 1, the function will be approximated with $f(x) \approx a_0 + a_1 x$. The algorithm for these integration methods is rather simple, and the number of approximations perhaps 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.
- With a given step length we rewrite the integral as
$$\int_a^b f(x)dx = \int_a^{a+h} f(x)dx + \int_{a+h}^{a+2h} f(x)dx + \dots \int_{b-h}^{b} f(x)dx.$$

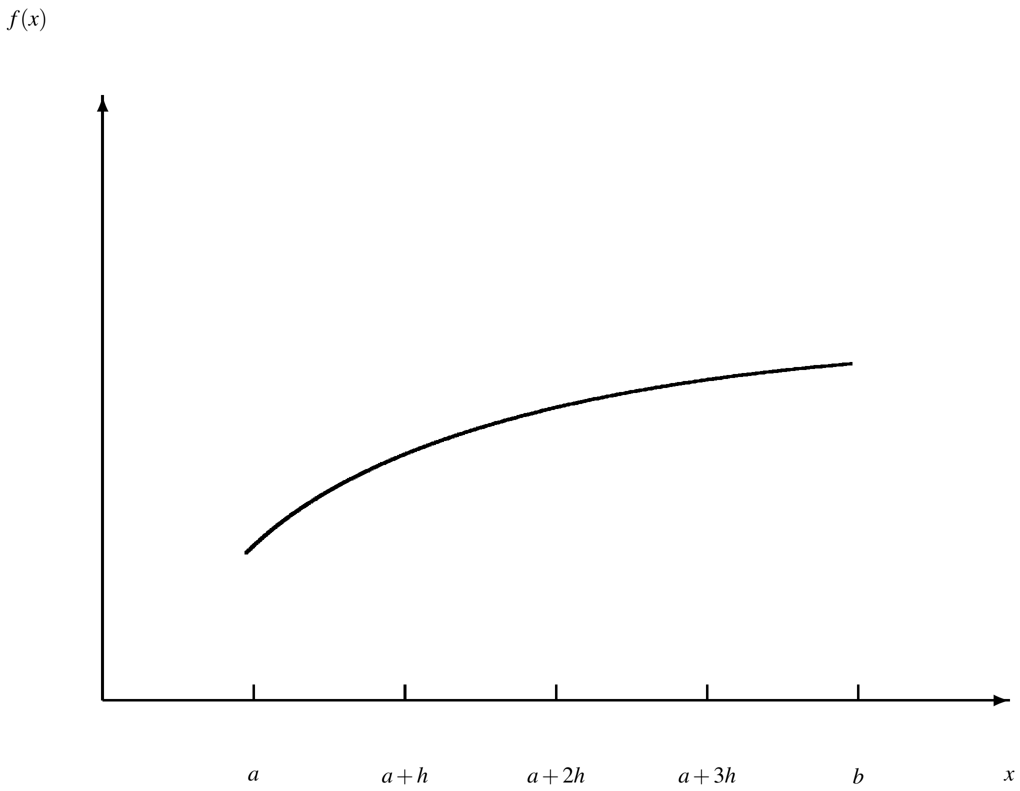


Fig. 5.1 The area enscribed by the function $f(x)$ starting from $x=a$ to $x=b$. It is subdivided in several smaller areas whose evaluation is to be approximated by the techniques discussed in the text. The areas under the curve can for example be approximated by rectangular boxes or trapezoids.

- The strategy then is to find a reliable polynomial approximation for $f(x)$ in the various intervals. Choosing a given approximation for $f(x)$, we obtain a specific approximation to the integral.
- With this approximation to $f(x)$ we perform the integration by computing the integrals over all subintervals.

Such a small measure may seemingly allow for the derivation of various integrals. To see this, we rewrite the integral as

$$\int_a^b f(x)dx = \int_a^{a+2h} f(x)dx + \int_{a+2h}^{a+4h} f(x)dx + \dots \int_{b-2h}^{b} f(x)dx.$$

One possible strategy then is to find a reliable polynomial expansion for $f(x)$ in the smaller subintervals. Consider for example evaluating

$$\int_a^{a+2h} f(x)dx,$$

which we rewrite as

$$\int_a^{a+2h} f(x)dx = \int_{x_0-h}^{x_0+h} f(x)dx. \tag{5.2}$$

We have chosen a midpoint x_0 and have defined $x_0 = a+h$. Using Lagrange's interpolation formula from Eq. (3.9), an equation we restate here,

$$P_N(x) = \sum_{i=0}^{N} \prod_{k\neq i} \frac{x-x_k}{x_i-x_k} y_i,$$

we could attempt to approximate the function $f(x)$ with a first-order polynomial in x in the two sub-intervals $x\in[x_0-h,x_0]$ and $x\in[x_0,x_0+h]$. A first order polynomial means simply that we have for say the interval $x\in[x_0,x_0+h]$

$$f(x) \approx P_1(x) = \frac{x-x_0}{(x_0+h)-x_0} f(x_0+h) + \frac{x-(x_0+h)}{x_0-(x_0+h)} f(x_0),$$

and for the interval $x\in[x_0-h,x_0]$

$$f(x) \approx P_1(x) = \frac{x-(x_0-h)}{x_0-(x_0-h)} f(x_0) + \frac{x-x_0}{(x_0-h)-x_0} f(x_0-h).$$

Having performed this subdivision and polynomial approximation, one from x_0-h to x_0 and the other from x_0 to x_0+h,

$$\int_a^{a+2h} f(x)dx = \int_{x_0-h}^{x_0} f(x)dx + \int_{x_0}^{x_0+h} f(x)dx,$$

we can easily calculate for example the second integral as

$$\int_{x_0}^{x_0+h} f(x)dx \approx \int_{x_0}^{x_0+h} \left(\frac{x-x_0}{(x_0+h)-x_0} f(x_0+h) + \frac{x-(x_0+h)}{x_0-(x_0+h)} f(x_0) \right) dx,$$

which can be simplified to

$$\int_{x_0}^{x_0+h} f(x)dx \approx \int_{x_0}^{x_0+h} \left(\frac{x-x_0}{h} f(x_0+h) - \frac{x-(x_0+h)}{h} f(x_0) \right) dx,$$

resulting in

$$\int_{x_0}^{x_0+h} f(x)dx = \frac{h}{2}\left(f(x_0+h)+f(x_0)\right) + O(h^3).$$

Here we added the error made in approximating our integral with a polynomial of degree 1. The other integral gives

$$\int_{x_0-h}^{x_0} f(x)dx = \frac{h}{2}\left(f(x_0)+f(x_0-h)\right) + O(h^3),$$

and adding up we obtain

$$\int_{x_0-h}^{x_0+h} f(x)dx = \frac{h}{2}\left(f(x_0+h)+2f(x_0)+f(x_0-h)\right) + O(h^3), \tag{5.3}$$

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. (5.1),

$$I = \int_a^b f(x)dx = h\left(f(a)/2 + f(a+h) + f(a+2h) + \cdots + f(b-h) + f_b/2\right), \tag{5.4}$$

with a global error which goes like $O(h^2)$.

Hereafter we use the shorthand notations $f_{-h} = f(x_0 - h)$, $f_0 = f(x_0)$ and $f_h = f(x_0 + h)$. 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 $\xi \in [a, b]$.

The trapezoidal rule is easy to implement 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

http://folk.uio.no/mhjensen/compphys/programs/chapter05/cpp/trapezoidal.cpp

```
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. ;
    TrapezSum=0.;
    for (j=1; j <= n-1; j++){
      x=j*step+a;
      trapez_sum+=(*func)(x);
    }
    trapez_sum=(trapez_um+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 **MyFunction** are called by reference.

The name trapezoidal rule follows from the simple fact that it has a simple geometrical interpretation, it corresponds namely to summing up a series of trapezoids, which are the approximations to the area below the curve $f(x)$.

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

$$I = \int_a^b f(x)dx \approx h\sum_{i=1}^{N} f(x_{i-1/2}), \tag{5.5}$$

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

http://folk.uio.no/mhjensen/compphys/programs/chapter05/cpp/rectangle.cpp

```
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 first-order polynomials approximations for f, we attempt at using a second-order polynomials. In this case we need three points in order to define a second-order polynomial approximation

$$f(x) \approx P_2(x) = a_0 + a_1 x + a_2 x^2.$$

Using again Lagrange's interpolation formula we have

$$P_2(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}y_2 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}y_1 + \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}y_0.$$

Inserting this formula in the integral of Eq. (5.2) we obtain

$$\int_{-h}^{+h} f(x)dx = \frac{h}{3}(f_h + 4f_0 + f_{-h}) + O(h^5),$$

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 again the *local error approximation*. Using Simpson's rule we can easily compute the integral of Eq. (5.1) to be

$$I = \int_a^b f(x)dx = \frac{h}{3}\left(f(a) + 4f(a+h) + 2f(a+2h) + \cdots + 4f(b-h) + f_b\right), \tag{5.6}$$

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

$$P_n(x_j) = y_j \qquad j = 0, \dots, n$$

In the Lagrange representation discussed in chapter 3, 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 [24] or Burlich and Stoer [25] 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. General expressions can be found in for example Refs. [24, 25].

5.2 Adaptive Integration

Before we proceed with more advanced methods like Gaussian quadrature, we mention breefly how an adaptive integration method can be implemented.

The above methods are all based on a defined step length, normally provided by the user, dividing the integration domain with a fixed number of subintervals. This is rather simple to implement may be inefficient, in particular if the integrand varies considerably in certain areas of the integration domain. In these areas the number of fixed integration points may not be adequate. In other regions, the integrand may vary slowly and fewer integration points may be needed.

In order to account for such features, it may be convenient to first study the properties of integrand, via for example a plot of the function to integrate. If this function oscillates largely in some specific domain we may then opt for adding more integration points to that particular domain. However, this procedure needs to be repeated for every new integrand and lacks obviously the advantages of a more generic code.

The algorithm we present here is based on a recursive procedure and allows us to automate an adaptive domain. The procedure is very simple to implement.

Assume that we want to compute an integral using say the trapezoidal rule. We limit ourselves to a one-dimensional integral. Our integration domain is defined by $x \in [a, b]$. The algorithm goes as follows

- We compute our first approximation by computing the integral for the full domain. We label this as $I^{(0)}$. It is obtained by calling our previously discussed function **trapezoidal_rule** as

```
I0 = trapezoidal_rule(a, b, n, function);
```

- In the next step we split the integration in two, with $c = (a+b)/2$. We compute then the two integrals $I^{(1L)}$ and $I^{(1R)}$

```
I1L = trapezoidal_rule(a, c, n, function);
```

  and

```
I1R = trapezoidal_rule(c, b, n, function);
```

  With a given defined tolerance, being a small number provided by us, we estimate the difference $|I^{(1L)} + I^{(1R)} - I^{(0)}| < \text{tolerance}$. If this test is satisfied, our first approximation is satisfactory.
- If not, we can set up a recursive procedure where the integral is split into subsequent subintervals until our tolerance is satisfied.

This recursive procedure can be easily implemented via the following function

```
//   Simple recursive function that implements the
//   adaptive integration using the trapezoidal rule
//   It is convenient to define as global variables
//   the tolerance and the number of recursive steps
const int maxrecursions = 50;
const double tolerance = 1.0E-10;
// Takes as input the integration limits, number of points, function to integrate
// and the number of steps
```

```
void adaptive_integration(double a, double b, double *Integral, int n, int steps, double
    (*func)(double))
    if ( steps > maxrecursions){
      cout << 'Too many recursive steps, the function varies too much' << endl;
      break;
    }
    double c = (a+b)*0.5;
    // the whole integral
    double I0 = trapezoidal_rule(a, b,n, func);
    // the left half
    double I1L = trapezoidal_rule(a, c,n, func);
    // the right half
    double I1R = trapezoidal_rule(c, b,n, func);
    if (fabs(I1L+I1R-I0) < tolerance ) integral = I0;
    else
    {
      adaptive_integration(a, c, integral, int n, ++steps, func)
      adaptive_integration(c, b, integral, int n, ++steps, func)
    }
}
// end function adaptive_integration
```

The variables **integral** and **steps** should be initialized to zero by the function that calls the adaptive procedure.

5.3 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,

$$I = \int_1^b x^{-2} f(x)dx,$$

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 integration 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 basic idea behind all integration methods is to approximate the integral

[1] You could e.g., impose that the integral should not change as function of increasing mesh points beyond the sixth digit.

$$I = \int_a^b f(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i),$$

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. (5.6) 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

$$I = \int_a^b f(x)dx = \int_a^b W(x)g(x)dx \approx \sum_{i=1}^{N} \omega_i g(x_i), \tag{5.7}$$

where g is smooth and W is the weight function, which is to be associated with a given orthogonal polynomial. Note that with a given weight function we end up evaluating the integrand for the function $g(x_i)$.

The weight function W is non-negative in the integration interval $x \in [a,b]$ such that for any $n \geq 0$, the integral $\int_a^b |x|^n W(x)dx$ 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

$$\int_a^b W(x)f(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i), \tag{5.8}$$

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

$$\int_a^b W(x)p(x)dx = \sum_{i=1}^{N} \omega_i p(x_i), \tag{5.9}$$

It is assumed that $W(x)$ is continuous and positive and that the integral

$$\int_a^b W(x)dx$$

[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.

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 science there are several important orthogonal polynomials which arise from the solution of differential equations. Well-known examples are the 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
$W(x) = x^{\alpha} e^{-x}$	$0 \leq x \leq \infty$	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.

5.3.1 Orthogonal polynomials, Legendre

The Legendre polynomials are the solutions of an important differential equation in Science, namely

$$C(1-x^2)P - m_l^2 P + (1-x^2)\frac{d}{dx}\left((1-x^2)\frac{dP}{dx}\right) = 0.$$

Here 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

$$L_k(x) = \frac{1}{2^k k!}\frac{d^k}{dx^k}(x^2-1)^k \qquad k = 0,1,2,\dots,$$

which, up to a factor, are the Legendre polynomials L_k. The latter fulfil the orthogonality relation

$$\int_{-1}^{1} L_i(x)L_j(x)dx = \frac{2}{2i+1}\delta_{ij}, \tag{5.10}$$

and the recursion relation

$$(j+1)L_{j+1}(x) + jL_{j-1}(x) - (2j+1)xL_j(x) = 0. \tag{5.11}$$

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 orthogonality 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 orthogonality 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

$$L_2(x) = \frac{1}{2}\left(3x^2 - 1\right). \tag{5.12}$$

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. (5.11), resulting in

$$2L_2(x) = 3xL_1(x) - L_0,$$

which leads to Eq. (5.12).

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

$$Q_{N-1}(x) = \sum_{k=0}^{N-1} \alpha_k L_k(x), \tag{5.13}$$

where α_k's are constants.

Using the orthogonality relation of Eq. (5.10) we see that

$$\int_{-1}^{1} L_N(x)Q_{N-1}(x)dx = \sum_{k=0}^{N-1} \int_{-1}^{1} L_N(x)\alpha_k L_k(x)dx = 0. \tag{5.14}$$

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. (5.11). 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 $L_{j-1}(x)$.

5.3.2 Integration 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. (5.14) 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. (5.13),

$$Q_{N-1}(x) = \sum_{i=0}^{N-1} \alpha_i L_i(x). \tag{5.15}$$

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. (5.15) as

$$Q_{N-1}(x_k) = \sum_{i=0}^{N-1} \alpha_i L_i(x_k) = \sum_{i=0}^{N-1} \alpha_i L_{ik} \qquad k = 0,1,\dots,N-1. \tag{5.16}$$

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

$$\hat{\mathbf{L}}^{-1}\hat{\mathbf{L}} = \hat{\mathbf{I}}.$$

Multiplying both sides of Eq. (5.16) with $\sum_{j=0}^{N-1} L_{ji}^{-1}$ results in

$$\sum_{i=0}^{N-1} (L^{-1})_{ki} Q_{N-1}(x_i) = \alpha_k. \tag{5.17}$$

We can derive this result in an alternative way by defining the vectors

$$\hat{\mathbf{x}}_k = \begin{pmatrix} x_0 \\ x_1 \\ . \\ . \\ x_{N-1} \end{pmatrix} \qquad \hat{\alpha} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ . \\ . \\ \alpha_{N-1} \end{pmatrix},$$

and the matrix

$$\hat{\mathbf{L}} = \begin{pmatrix} L_0(x_0) & L_1(x_0) & \dots & L_{N-1}(x_0) \\ L_0(x_1) & L_1(x_1) & \dots & L_{N-1}(x_1) \\ \dots & \dots & \dots & \dots \\ L_0(x_{N-1}) & L_1(x_{N-1}) & \dots & L_{N-1}(x_{N-1}) \end{pmatrix}.$$

We have then

$$Q_{N-1}(\hat{x}_k) = \hat{L}\hat{\alpha},$$

yielding (if $\hat{L}$ has an inverse)

$$\hat{L}^{-1} Q_{N-1}(\hat{x}_k) = \hat{\alpha},$$

which is Eq. (5.17).

Using the above results and the fact that

$$\int_{-1}^{1} P_{2N-1}(x)dx = \int_{-1}^{1} Q_{N-1}(x)dx,$$

we get

$$\int_{-1}^{1} P_{2N-1}(x)dx = \int_{-1}^{1} Q_{N-1}(x)dx = 2\alpha_0 = 2\sum_{i=0}^{N-1}(L^{-1})_{0i}P_{2N-1}(x_i).$$

If we identify the weights with $2(L^{-1})_{0i}$, where the points x_i are the zeros of L_N, 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 an orthogonal polynomial of degree N, that is L_N, while the weights are given by $2(L^{-1})_{0i}$.

5.3.3 Application to the case $N = 2$

Let us apply the above formal results to 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. (5.16)

$$Q_{N-1}(x_k) = \sum_{i=0}^{N-1} \alpha_i L_i(x_k) \qquad k = 0,1,\ldots,N-1.$$

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. (5.16) is then

$$\hat{\mathbf{L}} = \begin{pmatrix} 1 & -\frac{1}{\sqrt{3}} \\ 1 & \frac{1}{\sqrt{3}} \end{pmatrix},$$

with an inverse given by

$$\hat{\mathbf{L}}^{-1} = \frac{\sqrt{3}}{2}\begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ -1 & 1 \end{pmatrix}.$$

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?

5.3.4 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 fulfills this condition.

5.3.5 Other orthogonal polynomials

5.3.5.1 Laguerre polynomials

If we are able to rewrite our integral of Eq. (5.7) 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

$$\left(\frac{d^2}{dx^2} - \frac{d}{dx} + \frac{\lambda}{x} - \frac{l(l+1)}{x^2}\right)\mathscr{L}(x) = 0,$$

where l is an integer $l \geq 0$ and λ a constant. This equation arises for example from the solution of the radial Schrödinger equation with a centrally symmetric potential such as the Coulomb potential. The first few polynomials are

$$\mathscr{L}_0(x) = 1,$$

$$\mathscr{L}_1(x) = 1 - x,$$

$$\mathscr{L}_2(x) = 2 - 4x + x^2,$$

$$\mathscr{L}_3(x) = 6 - 18x + 9x^2 - x^3,$$

and

$$\mathscr{L}_4(x) = x^4 - 16x^3 + 72x^2 - 96x + 24.$$

They fulfil the orthogonality relation

$$\int_0^{\infty} e^{-x}\mathscr{L}_n(x)^2 dx = 1,$$

and the recursion relation

$$(n+1)\mathscr{L}_{n+1}(x) = (2n+1-x)\mathscr{L}_n(x) - n\mathscr{L}_{n-1}(x).$$

5.3.5.2 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

$$\frac{d^2H(x)}{dx^2} - 2x\frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0.$$

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 orthogonality relation

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x)^2 dx = 2^n n! \sqrt{\pi},$$

and the recursion relation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x).$$

5.3.6 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.

Table 5.1 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

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. For the corresponding Fortran program, replace program1.cpp with program1.f90. The Python program is listed as program1.py.

http://folk.uio.no/mhjensen/compphys/programs/chapter05/cpp/program1.cpp

```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
    gauss_legendre(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 5.2 we show the results for the first integral using various mesh points, while Table 5.3 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

Table 5.2 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

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.

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 5.3 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

5.4 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

$$I(x) = \mathscr{P}\int_a^b dt \frac{f(t)}{t-x} = \lim_{\varepsilon\to 0^+}\left[\int_a^{x-\varepsilon} dt\frac{f(t)}{t-x} + \int_{x+\varepsilon}^b dt\frac{f(t)}{t-x}\right],$$

and arises in applications of Cauchy's residue theorem when the pole x lies on the real axis within the interval of integration $[a,b]$. Here $\mathscr{P}$ stands for the principal value. *An important assumption is that the function $f(t)$ is continuous on the interval of integration.*

In case $f(t)$ is a closed form 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

$$\mathscr{P}\int_a^b dt\frac{f(t)}{t-x} = \int_a^{x-\Delta} dt\frac{f(t)}{t-x} + \int_{x+\Delta}^b dt\frac{f(t)}{t-x} + \mathscr{P}\int_{x-\Delta}^{x+\Delta} dt\frac{f(t)}{t-x},$$

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

$$I_\Delta(x) = \mathscr{P}\int_{-\Delta}^{+\Delta} du \frac{f(u+x)}{u}. \tag{5.18}$$

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. (5.18). Let us first introduce a new variable $s = u/\Delta$ and rewrite Eq. (5.18) as

$$I_\Delta(x) = \mathscr{P}\int_{-1}^{+1} ds \frac{f(\Delta s+x)}{s}. \tag{5.19}$$

The integration limits are now from -1 to 1, as for the Legendre polynomials. The principal value in Eq. (5.19) 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. (5.19) yields

$$I_\Delta(x) = \mathscr{P}\int_{-1}^{+1} ds \frac{f(\Delta s+x)}{s} = \int_{-1}^{+1} ds \frac{f(\Delta s+x)-f(x)}{s}, \tag{5.20}$$

and the integrand is no longer singular since we have that $\lim_{s\to 0}(f(s+x)-f(x)) = 0$ and for the particular case $s = 0$ the integrand is now finite.

Eq. (5.20) is now rewritten using the Gauss-Legendre method resulting in

$$\int_{-1}^{+1} ds \frac{f(\Delta s+x)-f(x)}{s} = \sum_{i=1}^{N} \omega_i \frac{f(\Delta s_i+x)-f(x)}{s_i}, \tag{5.21}$$

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. (5.21) will then diverge.

Let us apply this method to the integral

$$I(x) = \mathscr{P}\int_{-1}^{+1} dt \frac{e^t}{t}. \tag{5.22}$$

The integrand diverges at $x = t = 0$. We rewrite it using Eq. (5.20) as

$$\mathscr{P}\int_{-1}^{+1} dt \frac{e^t}{t} = \int_{-1}^{+1} \frac{e^t - 1}{t}, \tag{5.23}$$

since $e^x = e^0 = 1$. With Eq. (5.21) we have then

$$\int_{-1}^{+1} \frac{e^t - 1}{t} \approx \sum_{i=1}^{N} \omega_i \frac{e^{t_i} - 1}{t_i}. \tag{5.24}$$

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 ±∞ and use the fact that

$$\int_{-\infty}^{\infty} \frac{dk}{k - k_0} = \int_{-\infty}^{0} \frac{dk}{k - k_0} + \int_{0}^{\infty} \frac{dk}{k - k_0} = 0.$$

A change of variable $u = -k$ in the integral with limits from $-\infty$ to 0 gives

$$\int_{-\infty}^{\infty} \frac{dk}{k - k_0} = \int_{\infty}^{0} \frac{-du}{-u - k_0} + \int_{0}^{\infty} \frac{dk}{k - k_0} = \int_{0}^{\infty} \frac{dk}{-k - k_0} + \int_{0}^{\infty} \frac{dk}{k - k_0} = 0.$$

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

$$\mathscr{P}\int_{0}^{\infty} \frac{f(k)dk}{k^2 - k_0^2} = \int_{0}^{\infty} \frac{(f(k) - f(k_0))dk}{k^2 - k_0^2}, \tag{5.25}$$

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.

5.5 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].

5.5.1 Brief survey of supercomputing concepts and terminologies

Since many discoveries in science are nowadays obtained via large-scale simulations, there is an ever-lasting 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.

5.5.2 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 7 and of partial differential equations in chapter 10.

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 perfect parallelism, our speedup should be P, the number of processors available. We see that this is the 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. (6.5),

$$c = \sum_{j=1}^{n} x_j y_j.$$

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

5.5.3 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://folk.uio.no/mhjensen/compphys/programs/chapter05/program2.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

```
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 of your program the declaration

```
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
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

```
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)
```

[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.

while for C++ we use the function `MPI_Finalize()`.

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.

5.5.3.1 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
```

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 used 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 discuss this point in connection with our integration example below.

The above example produces 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://folk.uio.no/mhjensen/compphys/programs/chapter05/program3.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://folk.uio.no/mhjensen/compphys/programs/chapter05/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

```
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

```
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.

5.5.4 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. (5.5), 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://folk.uio.no/mhjensen/compphys/programs/chapter05/program5.cpp

```
1  //  Reactangle rule and numerical integration using MPI send and Receive
2  using namespace std;
3  #include <mpi.h>
4  #include <iostream>

5  int main (int nargs, char* args[])
6  {
7    int numprocs, my_rank, i, n = 1000;
8    double local_sum, rectangle_sum, x, h;
9    //  MPI initializations
10   MPI_Init (&nargs, &args);
11   MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
12   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
13   //  Read from screen a possible new vaue of n
14   if (my_rank == 0 && nargs > 1) {
15     n = atoi(args[1]);
16   }
17   h = 1.0/n;
18   // Broadcast n and h to all processes
19   MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
20   MPI_Bcast (&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
21   // Every process sets up its contribution to the integral
22   local_sum = 0.;
23   for (i = my_rank; i < n; i += numprocs) {
24     x = (i+0.5)*h;
25     local_sum += 4.0/(1.0+x*x);
26  }
27   local_sum *= h;
28   if (my_rank == 0) {
29     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 14 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);.
```

In case we have a floating point variable we need to declare

```
MPI_Bcast (&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

The general structure of this function is

```
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

```
MPI_Reduce( 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

```
MPI_Allreduce( 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
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://folk.uio.no/mhjensen/compphys/programs/chapter05/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.

5.6 An Integration Class

We end this chapter by presenting the usage of the integral class defined in the program library. Here we have defined two header files, the `Function.h` and the `Integral.h` files. The program below uses the classes defined in these header files to compute the integral

$$\int_0^1 \exp{(x)}\cos{(x)}.$$

```
#include <cmath>
#include <iostream>
#include "Function.h"
#include "Integral.h"

using namespace std;

class ExpCos: public Function{
 public:
  // Default constructor
  ExpCos(){}

  // Overloaded function operator().
  // Override the function operator() of the parent class.
  double operator()(double x){
     return exp(x)*cos(x);
  }
};

int main(){
 // Declare first an object of the function to be integrated
 ExpCos f;
 // Set integration bounds
 double a = 0.0;  // Lower bound
 double b = 1.0;  // Upper bound
 int npts = 100;  // Number of integration points


 // Declared (lhs) and instantiate an integral object of type Trapezoidal
 Integral *trapez = new Trapezoidal(a, b, npts, f);
 Integral *midpt = new MidPoint(a, b, npts, f);
 Integral *gl  = new Gauss_Legendre(a,b,npts, f);

 // Evaluate the integral of the function ExpCos and assign its
 // value to the variable result;
 double resultTP = trapez->evaluate();
 double resultMP = midpt->evaluate();
 double resultGL = gl->evaluate();

 // Print the result to screen
 cout << "Result with trapezoidal : " << resultTP << endl;
 cout << "Result with mid-point  : " << resultMP << endl;
 cout << "Result with Gauss-Legendre: " << resultGL << endl;
}
```

The header file Function.h is defined as

http://folk.uio.no/mhjensen/compphys/programs/chapter05/cpp/Function.h

```
/**
* @file Function.h
* Interface for mathematical functions with one or more independent variables.
* The subclasses are implemented as functors, i.e., objects behaving as functions.
* They overload the function operator().
*
* Example Usage:
// 1. Declare a functor, i.e., an object which
// overloads the function operator().
class Squared: public Function{
  public:
    // Overload function operator()
    double operator()(double x=0.0){
      return x*x;
    }
};

int main(){
  // Instance an object Functor
  Squared f;

  // Use the instance of the object as a normal function
  cout << f(3.0) << endl;
}
@endcode
*
**/
```

```
#ifndef FUNCTION_H
#define FUNCTION_H

#include "Array.h"

class Function{
  public:

  //! Destructor
  virtual ~Function(){}; // Not needed here.

    /**
    * @brief Overload the function operator().
    *
    * Used for evaluating functions with one independent variable.
    *
    **/
    virtual double operator()(double x){}

    /**
    * @brief Overload the function operator().
    *
    * Used for evaluating functions with more than one independent variable.
    **/
    virtual double operator()(const Array<double>& x){}
};
#endif
```

The header file Integral.h contains, with an example on how to use it, the following statements

http://folk.uio.no/mhjensen/compphys/programs/chapter05/cpp/Integral.h

```

#ifndef INTEGRAL_H
#define INTEGRAL_H

#include "Array.h"
#include "Function.h"
#include <cmath>

class Integral{
  protected:  // Access in the subclasses.
    double a; // Lower limit of integration.
    double b; // Upper limit of integration.
    int npts; // Number of integration points.
    Function &f; // Function to be integrated.

  public:

    /**
    * @brief Constructor.
    *
    * @param lower_. Lower limit of integration.
    * @param upper_. Upper limit of integration.
    * @param npts_. Number of points of integration.
    * @param f_. Reference to a functor representing the function to be integrated.
    **/
    Integral(double lower_, double upper_, int npts_, Function &f_);

    //! Destructor
    virtual ~Integral(){}
```

```
   /**
   * @brief Evaluate the integral.
   * @return The value of the integral in double precision.
   **/
   virtual double evaluate()=0;


   // virtual forloop

}; // End class Integral

class Trapezoidal: public Integral{
 private:
   double h;  // Step size.

 public:
   /**
   * @brief Constructor.
   *
   * @param lower_. Lower limit of integration.
   * @param upper_. Upper limit of integration.
   * @param npts_. Number of points of integration.
   * @param f_. Reference to a functor representing the function to be integrated.
   **/
   Trapezoidal(double lower_, double upper_, int npts_, Function &f_);

   //! Destructor
   ~Trapezoidal(){}

   /**
   * Evaluate the integral of a function f using the trapezoidal rule.
   * @return The value of the integral in double precision.
   **/
   double evaluate();
}; // End class Trapezoidal

class MidPoint: public Integral{
 private:
   double h;   // Step size.

 public:
   /**
   * @brief Constructor.
   *
   * @param lower_. Lower limit of integration.
   * @param upper_. Upper limit of integration.
   * @param npts_. Number of points of integration.
   * @param f_. Reference to a functor representing the function to be integrated.
   **/
   MidPoint(double lower_, double upper_, int npts_, Function &f_);

   //! Destructor
   ~MidPoint(){}

   /**
   * Evaluate the integral of a function f using the midpoint approximation.
   *
   * @return The value of the integral in double precision.
   **/
   double evaluate();
```

```
};

class Gauss_Legendre: public Integral{
 private:
   static const double ZERO = 1.0E-10;
   static const double PI = 3.14159265359;
   double h;

 public:
   /**
   * @brief Constructor.
   *
   * @param lower_. Lower limit of integration.
   * @param upper_. Upper limit of integration.
   * @param npts_. Number of points of integration.
   * @param f_. Reference to a functor representing the function to be integrated.
   **/
   Gauss_Legendre(double lower_, double upper_, int npts_, Function &f_);

   //! Destructor
   ~Gauss_Legendre(){}

   /**
   * Evaluate the integral of a function f using the Gauss-Legendre approximation.
   *
   * @return The value of the integral in double precision.
   **/
   double evaluate();
};
#endif
```

5.7 Exercises

5.1. Use Lagrange's interpolation formula for a second-order polynomial

$$P_2(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}y_2 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}y_1 + \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}y_0,$$

and insert this formula in the integral

$$\int_{-h}^{+h} f(x)dx \approx \int_{-h}^{+h} P_2(x)dx,$$

and derive Simpson's rule. You need to define properly the values x_0, x_1 and x_2 and link them with the integration limits x_0-h and x_0+h. Simpson's formula reads

$$\int_{-h}^{+h} f(x)dx = \frac{h}{3}\left(f_h + 4f_0 + f_{-h}\right) + O(h^5).$$

Write thereafter a class which implements both the Trapezoidal rule and Simpson's rule. You can for example follow the example given in the last section of this chapter. You can look up the header file for this class at `http://folk.uio.no/mhjensen/compphys/programs/chapter05/cpp/Integral.h`.

5.2. Write a program which then uses the above class containing the Trapezoidal rule and Simpson's rule to implement the adaptive algorithm discussed in section 5.2. Compute the integrals

$$I = \int_0^1 \frac{4}{1+x^2} = \pi,$$

and

$$I = \int_0^\infty x\exp(-x)\sin x = \frac{1}{2}.$$

Discuss strategies for choosing the integration limits using these methods

5.3. Add now to your integration class the possibility for extrapolating $h \to 0$ using Richardson's deferred extrapolation technique, see Eq. (3.13) and exercise 3.2 in chapter 3.

5.4. Write a class which includes your own functions for Gaussian quadrature using Legendre, Hermite and Laguerre polynomials. You can write your own functions for these methods or use those included with the programs of this book. For the latter see for example the programs in the directory programs/chapter05. The functions are called gausslegendre.cpp, gausshermite.cpp and gausslaguerre.cpp.

Use the Legendre and Laguerre polynomials to evaluate again

$$I = \int_0^\infty x\exp(-x)\sin x = \frac{1}{2}.$$

5.5. The task here is to integrate a six-dimensional integral which is used to determine the ground state correlation energy between two electrons in a helium atom. The integral appears in many quantum mechanical applications. However, if you are not too familiar with quantum mechanics, you can simply look at the mathematical details. We will employ both Gauss-Legendre and Gauss-Laguerre quadrature. Furthermore, you will need to parallelize your code. You can use your class from the previous problem.

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 so-called $1s$ wave functions as

$$\Psi(\mathbf{r}_1,\mathbf{r}_2) = e^{-\alpha(r_1+r_2)}.$$

Note that it is not possible to find a closed-form 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 which repel each other via the classical Coulomb interaction, namely

$$\langle\frac{1}{|\mathbf{r}_1-\mathbf{r}_2|}\rangle = \int_{-\infty}^{\infty} d\mathbf{r}_1 d\mathbf{r}_2 e^{-2\alpha(r_1+r_2)}\frac{1}{|\mathbf{r}_1-\mathbf{r}_2|}.$$

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.

This integral can be solved in closed form and the answer is $5\pi^2/16^2$. Can you derive this value?

1. 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 third leading digit? Hint: the single-particle wave function $e^{-\alpha r_i}$ is more or less zero at $r_i \approx ?$ (find the appropriate limit). You can therefore replace the integration limits $-\infty$ and ∞ with $-?$ and $?$, respectively. You need to check that this approximation is satisfactory, that is, make a plot of the function and check if the abovementioned limits are appropriate. You need also to account for the potential problems which may arise when $|\mathbf{r}_1 - \mathbf{r}_2| = 0$.
2. The Legendre polynomials are defined for $x \in [-1, 1]$. The previous exercise gave a very unsatisfactory ad hoc procedure. We wish to improve our results. It can therefore be useful to change to another coordinate frame and employ the Laguerre polynomials. The Laguerre polynomials are defined for $x \in [0, \infty)$ and if we change to spherical coordinates

   $$d\mathbf{r}_1 d\mathbf{r}_2 = r_1^2 dr_1 r_2^2 dr_2 dcos(\theta_1) dcos(\theta_2) d\phi_1 d\phi_2,$$

   with

   $$\frac{1}{r_{12}} = \frac{1}{\sqrt{r_1^2 + r_2^2 - 2r_1 r_2 cos(\beta)}}$$

   and

   $$cos(\beta) = cos(\theta_1)cos(\theta_2) + sin(\theta_1)sin(\theta_2)cos(\phi_1 - \phi_2))$$

   we can rewrite the above integral with different integration limits. Find these limits and replace the Gauss-Legendre approach in a) with Laguerre polynomials. Do your results improve? Compare with the results from a).
3. Make a detailed analysis of the time used by both methods and compare your results. Parallelize your codes and check that you have an optimal speed up.

Part II
Linear Algebra and Eigenvalues

This part of the text aims at giving an overview over several methods to solve linear algebra and eigenvalue problems. These methods span from standard Gaussian elimination to iterative eigenvalue solvers for large eigenvalue problems. Furthermore, several technicalities which pertain to handling arrays, their memory allocation and deallocation, classes for handling arrays and links to professional software packages such as LAPACK and BLAS will also be discussed. Parallelization of both eigenvalue solvers and linear algebra problems are also discussed.

Chapter 6
Linear Algebra

Abstract This chapter introduces several matrix related topics, from the solution of linear equations, computing determinants, conjugate-gradient methods, spline interpolation to efficient handling of matrices

6.1 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 also be discussed.

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.

The algorithms we describe and their original source codes are taken from the widely used software package LAPACK [26], 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 [27], 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 [28]. Good recent introductory texts are Kincaid and Cheney [23] and Datta [29]. For more advanced ones see Trefethen and Bau III [30], Kress [24] and Demmel [31]. Ref. [28] contains an extensive list of textbooks on eigenvalue problems and linear algebra. LAPACK [26] 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`. A library we highly recommend is Armadillo, see `http://arma.sourceforge.org`. Armadillo is an open-source C++ linear algebra library aiming towards a good balance between speed and ease of use. Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions.

Various matrix and vector operations are provided through optional integration with BLAS and LAPACK.

6.2 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 $\mathbf{I}$

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \qquad \mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{6.1}$$

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 $\mathbf{A}$ 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 $\mathbf{x}$ 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 6.2 lists some essential features of various types of matrices one may encounter. Some of the matrices we will encounter are listed here

Table 6.1 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	$\sum_k a_{ik}a_{jk}^* = \sum_k a_{ki}^*a_{kj} = \delta_{ij}$

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

$$\mathbf{A}=\mathbf{B}\pm\mathbf{C}\Longrightarrow a_{ij}=b_{ij}\pm c_{ij}, \tag{6.2}$$

scalar-matrix multiplication

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

vector-matrix multiplication

$$\mathbf{y} = \mathbf{A}\mathbf{x} \Longrightarrow y_i = \sum_{j=1}^{n} a_{ij} x_j, \tag{6.3}$$

matrix-matrix multiplication

$$\mathbf{A} = \mathbf{B}\mathbf{C} \Longrightarrow a_{ij} = \sum_{k=1}^{n} b_{ik} c_{kj}, \tag{6.4}$$

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

$$c = \mathbf{y}^T \mathbf{z} \Longrightarrow c = \sum_{j=1}^{n} y_j z_j, \tag{6.5}$$

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

$$\mathbf{A} = \mathbf{y}\mathbf{z}^T \Longrightarrow a_{ij} = y_i z_j, \tag{6.6}$$

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 $1 \leq i \leq n$. From these definitions, one can derive several important relations, of which the so-called Cauchy-Schwartz inequality is of great importance for many algorithms. For any $\mathbf{x}$ and $\mathbf{y}$ being real-valued or complex-valued quantities, the inner product space satisfies

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

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, the inner product space satisfies

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

Proofs can be found in for example Ref. [28]. 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 $\mathbf{x} \neq 0$, we define the relative error as

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

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 $\mathbf{x} \neq 0$. We refer the reader to the text of Golub and Van Loan [28] 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.

6.3 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 [18] and Barton and Nackman [19]. 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.

Fig. 6.1 Segmentation fault, again and again! Alas, this is a situation you will most likely end up in, unless you initialize, access, allocate and deallocate properly your arrays. Many program development environments such as Dev C++ at `www.bloodshed.net` provide debugging possibilities. 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.)

6.3.1 Declaration of fixed-sized vectors and matrices

In the program below we discuss some 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 subfunction(). 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.

```
 int main()
{
  int k,m, row = 3, col = 5;
  int  vec[5];   // line a
  int  matr[3][5]; // line b
  // Fill in vector vec
  for (k = 0; k < col; k++) vec[k] = k;
  // fill in matr
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++) matr[m][k] = m + 10*k;
  }
  // write out the vector
  cout << `` Content of vector vec:'' << endl;
  for (k = 0; k < col; k++){
     cout << vec[k] << endl;
  }
  // Then write out the matrix
  cout << `` Content of matrix matr:'' << endl;
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++){
       cout << matr[m][k] << endl;
     }
  }
  subfunction(row, col, vec, matr); // line c
  return 0;
} // end main function

void subfunction(int row, int col, int vec[], int matr[][5]); // line d
{
  int k, m;
  // write out the vector
  cout << `` Content of vector vec in subfunction:'' << endl;
  for (k = 0; k < col; k++){
     cout << vec[k] << endl;
  }
  // Then write out the matrix
  cout << `` Content of matrix matr in subfunction:'' << endl;
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++){
       cout << matr[m][k] << endl;
     }
  }
} // end of function subfunction
```

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 subfunction() are released when the execution returns to main().

6.3.2 Runtime Declarations of Vectors and Matrices in C++

We change thereafter our program in order to include dynamic allocation of arrays. 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 following program shows how we could change the previous one with static declarations to dynamic allocation of arrays.

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

   cout << `` Read in number of rows'' << endl; // line c
   cin >> row;
   cout << `` Read in number of columns'' << endl;
   cin >> col;

   vec = new int[col];                // line d
   matr = (int **)matrix(row,col,sizeof(int)); // line e
   // Fill in vector vec
   for (k = 0; k < col; k++) vec[k] = k;
   // fill in matr
   for (m = 0; m < row; m++){
      for (k = 0; k < col ; k++) matr[m][k] = m + 10*k;
   }
   // write out the vector
   cout << `` Content of vector vec:'' << endl;
   for (k = 0; k < col; k++){
      cout << vec[k] << endl;
   }
   // Then write out the matrix
   cout << `` Content of matrix matr:'' << endl;
   for (m = 0; m < row; m++){
      for (k = 0; k < col ; k++){
        cout << matr[m][k] << endl;
      }
   }
   subfunction(row, col, vec, matr); // line f
   free_matrix((void **) matr); // line g
   delete vec[];
   return 0;
} // end main function

void subfunction(int row, int col, int vec[], int matr[][5]); // line h
{
   int k, m;
   // write out the vector
```

```
  cout << `` Content of vector vec in subfunction:'' << endl;
  for (k = 0; k < col; k++){
     cout << vec[k] << endl;
  }
  // Then write out the matrix
  cout << `` Content of matrix matr in subfunction:'' << endl;
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++){
       cout << matr[m][k] << endl;
     }
  }
} // end of function subfunction
```

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 with dimensionality [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.

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 the program below 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://folk.uio.no/compphys/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 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 $\mathbf{A}$ is stored in memory is shown in Fig. 6.2.

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` or Armadillo makes life much easier when dealing with matrices.

6.3.3 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 as 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.

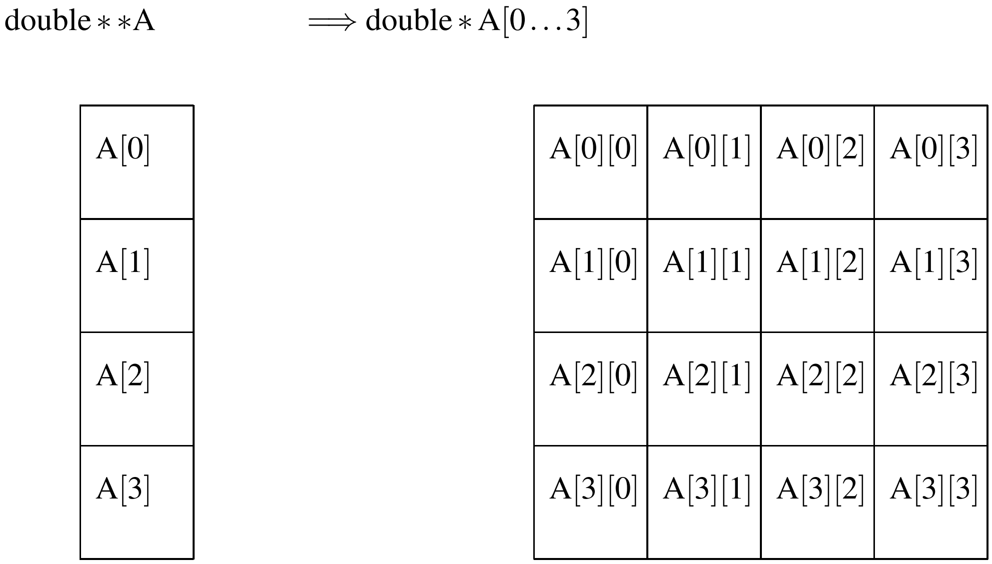

double ∗ ∗A
⟹ double ∗ A[0...3]
A[0]
A[1]
A[2]
A[3]
A[0][0] A[0][1] A[0][2] A[0][3]
A[1][0] A[1][1] A[1][2] A[1][3]
A[2][0] A[2][1] A[2][2] A[2][3]
A[3][0] A[3][1] A[3][2] A[3][3]

Fig. 6.2 Conceptual representation of the allocation of a matrix in C++.

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. (6.2). 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. 6.3 shows how a 3×3 matrix $\mathbf{A}$ is stored in both row-major and column-major ways.

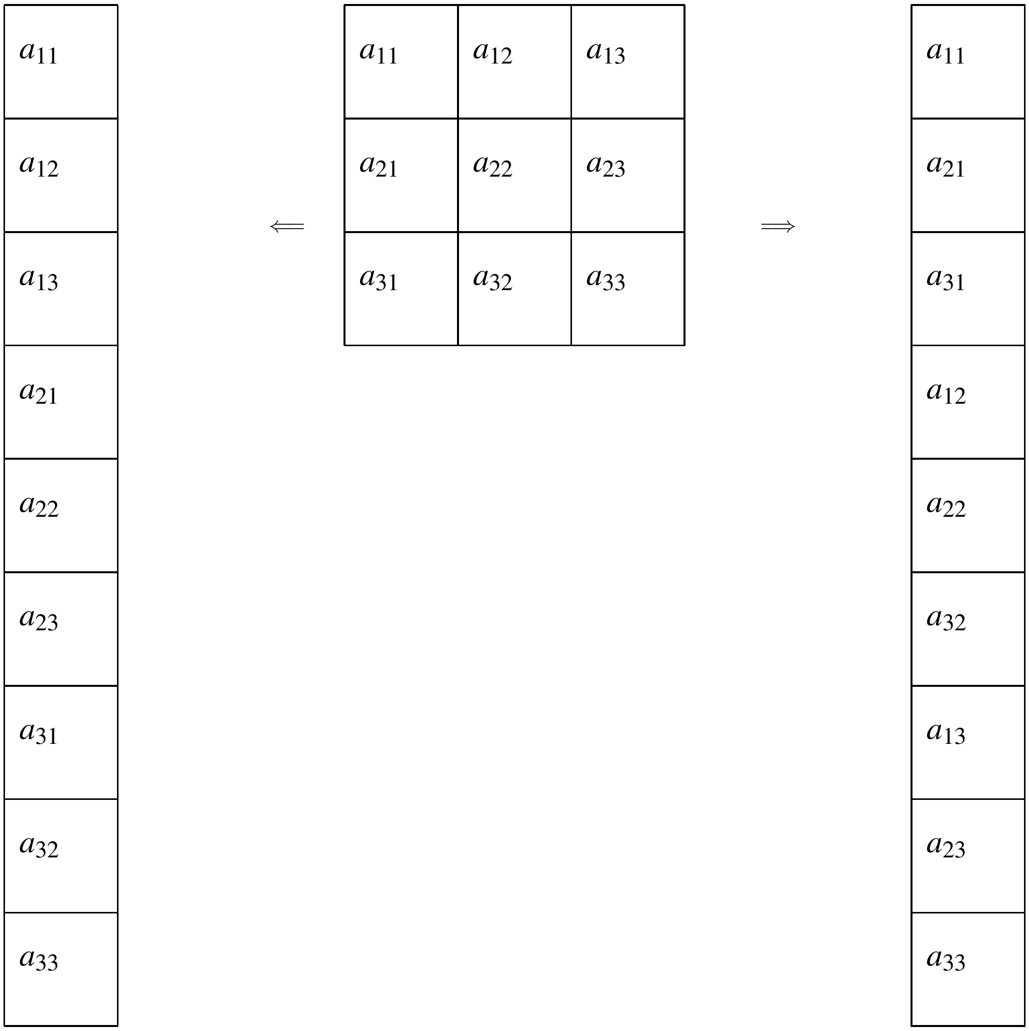


Fig. 6.3 Row-major storage of a matrix to the left (C++ way) and column-major to the right (Fortran way).

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. (6.4) 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]
    }
  }
}
```

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(k,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. (6.6) 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

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

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**.

6.3.3.1 Strassen's algorithm

As we have seen, the straightforward algorithm for matrix-matrix multiplication will require p multiplications and $p-1$ additions for each of the $m \times n$ elements. The total number of floating-point operations is then $mn(2p-1) \sim \mathscr{O}(mnp)$. When the matrices A and B can be divided into four equally sized blocks,

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \tag{6.7}$$

we get eight multiplications of smaller blocks,

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}. \tag{6.8}$$

Strassen discovered in 1968 how the number of multiplications could be reduced from eight to seven [28]. Following Strassen's approach we define some intermediates,

$$\begin{aligned} S_1 &= A_{21} + A_{22}, & T_1 &= B_{12} - B_{11}, \\ S_2 &= S_1 - A_{11}, & T_2 &= B_{22} - T_1, \\ S_3 &= A_{11} - A_{21}, & T_3 &= B_{22} - B_{12}, \\ S_4 &= A_{12} - S_2, & T_4 &= B_{21} - T_2, \end{aligned} \tag{6.9}$$

and need seven multiplications,

$$\begin{aligned}
P_1 &= A_{11}B_{11}, & U_1 &= P_1 + P_2,\\
P_2 &= A_{12}B_{21}, & U_2 &= P_1 + P_4,\\
P_3 &= S_1T_1, & U_3 &= U_2 + P_5,\\
P_4 &= S_2T_2, & U_4 &= U_3 + P_7,\\
P_5 &= S_3T_3, & U_5 &= U_3 + P_3,\\
P_6 &= S_4B_{22}, & U_6 &= U_2 + P_3,\\
P_7 &= A_{22}T_4, & U_7 &= U_6 + P_6,
\end{aligned} \tag{6.10}$$

to find the resulting C matrix as

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} U_1 & U_7 \\ U_4 & U_5 \end{bmatrix}. \tag{6.11}$$

In spite of the seemingly additional work, we have reduced the number of multiplications from eight to seven. Since the multiplications are the computational bottleneck compared to addition and subtraction, the number of flops are reduced.

In the case of square $n \times n$ matrices with n equal to a power of two, $n = 2^m$, the divided blocks will have $\frac{n}{2} = 2^{m-1}$. Letting $f(m)$ be the number of flops needed for the full matrix and applying Strassen recursively we find the total number of flops to be

$$f(m) = 7f(m-1) = 7^2 f(m-2) = \cdots = 7^m f(0), \tag{6.12}$$

where $f(0)$ is the one floating-point operation needed for multiplication of two numbers (two $2^0 \times 2^0$ matrices). For large matrices this can prove efficient, yielding a much better scaling,

$$\mathscr{O}\left(7^m\right) = \mathscr{O}\left(2^{\log_2 7^m}\right) = \mathscr{O}\left(2^{m \log_2 7}\right) = \mathscr{O}\left(n^{\log_2 7}\right) \approx \mathscr{O}\left(n^{2.807}\right), \tag{6.13}$$

effectively saving $7/8 = 12.5\%$ each time it is applied.

6.3.3.2 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.

6.4 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 [24,28] 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 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 $x_i = a+ih$, with $i = 0,1,\dots,n+1$. The step size is then given by $h = (b-a)/(n+1)$ with $n \in \mathbb{N}$. For the internal grid points $i = 1,2,\dots n$ 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 $i = 1,2,\dots,n$. 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 $\mathbf{u} = (u_1,u_2,\dots,u_n)^T$ and $\mathbf{f}(\mathbf{u}) = f(x_1,x_2,\dots,x_n,u_1,u_2,\dots,u_n)^T$ we can rewrite the differential equation including the boundary conditions as a system of linear equations with a large number of unknowns

$$\mathbf{A}\mathbf{u} = \mathbf{f}(\mathbf{u}). \tag{6.14}$$

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

$$\mathbf{A}\mathbf{u} = \mathbf{f}, \tag{6.15}$$

which is nothing but a simple linear equation with a tridiagonal matrix $\mathbf{A}$. We will solve such a system of equations in subsection 6.4.3.

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

$$\mathbf{A}=\begin{pmatrix} \frac{2}{h^2}+x_1^2 & -\frac{1}{h^2} & & & & \\ -\frac{1}{h^2} & \frac{2}{h^2}+x_2^2 & -\frac{1}{h^2} & & & \\ & -\frac{1}{h^2} & \frac{2}{h^2}+x_3^2 & -\frac{1}{h^2} & & \\ & \dots & \dots & \dots & \dots & \dots \\ & & & -\frac{1}{h^2} & \frac{2}{h^2}+x_{n-1}^2 & -\frac{1}{h^2} \\ & & & & -\frac{1}{h^2} & \frac{2}{h^2}+x_n^2 \end{pmatrix}, \tag{6.16}$$

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 7. These lecture notes contain however several other examples of rewriting mathematical expressions into matrix problems. In chapter 5 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.

6.4.1 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

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

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

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

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

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

or

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= y_1 \\ a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + a_{24}^{(2)}x_4 &= w_2^{(2)} \\ a_{32}^{(2)}x_2 + a_{33}^{(2)}x_3 + a_{34}^{(2)}x_4 &= w_3^{(2)} \\ a_{42}^{(2)}x_2 + a_{43}^{(2)}x_3 + a_{44}^{(2)}x_4 &= w_4^{(2)}, \end{aligned} \tag{6.17}$$

with the new coefficients

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

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

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

with a new right-hand side given by

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

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

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

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

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

This set of $n-1$ elimations leads us to Eq. (6.17), 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 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, that is $a_{21}a_{12}/a_{11}$. Assume also that a_{22} is 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 [28], 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 [24] 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 **A** via the product of the diagonal matrix elements 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 **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 note 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 **A**.

6.4.2 LU Decomposition of a Matrix

A frequently used form of Gaussian elimination is L(ower)U(pper) factorization 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 **A** given in Eq. (6.1). The LU decomposition method means that we can rewrite this matrix as the product of two matrices **L** and **U** where

$$\mathbf{A} = \mathbf{LU} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix}. \tag{6.18}$$

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

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

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 $\mathbf{A}$ 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. [28].

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 l_{41} in $\mathbf{L}$ and $\mathbf{U}$. 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 can all be evaluated by means of the results from the first column and the elements of $\mathbf{A}$. 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 $\mathbf{A}$ can be used to store the calculated matrix elements of $\mathbf{L}$ and $\mathbf{U}$. This saves memory.

We can generalize this procedure into three equations

$$\begin{aligned}
i<j: &\quad l_{i1}u_{1j}+l_{i2}u_{2j}+\dots+l_{ii}u_{ij} = a_{ij}\\
i=j: &\quad l_{i1}u_{1j}+l_{i2}u_{2j}+\dots+l_{ii}u_{jj} = a_{ij}\\
i>j: &\quad l_{i1}u_{1j}+l_{i2}u_{2j}+\dots+l_{ij}u_{jj} = a_{ij}
\end{aligned}$$

which gives the following algorithm:
Calculate the elements in $\mathbf{L}$ and $\mathbf{U}$ 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}

$$u_{jj} = a_{jj} - \sum_{k=1}^{j-1} l_{jk}u_{kj}. \tag{6.19}$$

- Finally, calculate the elements $l_{ij}, i > j$

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \right), \tag{6.20}$$

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 fulfill 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. [30]

$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

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

$$u_{22} = a_{11} - l_{21} = 1 - 10^{20},$$

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 $\varepsilon_M \sim 10^{-16}$ 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

$$\mathbf{U} = \begin{pmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{pmatrix}.$$

If we multiply the matrices **LU** we have

$$\begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix} \begin{pmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{pmatrix} = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 0 \end{pmatrix} \neq \mathbf{A}.$$

We do not get back the original matrix **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 **A**. The key point to notice is that Eqs. (6.19) and (6.20) 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 **A** it would have sufficed to interchange the two rows and start the LU decomposition with

$$\mathbf{A} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

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. [28],

$$\mathbf{LU} = \mathbf{A} + \mathbf{H},$$

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 **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 $|u_{ij}| \leq \varepsilon_M$ 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.

6.4.2.1 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

$$L_{ij}^T = L_{ji}.$$

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)
$$L_{ii} = \left(A_{ii} - \sum_{k=0}^{i-1} L_{ik}^2 \right)^{1/2}.$$
- 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 [28] 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.

6.4.3 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

$$\mathbf{Ax} \equiv \mathbf{LUx} = \mathbf{w}. \tag{6.21}$$

This equation can be calculated in two steps

$$\mathbf{Ly} = \mathbf{w}; \qquad \mathbf{Ux} = \mathbf{y}. \tag{6.22}$$

To show that this is correct we use to the LU decomposition to rewrite our system of linear equations as

$$\mathbf{LUx} = \mathbf{w},$$

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

$$\mathbf{Ux} = \mathbf{L}^{-1}\mathbf{w} = \mathbf{y},$$

which yields the intermediate step

$$\mathbf{L}^{-1}\mathbf{w} = \mathbf{y}$$

and multiplying with $\mathbf{L}$ on both sides we reobtain Eq. (6.22). As soon as we have $\mathbf{y}$ we can obtain $\mathbf{x}$ through $\mathbf{Ux} = \mathbf{y}$.

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.

6.4.4 Inverse of a Matrix and the Determinant

The basic definition of the determinant of **A** is

$$det\{\mathbf{A}\} = \sum_p (-1)^p a_{1p_1} \cdot a_{2p_2} \cdots a_{np_n},$$

where the sum runs over all permutations p of the indices $1,2,\dots,n$, altogether $n!$ terms. To calculate the inverse of $\mathbf{A}$ 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 $\mathbf{A}$ is then constructed as the transpose of 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 $\mathbf{A}$ in Eq. (6.18) 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 $\mathbf{L}$ equal 1. Thus the determinant can be written

$$det\{\mathbf{A}\} = \prod_{k=1}^{N} u_{kk}.$$

The inverse is slightly more difficult. However, with an LU decomposed matrix this reduces to solving a set of linear equations. To see this, we recall that if the inverse exists then

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I},$$

the identity matrix. With an LU decomposed matrix we can rewrite the last equation as

$$\mathbf{LUA}^{-1} = \mathbf{I}.$$

If we assume that the first column (that is column 1) of the inverse matrix can be written as a vector with unknown entries

$$\mathbf{A}_1^{-1} = \begin{pmatrix} a_{11}^{-1} \\ a_{21}^{-1} \\ \dots \\ a_{n1}^{-1} \end{pmatrix},$$

then we have a linear set of equations

$$\mathbf{LU}\begin{pmatrix} a_{11}^{-1} \\ a_{21}^{-1} \\ \dots \\ a_{n1}^{-1} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \dots \\ 0 \end{pmatrix}.$$

In a similar way we can compute the unknow entries of the second column,

$$\mathbf{LU}\begin{pmatrix} a_{12}^{-1} \\ a_{22}^{-1} \\ \dots \\ a_{n2}^{-1} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ \dots \\ 0 \end{pmatrix},$$

and continue till we have solved all n sets of linear equations.

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 the sets of linear equations.

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 which use these functions. We list first the C++ code.

http://folk.uio.no/compphys/programs/chapter06/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 linear equations 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 following program and reads

http://folk.uio.no/compphys/programs/chapter06/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. Alternatively, you could have used either Blitz++ or Armadillo.

The corresponding Fortran program for the inverse of a matrix reads

http://folk.uio.no/compphys/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.

6.4.4.1 Scattering Equation and Principal Value Integrals via Matrix Inversion

In quantum mechanics, it is often common to rewrite Schrödinger's equation in momentum space, after having made a so-called partial wave expansion of the interaction. We will not go into the details of these expressions but limit ourselves to study the equivalent problem for so-called scattering states, meaning that the total energy of two particles which collide is larger than or equal zero. The benefit of rewriting the equation in momentum space, after having performed a Fourier transformation, is that the coordinate space equation, being an integro-differantial equation, is transformed into an integral equation. The latter can be solved by standard matrix inversion techniques. Furthermore, the results of solving these equation can be related directly to experimental observables like the scattering phase shifts. The latter tell us how much the incoming two-particle wave function is modified by a collision. Here we take a more technical stand and consider the technical aspects of solving an integral equation with a principal value.

For scattering states, $E > 0$, the corresponding equation to solve is the so-called Lippman-Schwinger equation. This is an integral equation where we have to deal with the amplitude $R(k,k')$ (reaction matrix) defined through the integral equation

$$R_l(k,k') = V_l(k,k') + \frac{2}{\pi}\mathscr{P}\int_0^\infty dqq^2 V_l(k,q)\frac{1}{E-q^2/m}R_l(q,k'), \tag{6.23}$$

where the total kinetic energy of the two incoming particles in the center-of-mass system is

$$E = \frac{k_0^2}{m}. \tag{6.24}$$

The symbol $\mathscr{P}$ indicates that Cauchy's principal-value prescription is used in order to avoid the singularity arising from the zero of the denominator. We will discuss below how to solve this problem. Equation (6.23) represents then the problem you will have to solve numerically. The interaction between the two particles is given by a partial-wave decomposed version $V_l(k,k')$, where l stands for a quantum number like the orbital momentum. We have assumed that interaction does not coupled to partial waves with different orbital momenta. The variables k and k' are the outgoing and incoming relative momenta of the two interacting particles.

The matrix $R_l(k,k')$ relates to the experimental the phase shifts δ_l through its diagonal elements as

$$R_l(k_0,k_0) = -\frac{tan\delta_l}{mk_0}, \tag{6.25}$$

where m is the reduced mass of the interacting particles. Furthemore, the interaction between the particles, V, carries

In order to solve the Lippman-Schwinger equation in momentum space, we need first to write a function which sets up the integration points. We need to do that since we are going to approximate the integral through

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} w_i f(x_i),$$

where we have fixed N integration points through the corresponding weights w_i and points x_i. These points can for example be determined using Gaussian quadrature.

The principal value in Eq. (6.23) is 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 use the calculus relation from the previous section

$$\int_{-\infty}^{\infty} \frac{dk}{k-k_0} = 0,$$

or

$$\int_0^{\infty} \frac{dk}{k^2-k_0^2} = 0.$$

We can use this to express a principal values integral as

$$\mathscr{P}\int_0^{\infty} \frac{f(k)dk}{k^2-k_0^2} = \int_0^{\infty} \frac{(f(k)-f(k_0))dk}{k^2-k_0^2}, \tag{6.26}$$

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.

We can then use the trick in Eq. (6.26) to rewrite Eq. (6.23) as

$$R(k,k') = V(k,k') + \frac{2}{\pi}\int_0^{\infty} dq \frac{q^2 V(k,q)R(q,k') - k_0^2 V(k,k_0)R(k_0,k')}{(k_0^2-q^2)/m}. \tag{6.27}$$

We are interested in obtaining $R(k_0,k_0)$, since this is the quantity we want to relate to experimental data like the phase shifts.

How do we proceed in order to solve Eq. (6.27)?

1. Using the mesh points k_j and the weights ω_j, we can rewrite Eq. (6.27) as

$$R(k,k') = V(k,k') + \frac{2}{\pi}\sum_{j=1}^{N} \frac{\omega_j k_j^2 V(k,k_j)R(k_j,k')}{(k_0^2-k_j^2)/m} - \frac{2}{\pi}k_0^2 V(k,k_0)R(k_0,k')\sum_{n=1}^{N}\frac{\omega_n}{(k_0^2-k_n^2)/m}. \tag{6.28}$$

This equation contains now the unknowns $R(k_i,k_j)$ (with dimension $N\times N$) and $R(k_0,k_0)$.

2. We can turn Eq. (6.28) into an equation with dimension $(N+1)\times(N+1)$ with an integration domain which contains the original mesh points k_j for $j = 1,N$ and the point which corresponds to the energy k_0. Consider the latter as the 'observable' point. The mesh points become then k_j for $j = 1,n$ and $k_{N+1} = k_0$.
3. With these new mesh points we define the matrix

$$A_{i,j} = \delta_{i,j} - V(k_i,k_j)u_j, \tag{6.29}$$

where δ is the Kronecker δ and

$$u_j = \frac{2}{\pi}\frac{\omega_j k_j^2}{(k_0^2-k_j^2)/m} \qquad j = 1,N \tag{6.30}$$

and

$$u_{N+1} = -\frac{2}{\pi}\sum_{j=1}^{N}\frac{k_0^2\omega_j}{(k_0^2-k_j^2)/m}. \quad (6.31)$$

The first task is then to set up the matrix A for a given k_0. This is an $(N+1)\times(N+1)$ matrix. It can be convenient to have an outer loop which runs over the chosen observable values for the energy k_0^2/m. *Note that all mesh points k_j for $j = 1,N$ must be different from k_0. Note also that $V(k_i,k_j)$ is an $(N+1)\times(N+1)$ matrix.*

4. With the matrix A we can rewrite Eq. (6.28) as a matrix problem of dimension $(N+1)\times(N+1)$. All matrices R, A and V have this dimension and we get

$$A_{i,l}R_{l,j} = V_{i,j}, \quad (6.32)$$

or just

$$AR = V. \quad (6.33)$$

5. Since we already have defined A and V (these are stored as $(N+1)\times(N+1)$ matrices) Eq. (6.33) involves only the unknown R. We obtain it by matrix inversion, i.e.,

$$R = A^{-1}V. \quad (6.34)$$

Thus, to obtain R, we need to set up the matrices A and V and invert the matrix A. With the inverse A^{-1} we perform a matrix multiplication with V and obtain R.

With R we can in turn evaluate the phase shifts by noting that

$$R(k_{N+1},k_{N+1}) = R(k_0,k_0), \quad (6.35)$$

and we are done.

6.4.4.2 Inverse of the Vandermonde Matrix

In chapter 3 we discussed 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 expression 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

$$f(x) \approx a_0 + a_1x + a_2x^2 + \dots + a_nx^n.$$

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

$$\mathbf{Xa} = \mathbf{f},$$

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 discussed in chapter 3, together with spline interpolation discussed in the next section, 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. Spline interpolation will be discussed in the next section.

6.4.5 Tridiagonal Systems of Linear Equations

We start with the linear set of equations from Eq. (6.15), viz

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

where $\mathbf{A}$ is a tridiagonal matrix which we rewrite as

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{pmatrix}$$

where a,b,c are one-dimensional arrays of length $1:n$. In the example of Eq. (6.15) the arrays a and c are equal, namely $a_i = c_i = -1/h^2$. We can rewrite Eq. (6.15) as

$$\mathbf{Au} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \dots \\ \dots \\ \dots \\ f_n \end{pmatrix}.$$

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 $i = 1,2,\dots,n$. 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 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 10.

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

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix},$$

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 that 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 $x_i = a + ih$, 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.

6.5 Spline Interpolation

Cubic spline interpolation is among one of the most 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 cubic spline method to be described below. The linear equation solver we developed in the previous section for tridiagonal matrices can be reused for spline interpolation.

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 $x_0, x_1, \ldots x_n$ arranged so that $x_0 < x_1 < x_2 < \ldots x_{n-1} < x_n$ (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

$$s(x) = \begin{cases} s_0(x) = a_0x + b_0 & x \in [x_0, x_1) \\ s_1(x) = a_1x + b_1 & x \in [x_1, x_2) \\ \dots & \dots \\ s_{n-1}(x) = a_{n-1}x + b_{n-1} & x \in [x_{n-1}, x_n] \end{cases} \tag{6.36}$$

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 cubic spline function. Assume that we have in addition 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.,

$$s_{i-1}(x_i) = y_i = s_i(x_i), \tag{6.37}$$

with $1 \leq i \leq n-1$. In total we have n polynomials of the type

$$s_i(x) = a_{i0} + a_{i1}x + a_{i2}x^2 + a_{i3}x^3, \tag{6.38}$$

yielding $4n$ coefficients to determine. Every subinterval provides in addition two conditions

$$y_i = s(x_i), \tag{6.39}$$

and

$$y_{i+1} = s(x_{i+1}), \tag{6.40}$$

to be fulfilled. If we also assume that s' and s'' are continuous, then

$$s'_{i-1}(x_i) = s'_i(x_i), \tag{6.41}$$

yields $n-1$ conditions. Similarly,

$$s''_{i-1}(x_i) = s''_i(x_i), \tag{6.42}$$

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

$$s''_i(x_i) = f_i, \tag{6.43}$$

and

$$s''_i(x_{i+1}) = f_{i+1}, \tag{6.44}$$

and setting up a straight line between f_i and f_{i+1} we have

$$s''_i(x) = \frac{f_i}{x_{i+1} - x_i}(x_{i+1} - x) + \frac{f_{i+1}}{x_{i+1} - x_i}(x - x_i), \tag{6.45}$$

and integrating twice one obtains

$$s_i(x) = \frac{f_i}{6(x_{i+1} - x_i)}(x_{i+1} - x)^3 + \frac{f_{i+1}}{6(x_{i+1} - x_i)}(x - x_i)^3 + c(x - x_i) + d(x_{i+1} - x). \tag{6.46}$$

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

$$
\begin{aligned}
s_i(x) = \quad & \tfrac{f_i}{6(x_{i+1}-x_i)}(x_{i+1}-x)^3 + \tfrac{f_{i+1}}{6(x_{i+1}-x_i)}(x-x_i)^3 \\
& + \big(\tfrac{y_{i+1}}{x_{i+1}-x_i} - \tfrac{f_{i+1}(x_{i+1}-x_i)}{6}\big)(x-x_i) + \big(\tfrac{y_i}{x_{i+1}-x_i} - \tfrac{f_i(x_{i+1}-x_i)}{6}\big)(x_{i+1}-x). \qquad (6.47)
\end{aligned}
$$

How to determine the values of the second derivatives f_i and f_{i+1}? We use the continuity assumption of the first derivatives

$$
s'_{i-1}(x_i) = s'_i(x_i), \qquad (6.48)
$$

and set $x = x_i$. Defining $h_i = x_{i+1} - x_i$ we obtain finally the following expression

$$
h_{i-1}f_{i-1} + 2(h_i + h_{i-1})f_i + h_i f_{i+1} = \frac{6}{h_i}(y_{i+1}-y_i) - \frac{6}{h_{i-1}}(y_i - y_{i-1}), \qquad (6.49)
$$

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

$$
\begin{bmatrix}
u_1 & h_1 & 0 & \dots & & & & \\
h_1 & u_2 & h_2 & 0 & \dots & & & \\
0 & h_2 & u_3 & h_3 & 0 & \dots & & \\
\dots & & \dots & \dots & \dots & \dots & \dots & \\
 & \dots & & & 0 & h_{n-3} & u_{n-2} & h_{n-2} \\
 & & & & & 0 & h_{n-2} & u_{n-1}
\end{bmatrix}
\begin{bmatrix}
f_1 \\ f_2 \\ f_3 \\ \dots \\ f_{n-2} \\ f_{n-1}
\end{bmatrix}
=
\begin{bmatrix}
v_1 \\ v_2 \\ v_3 \\ \dots \\ v_{n-2} \\ v_{n-1}
\end{bmatrix}. \qquad (6.50)
$$

Note that this is a set of tridiagonal equations and can be solved through only $O(n)$ operations.

It is easy to write your own program for the cubic spline method when you have written a slover for tridiagonal equations. We split the program into two tasks, one which finds the polynomial approximation and one which uses the polynomials approximation to find an interpolated value for a function. These functions are included in the programs of this chapter, see the codes cubicpsline.cpp and cubicsinterpol.cpp. Alternatively, you can solve exercise 6.4!

6.6 Iterative Methods

Till now we have dealt with so-called direct solvers such as Gaussian elimination and LU decomposition. Iterative solvers offer another strategy and are much used in partial differential equations. We start with a guess for the solution and then iterate till the solution does not change anymore.

6.6.1 Jacobi's method

It is a simple method for solving

$$
\hat{A}\mathbf{x} = \mathbf{b},
$$

where $\hat{A}$ is a matrix and $\mathbf{x}$ and $\mathbf{b}$ are vectors. The vector $\mathbf{x}$ is the unknown.

It is an iterative scheme where we start with a guess for the unknown, and after $k+1$ iterations we have

$$
\mathbf{x}^{(k+1)} = \hat{D}^{-1}(\mathbf{b} - (\hat{L} + \hat{U})\mathbf{x}^{(k)}),
$$

with $\hat{A} = \hat{D} + \hat{U} + \hat{L}$ and $\hat{D}$ being a diagonal matrix, $\hat{U}$ an upper triangular matrix and $\hat{L}$ a lower triangular matrix.

If the matrix $\hat{A}$ is positive definite or diagonally dominant, one can show that this method will always converge to the exact solution.

We can demonstrate Jacobi's method by a 4×4 matrix problem. We assume a guess for the initial vector elements, labeled $x_i^{(0)}$. This guess represents our first iteration. The new values are obtained by substitution

$$\begin{aligned}
x_1^{(1)} =& \ (b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)} - a_{14}x_4^{(0)})/a_{11}\\
x_2^{(1)} =& \ (b_2 - a_{21}x_1^{(0)} - a_{23}x_3^{(0)} - a_{24}x_4^{(0)})/a_{22}\\
x_3^{(1)} =& \ (b_3 - a_{31}x_1^{(0)} - a_{32}x_2^{(0)} - a_{34}x_4^{(0)})/a_{33}\\
x_4^{(1)} =& \ (b_4 - a_{41}x_1^{(0)} - a_{42}x_2^{(0)} - a_{43}x_3^{(0)})/a_{44},
\end{aligned}$$

which after $k+1$ iterations result in

$$\begin{aligned}
x_1^{(k+1)} =& \ (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11}\\
x_2^{(k+1)} =& \ (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22}\\
x_3^{(k+1)} =& \ (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)} - a_{34}x_4^{(k)})/a_{33}\\
x_4^{(k+1)} =& \ (b_4 - a_{41}x_1^{(k)} - a_{42}x_2^{(k)} - a_{43}x_3^{(k)})/a_{44},
\end{aligned}$$

We can generalize the above equations to

$$x_i^{(k+1)} = (b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k)})/a_{ii}$$

or in an even more compact form as

$$\mathbf{x}^{(k+1)} = \hat{D}^{-1}(\mathbf{b} - (\hat{L}+\hat{U})\mathbf{x}^{(k)}),$$

with $\hat{A} = \hat{D}+\hat{U}+\hat{L}$ and $\hat{D}$ being a diagonal matrix, $\hat{U}$ an upper triangular matrix and $\hat{L}$ a lower triangular matrix.

6.6.2 Gauss-Seidel

Our 4×4 matrix problem

$$\begin{aligned}
x_1^{(k+1)} =& \ (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11}\\
x_2^{(k+1)} =& \ (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22}\\
x_3^{(k+1)} =& \ (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)} - a_{34}x_4^{(k)})/a_{33}\\
x_4^{(k+1)} =& \ (b_4 - a_{41}x_1^{(k)} - a_{42}x_2^{(k)} - a_{43}x_3^{(k)})/a_{44},
\end{aligned}$$

can be rewritten as

$$\begin{aligned}
x_1^{(k+1)} =& \ (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11}\\
x_2^{(k+1)} =& \ (b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22}\\
x_3^{(k+1)} =& \ (b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)} - a_{34}x_4^{(k)})/a_{33}\\
x_4^{(k+1)} =& \ (b_4 - a_{41}x_1^{(k+1)} - a_{42}x_2^{(k+1)} - a_{43}x_3^{(k+1)})/a_{44},
\end{aligned}$$

which allows us to utilize the preceding solution (forward substitution). This improves normally the convergence behavior and leads to the Gauss-Seidel method!

We can generalize these equations to the following form

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j>i} a_{ij}x_j^{(k)} - \sum_{j<i} a_{ij}x_j^{(k+1)}\right), \quad i = 1,2,\dots,n.$$

The procedure is generally continued until the changes made by an iteration are below some tolerance.

The convergence properties of the Jacobi method and the Gauss-Seidel method depend on the matrix $\hat{A}$. These methods converge when the matrix is symmetric positive-definite, or is strictly or irreducibly diagonally dominant. Both methods sometimes converge even if these conditions are not satisfied.

6.6.3 Successive over-relaxation

We can rewrite the above in a slightly more formal way and extend the methods to what is called successive over-relaxation. Given a square system of n linear equations with unknown $\mathbf{x}$:

$$\hat{A}\mathbf{x} = \mathbf{b}$$

where:

$$\hat{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Then A can be decomposed into a diagonal component D, and strictly lower and upper triangular components L and U:

$$\hat{A} = \hat{D} + \hat{L} + \hat{U},$$

where

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

The system of linear equations may be rewritten as:

$$(D + \omega L)\mathbf{x} = \omega\mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x}$$

for a constant $\omega > 1$. The method of successive over-relaxation is an iterative technique that solves the left hand side of this expression for x, using previous value for x on the right hand side. Analytically, this may be written as:

$$\mathbf{x}^{(k+1)} = (D + \omega L)^{-1}\big(\omega\mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x}^{(k)}\big).$$

However, by taking advantage of the triangular form of $(D + \omega L)$, the elements of $x^{(k+1)}$ can be computed sequentially using forward substitution:

$$x_i^{(k+1)} = (1 - \omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j>i} a_{ij}x_j^{(k)} - \sum_{j<i} a_{ij}x_j^{(k+1)}\right), \quad i = 1,2,\dots,n.$$

The choice of relaxation factor is not necessarily easy, and depends upon the properties of the coefficient matrix. For symmetric, positive-definite matrices it can be proven that $0 < \omega < 2$ will lead to convergence, but we are generally interested in faster convergence rather than just convergence.

6.6.4 Conjugate Gradient Method

The success of the Conjugate Gradient method for finding solutions of non-linear problems is based on the theory for of conjugate gradients for linear systems of equations. It belongs to the class of iterative methods for solving problems from linear algebra of the type

$$\hat{\mathbf{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}.$$

In the iterative process we end up with a problem like

$$\hat{\mathbf{r}} = \hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}},$$

where $\hat{\mathbf{r}}$ is the so-called residual or error in the iterative process.

The residual is zero when we reach the minimum of the quadratic equation

$$P(\hat{\mathbf{x}}) = \frac{1}{2}\hat{\mathbf{x}}^T\hat{\mathbf{A}}\hat{\mathbf{x}} - \hat{\mathbf{x}}^T\hat{\mathbf{b}},$$

with the constraint that the matrix $\hat{\mathbf{A}}$ is positive definite and symmetric. If we search for a minimum of the quantum mechanical variance, then the matrix $\hat{\mathbf{A}}$, which is called the Hessian, is given by the second-derivative of the variance. This quantity is always positive definite. If we vary the energy, the Hessian may not always be positive definite.

In the Conjugate Gradient method we define so-called conjugate directions and two vectors $\hat{\mathbf{s}}$ and $\hat{\mathbf{t}}$ are said to be conjugate if

$$\hat{\mathbf{s}}^T\hat{\mathbf{A}}\hat{\mathbf{t}} = 0.$$

The philosophy of the Conjugate Gradient method is to perform searches in various conjugate directions of our vectors $\hat{\mathbf{x}}_i$ obeying the above criterion, namely

$$\hat{\mathbf{x}}_i^T\hat{\mathbf{A}}\hat{\mathbf{x}}_j = 0.$$

Two vectors are conjugate if they are orthogonal with respect to this inner product. Being conjugate is a symmetric relation: if $\hat{\mathbf{s}}$ is conjugate to $\hat{\mathbf{t}}$, then $\hat{\mathbf{t}}$ is conjugate to $\hat{\mathbf{s}}$.

An example is given by the eigenvectors of the matrix

$$\hat{\mathbf{v}}_i^T\hat{\mathbf{A}}\hat{\mathbf{v}}_j = \lambda\hat{\mathbf{v}}_i^T\hat{\mathbf{v}}_j,$$

which is zero unless $i = j$.

Assume now that we have a symmetric positive-definite matrix $\hat{\mathbf{A}}$ of size $n \times n$. At each iteration $i+1$ we obtain the conjugate direction of a vector

$$\hat{\mathbf{x}}_{i+1} = \hat{\mathbf{x}}_i + \alpha_i\hat{\mathbf{p}}_i.$$

We assume that $\hat{\mathbf{p}}_i$ is a sequence of n mutually conjugate directions. Then the $\hat{\mathbf{p}}_i$ form a basis of R^n and we can expand the solution $\hat{\mathbf{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}$ in this basis, namely

$$\hat{\mathbf{x}} = \sum_{i=1}^{n}\alpha_i\hat{\mathbf{p}}_i.$$

The coefficients are given by

$$\mathbf{Ax} = \sum_{i=1}^{n} \alpha_i \mathbf{Ap}_i = \mathbf{b}.$$

Multiplying with $\hat{\mathbf{p}}_k^T$ from the left gives

$$\hat{\mathbf{p}}_k^T \hat{\mathbf{A}}\hat{\mathbf{x}} = \sum_{i=1}^{n} \alpha_i \hat{\mathbf{p}}_k^T \hat{\mathbf{A}}\hat{\mathbf{p}}_i = \hat{\mathbf{p}}_k^T \hat{\mathbf{b}},$$

and we can define the coefficients α_k as

$$\alpha_k = \frac{\hat{\mathbf{p}}_k^T \hat{\mathbf{b}}}{\hat{\mathbf{p}}_k^T \hat{\mathbf{A}}\hat{\mathbf{p}}_k}$$

If we choose the conjugate vectors $\hat{\mathbf{p}}_k$ carefully, then we may not need all of them to obtain a good approximation to the solution $\hat{\mathbf{x}}$. So, we want to regard the conjugate gradient method as an iterative method. This also allows us to solve systems where n is so large that the direct method would take too much time.

We denote the initial guess for $\hat{\mathbf{x}}$ as $\hat{\mathbf{x}}_0$. We can assume without loss of generality that

$$\hat{\mathbf{x}}_0 = 0,$$

or consider the system

$$\hat{\mathbf{A}}\hat{\mathbf{z}} = \hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}}_0,$$

instead.

One can show that the solution $\hat{\mathbf{x}}$ is also the unique minimizer of the quadratic form

$$f(\hat{\mathbf{x}}) = \frac{1}{2}\hat{\mathbf{x}}^T \hat{\mathbf{A}}\hat{\mathbf{x}} - \hat{\mathbf{x}}^T \hat{\mathbf{x}}, \quad \hat{\mathbf{x}} \in \mathbf{R}^n.$$

This suggests taking the first basis vector $\hat{\mathbf{p}}_1$ to be the gradient of f at $\hat{\mathbf{x}} = \hat{\mathbf{x}}_0$, which equals

$$\hat{\mathbf{A}}\hat{\mathbf{x}}_0 - \hat{\mathbf{b}},$$

and $\hat{\mathbf{x}}_0 = 0$ it is equal $-\hat{\mathbf{b}}$. The other vectors in the basis will be conjugate to the gradient, hence the name conjugate gradient method.

Let $\hat{\mathbf{r}}_k$ be the residual at the k-th step:

$$\hat{\mathbf{r}}_k = \hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}}_k.$$

Note that $\hat{\mathbf{r}}_k$ is the negative gradient of f at $\hat{\mathbf{x}} = \hat{\mathbf{x}}_k$, so the gradient descent method would be to move in the direction $\hat{\mathbf{r}}_k$. Here, we insist that the directions $\hat{\mathbf{p}}_k$ are conjugate to each other, so we take the direction closest to the gradient $\hat{\mathbf{r}}_k$ under the conjugacy constraint. This gives the following expression

$$\hat{\mathbf{p}}_{k+1} = \hat{\mathbf{r}}_k - \frac{\hat{\mathbf{p}}_k^T \hat{\mathbf{A}}\hat{\mathbf{r}}_k}{\hat{\mathbf{p}}_k^T \hat{\mathbf{A}}\hat{\mathbf{p}}_k}\hat{\mathbf{p}}_k.$$

We can also compute the residual iteratively as

$$\hat{\mathbf{r}}_{k+1} = \hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}}_{k+1},$$

which equals

$$\hat{\mathbf{b}} - \hat{\mathbf{A}}(\hat{\mathbf{x}}_k + \alpha_k \hat{\mathbf{p}}_k),$$

or

$$(\hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}}_k) - \alpha_k \hat{\mathbf{A}}\hat{\mathbf{p}}_k,$$

which gives

$$\hat{\mathbf{r}}_{k+1} = \hat{\mathbf{r}}_k - \hat{\mathbf{A}}\hat{\mathbf{p}}_k,$$

If we consider finding the minimum of a function f using Newton's method, that implies a search for a zero of the gradient of a function. Near a point x_i we have to second order

$$f(\hat{\mathbf{x}}) = f(\hat{\mathbf{x}}_i) + (\hat{\mathbf{x}} - \hat{\mathbf{x}}_i)\nabla f(\hat{\mathbf{x}}_i)\frac{1}{2}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_i)\hat{\mathbf{A}}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_i)$$

giving

$$\nabla f(\hat{\mathbf{x}}) = \nabla f(\hat{\mathbf{x}}_i) + \hat{\mathbf{A}}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_i).$$

In Newton's method we set $\nabla f = 0$ and we can thus compute the next iteration point

$$\hat{\mathbf{x}} - \hat{\mathbf{x}}_i = \hat{\mathbf{A}}^{-1}\nabla f(\hat{\mathbf{x}}_i).$$

Subtracting this equation from that of $\hat{\mathbf{x}}_{i+1}$ we have

$$\hat{\mathbf{x}}_{i+1} - \hat{\mathbf{x}}_i = \hat{\mathbf{A}}^{-1}(\nabla f(\hat{\mathbf{x}}_{i+1}) - \nabla f(\hat{\mathbf{x}}_i)).$$

6.7 A vector and matrix class

We end this chapter by presenting a class which allows to manipulate one- and two-dimensional arrays. However, before we proceed, we would like to come with some general recommendations. Although it is useful to write your own classes, like the one included here, in general these classes may not be very efficient from a computational point of view. There are several libraries which include many interesting array features that allow us to write more compact code. The latter has the advantage that the code is lost likely easier to debug in case of errors (obviously assuming that the library is functioning correctly). Furthermore, if the proper functionalities are included, the final code may closely resemble the mathematical operations we wish to perform, increasing considerably the readability of our program. And finally, the code is in almost all casesmuch faster than the one we wrote!

In particular, we would like to recommend the C++ linear algebra library Armadillo, see `http://arma.sourceforgenet`. For those of you who are familiar with compiled programs like Matlab, the syntax is deliberately similar. Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions. Various matrix decompositions are provided through optional integration with LAPACK, or one of its high performance drop-in replacements (such as the multi-threaded MKL or ACML libraries). The selected examples included here show some examples on how to declare arrays and rearrange arrays or perform mathematical operations on say vectors or matrices. The first example here defines two random matrices of dimensionality 10×10 and performs a matrix-matrix multiplication using the *dgemm* function of the library BLAS.

Simple matrix-matrix multiplication of two random matrices

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

int main(int argc, char** argv)
  {
```

```
 mat A = randu<mat>(10,10);
 mat B = randu<mat>(10,10);
 // Matrix-matrix multiplication
 cout << A*B << endl;
 return 0;
 }
```

In the next example we compute the determinant of a 5×5 matrix, its inverse and perform thereafter several operations on various matrices.

Determinant and inverse of a matrix

```
#include <iostream>
#include "armadillo"
using namespace arma;
using namespace std;

int main(int argc, char** argv)
 {
 cout << "Armadillo version: " << arma_version::as_string() << endl;
 mat A;
 // Hard coding of the matrix
 // endr indicates "end of row"
 A << 0.165300 << 0.454037 << 0.995795 << 0.124098 << 0.047084 << endr
   << 0.688782 << 0.036549 << 0.552848 << 0.937664 << 0.866401 << endr
   << 0.348740 << 0.479388 << 0.506228 << 0.145673 << 0.491547 << endr
   << 0.148678 << 0.682258 << 0.571154 << 0.874724 << 0.444632 << endr
   << 0.245726 << 0.595218 << 0.409327 << 0.367827 << 0.385736 << endr;
 // .n_rows = number of rows
 // .n_cols = number of columns
 cout << "A.n_rows = " << A.n_rows << endl;
 cout << "A.n_cols = " << A.n_cols << endl;
 // Print the matrix A
 A.print("A =");
 // Computation of the determinant
 cout << "det(A) = " << det(A) << endl;
 // inverse
 cout << "inv(A) = " << endl << inv(A) << endl;
 // save to disk
 A.save("MatrixA.txt", raw_ascii);
 // Define a new matrix B which reads A from file
 mat B;
 B.load("MatrixA.txt");
 B += 5.0*A;
 B.print("The matrix B:");
 // generate the identity matrix
 mat C = eye<mat>(4,4);
 // transpose of B
 cout << "trans(B) =" << endl;
 // maximum from each column (traverse along rows)
 cout << "max(B) =" << endl;
 cout << max(B) << endl;
 // sum of all elements B
 cout << "sum(sum(B)) = " << sum(sum(B)) << endl;
 cout << "accu(B) = " << accu(B) << endl;
 // trace = sum along diagonal
 cout << "trace(B) = " << trace(B) << endl;
 // random matrix -- values are uniformly distributed in the [0,1] interval
 mat D = randu<mat>(4,4);
 D.print("Matrix D:");
 // sum of four matrices (no temporary matrices are created)
 mat E = A+B + C + D;
```

```
  F.print("F:");
  return 0;
}
```

For more examples, please consult the online manual, see `http://arma.sourceforgenet`.

6.7.1 How to construct your own matrix-vector class

The rest of this section shows how one can build a matrix-vector class. We first give an example of a function which use the header file `Array.h`.

```
#include "Array.h"

#include <iostream>
using namespace std;

int main(){

  // Create an array with (default) nrows = 1, ncols = 1:
  Array<double> v1;

  // Redimension the array to have length n:
  int n1 = 3;
  v1.redim(n1);

  // Extract the length of the array:
  const int length = v1.getLength();

  // Create a narray of specific length:
  int n2 = 5;
  Array<double> v2(n2);

  // Create an array as a copy of another one:
  Array<double> v5(v1);

  // Assign the entries in an array:
  v5(0) = 3.0;
  v5(1) = 2.5;
  v5(2) = 1.0;

  for(int i=0; i<3; i++){
    cout << v5(i) << endl;
  }

  // Extract the ith component of an array:
  int i = 2;
  double value = v5(1);
  cout << "value: " << value << endl;

  // Set an array equal another one:
  Array<double> v6 = v5;

  for(int i=0; i<3; i++){
    v1(i) = 1.0;
    v2(i) = 2.0;
  }

  // Create a two-dimensional array (matrix):
```

```
  Array<double> matrix(2, 2);

  // Fill the array:
  matrix(0,0) = 1;
  matrix(0,1) = 2;
  matrix(1,0) = 3;
  matrix(1,1) = 4;

  // Get the entries in the array:
  cout << "\nMatrix: " << endl;
  for(int i=0; i<2; i++){
    for(int j=0; j<2; j++){
      cout << matrix(i,j) << " ";
    }
    cout << endl;
  }

  // Assign an entry of the matrix to a variable:
  double scalar = matrix(0,0);
  const double b = matrix(1,1);


  Array<double> vector(2);
  vector(0) = 1.0;
  vector(1) = 2.0;

  Array<double> v = vector;
  Array<double> A = matrix;
  Array<double> u(2);

  cout << "\nMatrix: " << endl;
  for(int i=0; i<2; i++){
    for(int j=0; j<2; j++){
      cout << matrix(i,j) << " ";
    }
    cout << endl;
  }

  Array<double> a(2,2);
  a(1,1) = 5.0;

  // Arithmetic operations with arrays using a
  // syntax close to the mathematical language
  Array<double> w = v1 + 2.0*v2;

  // Create multidimensional matrices and assign values to them:
  int N = 3;
  Array<double> multiD; multiD.redim(N,N,N);
  for(int i=0; i<N; i++){
    for(int j=0; j<N; j++){
      for(int k=0; k<N; k++){
        cout << "multD(i,j,k) = " << multiD(i,j,k) << endl;
      }
    }
  }

  multiD(1,2,3) = 4.0;
  cout << "multiD(1,2,3) = " << multiD(1,2,3) << endl;
}
```

The header file follows here

```
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
#include <sstream>
#include <iomanip>
#include <cstdlib>

using namespace std;

template<class T>
class Array{
 private:
  static const int MAXDIM = 6;
  T *data ;      /**> One-dimensional array of data.*/
  int size[MAXDIM];  /**> Size of each dimension.*/
  int ndim;      /**> Number of dimensions occupied. */
  int length;     /**> Total number of entries.*/

  int dx1, dx2, dx3, dx4, dx5;

  void allocate(int ni=0, int nj=0, int nk=0, int nl=0, int nm=0, int nn=0){
   ndim = MAXDIM;

   // Set the number of entries in each dimension.
   size[0]=ni;
   size[1]=nj;
   size[2]=nk;
   size[3]=nl;
   size[4]=nm;
   size[5]=nn;


   // Set the number of dimensions used.
   if(size[5] == 0)
    ndim--;
   if(size[4] == 0)
    ndim--;
   if(size[3] == 0)
    ndim--;
   if(size[2] == 0)
    ndim--;
   if(size[1] == 0)
    ndim--;
   if(size[0] == 0){
    ndim   = 0;
    length  = 0;
    data  = NULL;
   }else{
    try{
     int i;

     // Set the length (total number of entries) of the one-dimensional array.
     length = 1;
     for(i=0; i<ndim; i++)
      length *= size[i];

      data = new T[length];

      dx1 =   size[0];
```

```
          dx2 = dx1*size[1];
          dx3 = dx2*size[2];
          dx4 = dx3*size[3];
          dx5 = dx4*size[4];

      }catch(std::bad_alloc&){
        std::cerr << "Array::allocate -- unable to allocate array of length " << length <<
              std::endl;
        exit(1);
      }
    }

  }

public:


  /**
  * @brief Constructor with default arguments.
  *
  * Creates an array with one or two-dimensions.
  *
  * @param int nrows. Number of rows in the array.
  * @param int ncolsd. Number of columns in the array.
  **/
  Array(int ni=0, int nj=0, int nk=0, int nl=0, int nm=0, int nn=0){
    // Allocate memory
    allocate(ni,nj,nk,nl,nm,nn);
  } // end constructor


  //! Constructor
  Array(T* array, int ndim_, int size_[]){
    ndim = ndim_;

    length = 1;
    int i;
    for(i=0; i<ndim; i++){
      size[i] = size_[i]; // Copy only the ndim entries. The rest is zero by default.
      length *= size[i];
    }

    // Now when we known the length, we should not forget to allocate memory!!!!
    data = new T[length];

    // Copy the entries from array to data:
    for(i=0; i<length; i++){
      data[i] = array[i];
    }

  } // End constructor.




  //! Copy constructor
  Array(const Array<T>& array);

  //! Destructor
  ~Array();
```

```
/**
* @brief Checks the validity of the indexing.
* @param i, an integer for indexing the rows.
* @param j, an integer for indexing the columns.
**/
bool indexOk(int i, int j=0) const;

/**
* @brief Change the dimensions of an array.
* @param ni number of entries in the first dimension.
* @param nj number of entries in the second dimension.
* @param nk number of entries in the third dimension.
* @param nl number of entries in the fourth dimension.
* @param nm number of entries in the fifth dimension.
* @param nn number of entries in the sixth dimension.
**/
bool redim(int ni, int nj=0, int nk=0, int nl=0, int nm=0, int nn=0);

/**
* @return The total number of entries in the array, i.e., the sum of the entries in all
    the dimensions.
**/
int getLength()const{return length;}


/**
* @return The number of rows in a matrix.
**/
int getRows() const {return size[0];}

/**
* @return Returns the number of columns in a matrix.
**/
int getColumns() const {return size[1];}


/** @brief Gives the number of entries in a dimension.
*
* @param i An integer from 0 to 5 indicating the dimension we want to explore.
* @return size[i] An integer for the number of elements in the dimension number i.
**/
int dimension(int i) const{return size[i];}


/**
* The number of dimensions in the array.
**/
int getNDIM()const{return ndim;}


/**
* @return A constant pointer to the array of data.
* This function can be used to interface C++ with Python/Fortran/C.
**/
const T* getPtr() const;


/**
* @return A pointer to the array of data.
* This function can be used to interface C++ with Python/Fortran/C.
```

```
**/
T* getPtr();


/**
* @return A pointer to an array with information on the length of each dimension.
**/
int* getPtrSize();




/*************************************************************/
/*                OPERATORS                  */
/*************************************************************/

//! Assignment operator
Array<T>& operator=(const Array<T>& array);

//! Sum operator
Array<T> operator+(const Array<T>& array);

//! Substraction operator
Array<T> operator-(const Array<T>& array)const; /// w=u-v;


//! Multiplication operator
//Array<T> operator*(const Array<T>& array);


//! Assigment by addition operator
Array<T>& operator+=(const Array<T>& w);


//! Assignment by substraction operator
Array<T>& operator-=(const Array<T>& w);


//! Assignment by scalar product operator
Array<T>& operator*=(double scalar);

//! Assignment by division operator
Array<T>& operator/=(double scalar);

//! Index operators
const T& operator()(int i)const;
const T& operator()(int i, int j)const;
const T& operator()(int i, int j, int k)const;
const T& operator()(int i, int j, int k, int l)const;
const T& operator()(int i, int j, int k, int l, int m)const;
const T& operator()(int i, int j, int k, int l, int m, int n)const;

T& operator()(int i);
T& operator()(int i, int j);
T& operator()(int i, int j, int k);
T& operator()(int i, int j, int k, int l);
T& operator()(int i, int j, int k, int l, int m);
T& operator()(int i, int j, int k, int l, int m, int n);
```

```
   /**************************************************************/
   /*           FRIEND FUNCTIONS                 */
   /**************************************************************/
   //! Unary operator +
   template <class T2>
   friend Array<T> operator+ (const Array<T>&);  // u = + v

   //! Unary operator -
   template <class T2>
   friend Array<T> operator-(const Array<T>&);  // u = - v




   /**
   * Premultiplication by a floating point number:
   * \f$\mathbf{u} = a \mathbf{v}\f$,
   * where \f$a\f$ is a scalar and \f$\mathbf{v}\f$ is a array.
   **/
   template <class T2>
   friend Array<T> operator*(double, const Array<T>&); // u = a*v

   /**
   * Postmultiplication by a floating point number:
   * \f$\mathbf{u} = \mathbf{v} a\f$,
   * where \f$a\f$ is a scalar and \f$\mathbf{v}\f$ is a array.
   **/
   template <class T2>
   friend Array<T> operator*(const Array<T>&, double); // u = v*a


   /**
   * Division of the entries of a array by a scalar.
   **/
   template <class T2>
   friend Array<T> operator/(const Array<T>&, double); // u = v/a





};

#include "Array.cpp"


// Destructor
template <class T>
inline Array<T>::~Array(){delete[] data;}

// Index operators
template <class T>
inline const T& Array<T>::operator()(int i)const {
  #if CHECKBOUNDS_ON
   indexOk(i);
  #endif
  return data[i];
}
```

```
template <class T>
inline const T& Array<T>::operator()(int i, int j)const {
  #if CHECKBOUNDS_ON
    indexOk(i,j);
  #endif

  return data[i + j*dx1];
}

template <class T>
inline const T& Array<T>::operator()(int i, int j, int k)const {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k);
  #endif

  return data[i + j*dx1 + k*dx2];
}


template <class T>
inline const T& Array<T>::operator()(int i, int j, int k, int l)const {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k,l);
  #endif

  return data[i + j*dx1 + k*dx2 + l*dx3];
}


template <class T>
inline const T& Array<T>::operator()(int i, int j, int k, int l, int m)const {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k,l, m);
  #endif

  return data[i + j*dx1 + k*dx2 + l*dx3 + m*dx4];
}


template <class T>
inline const T& Array<T>::operator()(int i, int j, int k, int l, int m, int n)const {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k,l,m,n);
  #endif

  return data[i + j*dx1 + k*dx2 + l*dx3 + m*dx4 + n*dx5];
}

template <class T>
inline T& Array<T>::operator()(int i) {
  #if CHECKBOUNDS_ON
    indexOk(i);
  #endif
  return data[i];
}


template <class T>
inline T& Array<T>::operator()(int i, int j) {
  #if CHECKBOUNDS_ON
    indexOk(i,j);
  #endif

  return data[i + j*dx1];
}


template <class T>
inline T& Array<T>::operator()(int i, int j, int k) {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k);
  #endif

  return data[i + j*dx1 + k*dx2];
}


template <class T>
inline T& Array<T>::operator()(int i, int j, int k, int l) {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k,l);
  #endif

  return data[i + j*dx1 + k*dx2 + l*dx3];
}


template <class T>
inline T& Array<T>::operator()(int i, int j, int k, int l, int m) {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k,l,m);
  #endif

  return data[i + j*dx1 + k*dx2 + l*dx3 + m*dx4];
}


template <class T>
inline T& Array<T>::operator()(int i, int j, int k, int l, int m, int n) {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k,l,m,n);
  #endif

  return data[i + j*dx1 + k*dx2 + l*dx3 + m*dx4 + n*dx5];
}



template <class T>
inline const T* Array<T>::getPtr() const {return data;}



template <class T>
inline T* Array<T>::getPtr(){return data; }


template <class T>
inline int* Array<T>::getPtrSize(){return size;}


// template <class T>
// inline int Array<T>::dim()const{return ndim;}


/*******************************************************************/
/*          IMPLEMENTATION OF FRIEND FUNCTIONS        */
/*******************************************************************/

/*******************************************************************/
/*         (Arithmetic) Unary operators          */
/*******************************************************************/
//! Unary operator +
template <class T>
inline Array<T> operator+(const Array<T>& v){ // u = + v
  return v;
}


//! Unary operator -
template <class T>
inline Array<T> operator-(const Array<T>& v){ // u = - v
  return Array<T>(v.size[0],v.size[1]) -v;
}


//! Postmultiplication operator
template <class T>
inline Array<T> operator*(const Array<T>& v, double scalar){ // u = v*a
  return Array<T>(v) *= scalar;
}


//! Premultiplication operator.
template <class T>
inline Array<T> operator*(double scalar, const Array<T>& v){ // u = a*v
  return v*scalar; // Note the call to postmultiplication operator defined above
}


//! Division of the entries in a array by a scalar
template <class T>
inline Array<T> operator/(const Array<T>& v, double scalar){
  if(!scalar) std::cout << "Division by zero!" << std::endl;
  return (1.0/scalar)*v;
}

#endif
```

6.8 Exercises

6.1. The aim of this exercise is to write your own Gaussian elimination code.

1. 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

$$\mathbf{A}\mathbf{x} = \mathbf{w}.$$

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.

2. 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?

6.2. 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)

$$L_{ii} = \left(A_{ii} - \sum_{k=0}^{i-1} L_{ik}^2 \right)^{1/2}. \tag{6.51}$$

- within the loop over i, introduce a new loop which goes from $j = i+1$ to $n-1$ and calculate

$$L_{ji} = \frac{1}{L_{ii}} \left(A_{ij} - \sum_{k=0}^{i-1} L_{ik} l_{jk} \right). \tag{6.52}$$

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

$$\mathbf{A} = \begin{pmatrix} 6 & 3 & 2 \\ 3 & 2 & 1 \\ 2 & 1 & 1 \end{pmatrix} \tag{6.53}$$

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.

6.3. In this exercise we are going to solve the one-dimensional Poisson equation in terms of linear equations.

1. 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,

$$\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}+\frac{\partial^2\phi}{\partial z^2}=-\frac{\rho(x,y,z)}{\varepsilon_0},$$

whose solution we will discuss in chapter 10. 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

$$\frac{d^2\phi}{dx^2}=-\frac{\rho(x)}{\varepsilon_0},$$

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

$$\frac{1}{2}m_ev^2(x)-e\phi(x)=0.$$

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

$$\frac{d^2\phi}{dx^2}=\frac{I}{\varepsilon_0A}\left(\frac{m_e}{2e}\right)^{1/2}\phi^{-1/2}.$$

The solution of the above equation which satisfies the boundary conditions is

$$\phi=V_0\left(\frac{x}{d}\right)^{4/3},$$

with

$$I=\frac{4}{9}\frac{\varepsilon_0A}{d^2}\left(\frac{2e}{m_e}\right)^{1/2}V_0^{3/2}.$$

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

$$-u''(x) = f(x), \qquad x \in (0,1), \qquad u(0) = u(1) = 0.$$

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

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \qquad \text{for } i = 1, \ldots, n,$$

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

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & \ldots \\ 0 & -1 & 2 & -1 & 0 & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & & -1 & 2 & -1 \\ 0 & \ldots & & 0 & -1 & 2 \end{pmatrix}$$

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.

2. We can rewrite our matrix $\mathbf{A}$ in terms of one-dimensional vectors a, b, c of length $1:n$. Our linear equation reads

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \ldots & \ldots & \ldots \\ a_2 & b_2 & c_2 & \ldots & \ldots & \ldots \\ & a_3 & b_3 & c_3 & \ldots & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \ldots \\ \ldots \\ \ldots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \ldots \\ \ldots \\ \ldots \\ \tilde{b}_n \end{pmatrix}.$$

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 $i = 1, 2, \ldots, n$. 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

$$\varepsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right),$$

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.

3. 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.

6.8.1 Solution

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. 6.4 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 almost impossible to distinguish the numerical solution from the analytical one, as shown in Fig. 6.5. It is therefore instructive to study the relative error, which we display in Table 6.2 as function of the step length $h = 1/(n+1)$.

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

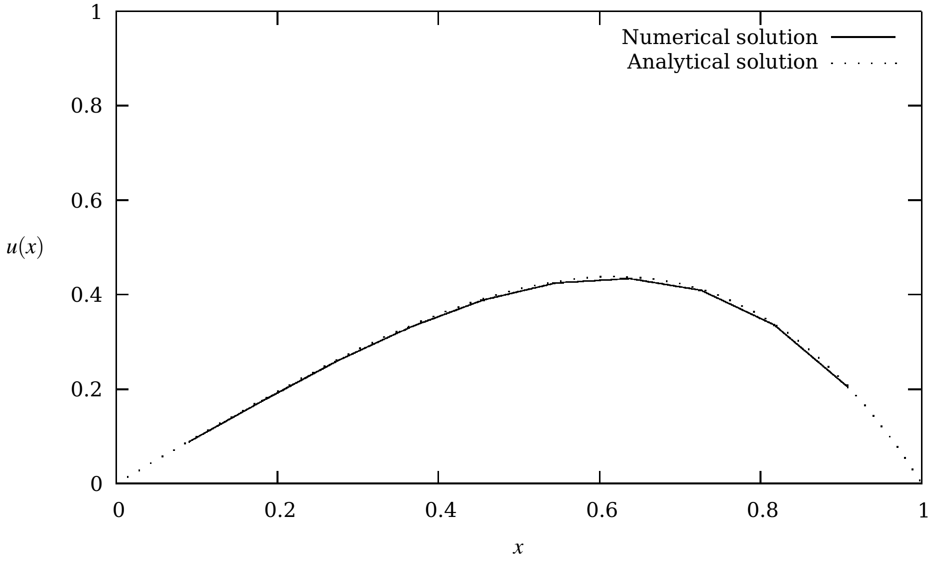


Fig. 6.4 Numerical solution obtained with $n = 10$ compared with the analytical solution.

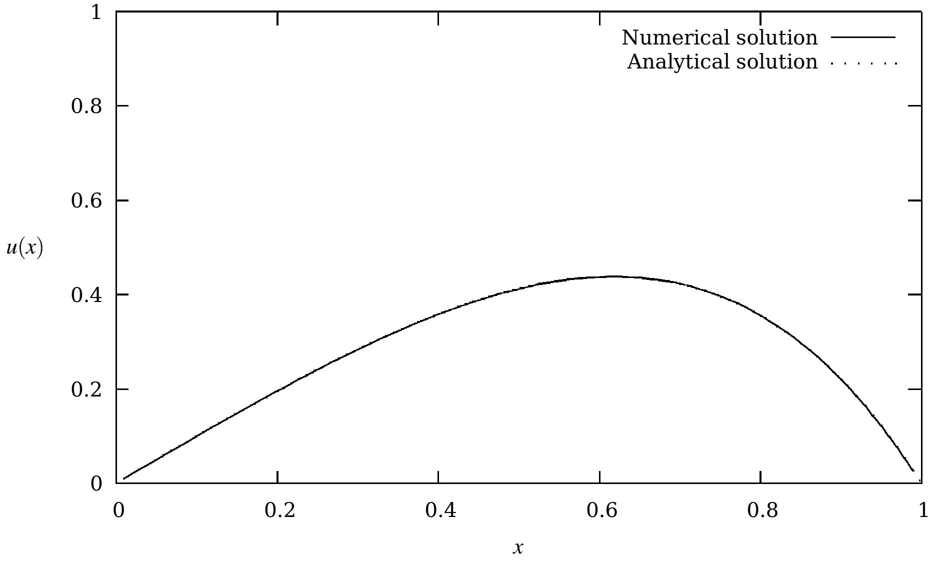


Fig. 6.5 Numerical solution obtained with $n = 10$ compared with the analytical solution.

Table 6.2 log_{10} values for the relative error and the step length h computed at $x = 0.5$.

n	$log_{10}(h)$	$\varepsilon_i = log_{10}\left(\lvert(v_i - u_i)/u_i\rvert\right)$
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

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.

6.4. Write your own code for performing the cubic spline interpolation using either Blitz++ or Armadillo. Alternatively you can use the vector-matrix class included in this text.

6.5. Write your own code for the LU decomposition using the same libraries as in the previous exercise. Find also the number of floating point operations.

6.6. Solve exercise 6.3 by writing a code which implements both the iterative Jacobi method and the Gauss-Seidel method. Study carefully the number of iterations needed to achieve the exact result.

6.7. Extend thereafter your code for the iterative Jacobi method to a parallel version and compare with the results from the previous exercise.

6.8. Write your own code for the Conjugate gradient method.

6.9. Write your own code for matrix-matrix multiplications using Strassen's algorithm discussed in subsection 6.3.3.1 and compare the speed of your program with the matrix-matrix multiplication provided by the Armadillo library.

Chapter 7
Eigensystems

Abstract We present here two methods for solving directly eigenvalue problems using similarity transformations. One is the familiar Jacobi rotation method while the second method is based on transforming the matrix to tridiagonal form using Householder's algorithm. We discuss also so-called power methods and conclude with a discussion of iterative algorithms. These are particularly interesting for eigenvalue problems of large dimnesionality.

7.1 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. [28].

7.2 Eigenvalue problems

Let us consider the matrix **A** of dimension n. The eigenvalues of **A** are defined through the matrix equation

$$\mathbf{A}\mathbf{x}^{(\nu)} = \lambda^{(\nu)}\mathbf{x}^{(\nu)}, \tag{7.1}$$

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

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n =& \; \lambda x_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n =& \; \lambda x_2 \\
\dots & \; \dots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n =& \; \lambda x_n.
\end{aligned}$$

We can rewrite Eq. (7.1) 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. The eigenvalues of a matrix $\mathbf{A}\in\mathbb{C}^{n\times n}$ are thus the n roots of its characteristic polynomial

$$P(\lambda)=det(\lambda\mathbf{I}-\mathbf{A}), \tag{7.2}$$

or

$$P(\lambda)=\prod_{i=1}^{n}(\lambda_i-\lambda). \tag{7.3}$$

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

$$det(\mathbf{A})=\lambda_1\lambda_2\dots\lambda_n,$$

the trace of $\mathbf{A}$ is $Tr(\mathbf{A})=\lambda_1+\lambda_2+\dots+\lambda_n$.

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. (7.1) 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.

7.3 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

$$\mathbf{D}=\left(\begin{array}{ccccccc}\lambda_1 & 0 & 0 & 0 & \dots & 0 & 0\\ 0 & \lambda_2 & 0 & 0 & \dots & 0 & 0\\ 0 & 0 & \lambda_3 & 0 & 0 & \dots & 0\\ \dots & \dots & \dots & \dots & \dots & \dots & \dots\\ 0 & \dots & \dots & \dots & \dots & \lambda_{n-1} & \\ 0 & \dots & \dots & \dots & \dots & 0 & \lambda_n\end{array}\right).$$

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. [28] 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 **B**.

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{x} \qquad \text{and} \qquad \mathbf{B} = \mathbf{S}^T\mathbf{A}\mathbf{S}.$$

We multiply the first equation on the left by $\mathbf{S}^T$ and insert $\mathbf{S}^T\mathbf{S} = \mathbf{I}$ between **A** and **x**. Then we get

$$(\mathbf{S^T}\mathbf{A}\mathbf{S})(\mathbf{S^T}\mathbf{x}) = \lambda \mathbf{S^T}\mathbf{x}, \tag{7.4}$$

which is the same as

$$\mathbf{B}\left(\mathbf{S^T}\mathbf{x}\right) = \lambda\left(\mathbf{S^T}\mathbf{x}\right).$$

The variable λ is an eigenvalue of **B** as well, but with eigenvector $\mathbf{S^T}\mathbf{x}$.

The basic philosophy is to

- either apply subsequent similarity transformations so that

$$\mathbf{S_N^T} \dots \mathbf{S_1^T}\mathbf{A}\mathbf{S_1} \dots \mathbf{S_N} = \mathbf{D}, \tag{7.5}$$

- or apply subsequent similarity transformations so that **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.

7.4 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 the matrix elements that differ from zero are given by

$$s_{kk} = s_{ll} = cos\theta, s_{kl} = -s_{lk} = -sin\theta, s_{ii} = -s_{ii} = 1 \qquad i \neq k \qquad i \neq l,$$

A similarity transformation

$$\mathbf{B} = \mathbf{S}^T\mathbf{A}\mathbf{S},$$

results in

$$
\begin{aligned}
b_{ii} &= a_{ii}, i \neq k, i \neq l \\
b_{ik} &= a_{ik}cos\theta - a_{il}sin\theta, i \neq k, i \neq l \\
b_{il} &= a_{il}cos\theta + a_{ik}sin\theta, i \neq k, i \neq l \\
b_{kk} &= a_{kk}cos^2\theta - 2a_{kl}cos\theta sin\theta + a_{ll}sin^2\theta \\
b_{ll} &= a_{ll}cos^2\theta + 2a_{kl}cos\theta sin\theta + a_{kk}sin^2\theta \\
b_{kl} &= (a_{kk} - a_{ll})cos\theta sin\theta + a_{kl}(cos^2\theta - sin^2\theta)
\end{aligned}
$$

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 until 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

$$\begin{pmatrix} b_{kk} & 0 \\ 0 & b_{ll} \end{pmatrix} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} a_{kk} & a_{kl} \\ a_{lk} & a_{ll} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}.$$

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 result means that the matrix $\mathbf{A}$ moves closer to diagonal form for each transformation.

Defining the quantities $\tan\theta = t = s/c$ and

$$\tau = \frac{a_{ll} - a_{kk}}{2a_{kl}},$$

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

$$||\mathbf{B}-\mathbf{A}||_F^2 = 4(1-c)\sum_{i=1,i\neq k,l}^{n}(a_{ik}^2+a_{il}^2)+\frac{2a_{kl}^2}{c^2}.$$

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 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

  $$\mathrm{off}(\mathbf{A}) = \sqrt{\sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n} a_{ij}^2} > \varepsilon.$$

  This is however a very time-comsuming test which can be replaced by the simpler test

  $$\max(a_{ij}^2) > \varepsilon.$$

- 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 $\mathbf{B} = \mathbf{S}(k,l,\theta)^T\mathbf{A}\mathbf{S}(k,l,\theta)$.
- Continue till

  $$\max(a_{ij}^2) \leq \varepsilon.$$

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.

The slow convergence is related to the fact that when a new rotation is performed, matrix elements which were previously zero, may change to non-zero values in the next rotation. To see this, consider the following simple example.

We specialize to a symmetric 3×3 matrix $\mathbf{A}$. We start the process as follows (assuming that $a_{23} = a_{32}$ is the largest non-diagonal matrix element) with $c = \cos\theta$ and $s = \sin\theta$

$$\mathbf{B} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{pmatrix}\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & c & s \\ 0 & -s & c \end{pmatrix}.$$

We will choose the angle θ in order to have $b_{23} = b_{32} = 0$. We get the new symmetric matrix

$$\mathbf{B} = \begin{pmatrix} a_{11} & a_{12}c - a_{13}s & a_{12}s + a_{13}c \\ a_{12}c - a_{13}s & a_{22}c^2 + a_{33}s^2 - 2a_{23}sc & (a_{22} - a_{33})sc + a_{23}(c^2 - s^2) \\ a_{12}s + a_{13}c & (a_{22} - a_{33})sc + a_{23}(c^2 - s^2) & a_{22}s^2 + a_{33}c^2 + 2a_{23}sc \end{pmatrix}.$$

Note that a_{11} is unchanged! As it should.

We have then

$$\begin{aligned} b_{11} &= a_{11} \\ b_{12} &= a_{12}cos\theta - a_{13}sin\theta, 1 \neq 2, 1 \neq 3 \\ b_{13} &= a_{13}cos\theta + a_{12}sin\theta, 1 \neq 2, 1 \neq 3 \\ b_{22} &= a_{22}cos^2\theta - 2a_{23}cos\theta sin\theta + a_{33}sin^2\theta \\ b_{33} &= a_{33}cos^2\theta + 2a_{23}cos\theta sin\theta + a_{22}sin^2\theta \\ b_{23} &= (a_{22} - a_{33})cos\theta sin\theta + a_{23}(cos^2\theta - sin^2\theta) \end{aligned}$$

We will fix the angle θ so that $b_{23} = 0$.

We get then a new matrix

$$\mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{12} & b_{22} & 0 \\ b_{13} & 0 & a_{33} \end{pmatrix}.$$

We repeat assuming that b_{12} is the largest non-diagonal matrix element and get a new matrix

$$\mathbf{C} = \begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{12} & b_{22} & 0 \\ b_{13} & 0 & b_{33} \end{pmatrix} \begin{pmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

We continue this process till all non-diagonal matrix elements are zero. It is easy to convince oneself that when performing the above operations, the matrix element b_{23} which was previously set to zero may become different from zero. This is one of the problems which slows down the Jacobi procedure. We leave this experience to the reader in form of a large numerical project at the end of this chapter.

An implementation of the above algorithm, normally referred to as the classical Jacobi algorithm, is exposed partially in the code here.

http://folk.uio.no/compphys/programs/chapter07/cpp/jacobi.cpp

```
/*
   Jacobi's method for finding eigenvalues
   eigenvectors of the symetric matrix A.

   The eigenvalues of A will be on the diagonal
   of A, with eigenvalue i being A[i][i].
   The j-th component of the i-th eigenvector
   is stored in R[i][j].

   A: input matrix (n x n)
   R: empty matrix for eigenvectors (n x n)
   n: dimention of matrices
*/
#include <iostream>
#include <cmath>
#include "jacobi.h"

void jacobi_method ( double ** A, double ** R, int n )
```

```
{
// Setting up the eigenvector matrix
 for ( int i = 0; i < n; i++ ) {
  for ( int j = 0; j < n; j++ ) {
   if ( i == j ) {
 R[i][j] = 1.0;
   } else {
 R[i][j] = 0.0;
   }
  }
 }

 int k, l;
 double epsilon = 1.0e-8;
 double max_number_terations = (double) n * (double) n * (double) n;
 int iterations = 0;
 double max_offdiag = maxoffdiag ( A, &k, &l, n );

 while ( fabs(max_offdiag) > epsilon && (double) iterations < max_number_iterations ) {
  max:offdiag = maxoffdiag ( A, &k, &l, n );
  rotate ( A, R, k, l, n );
  iterations++;
 }
 std::cout << "Number of iterations: " << iterations << "\n";
 return;
}
// Function to find the maximum matrix element. Can you figure out a more
// elegant algorithm?
double maxoffdiag ( double ** A, int * k, int * l, int n )
{
 double max = 0.0;

 for ( int i = 0; i < n; i++ ) {
  for ( int j = i + 1; j < n; j++ ) {
   if ( fabs(A[i][j]) > max ) {
 max = fabs(A[i][j]);
 *l = i;
 *k = j;
   }
  }
 }
 return max;
}
// Function to find the values of cos and sin
void rotate ( double ** A, double ** R, int k, int l, int n )
{
 double s, c;
 if ( A[k][l] != 0.0 ) {
  double t, tau;
  tau = (A[l][l] - A[k][k])/(2*A[k][l]);
  if ( tau > 0 ) {
   t = 1.0/(tau + sqrt(1.0 + tau*tau);
  } else {
   t = -1.0/( -tau + sqrt(1.0 + tau*tau);
  }

  c = 1/sqrt(1+t*t);
  s = c*t;
 } else {
  c = 1.0;
  s = 0.0;
 }
 double a_kk, a_ll, a_ik, a_il, r_ik, r_il;
 a_kk = A[k][k];
 a_ll = A[l][l];
 // changing the matrix elements with indices k and l
 A[k][k] = c*c*a_kk - 2.0*c*s*A[k][l] + s*s*a_ll;
 A[l][l] = s*s*a_kk + 2.0*c*s*A[k][l] + c*c*a_ll;
 A[k][l] = 0.0; // hard-coding of the zeros
 A[l][k] = 0.0;
 // and then we change the remaining elements
 for ( int i = 0; i < n; i++ ) {
  if ( i != k && i != l ) {
   a_ik = A[i][k];
   a_il = A[i][l];
   A[i][k] = c*a_ik - s*a_il;
   A[k][i] = A[i][k];
   A[i][l] = c*a_il + s*a_ik;
   A[l][i] = A[i][l];
  }
  // Finally, we compute the new eigenvectors
  r_ik = R[i][k];
  r_il = R[i][l];
  R[i][k] = c*r_ik - s*r_il;
  R[i][l] = c*r_il + s*r_ik;
 }
 return;
}
```

7.5 Similarity Transformations with Householder's method

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 diagonalizing a tridiagonal matrix is computational much faster than the corresponding diagonalization of a general symmetric matrix. Let us discuss the two steps in more detail.

7.5.1 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

$$a_{11}, a'_{22}, a''_{33} \dots a^{n-1}_{nn},$$

and off-diagonal matrix elements

$$e_1, e_2, e_3, \dots, e_{n-1}.$$

The resulting matrix reads

$$\mathbf{S}^T\mathbf{A}\mathbf{S} = \begin{pmatrix} a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & a'_{22} & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & a''_{33} & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & a^{(n-1)}_{n-1n-1} & e_{n-1} \\ 0 & \dots & \dots & \dots & \dots & e_{n-1} & a^{(n-1)}_{nn} \end{pmatrix}.$$

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 fulfills 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 $\mathbf{u}^T\mathbf{u} = 1$, that is its 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

$$P_{ij} = \delta_{ij} - 2u_iu_j,$$

where i and j range from 1 to $n-1$. Applying the transformation $\mathbf{S}_1$ results in

$$\mathbf{S}_1^T\mathbf{A}\mathbf{S}_1 = \begin{pmatrix} a_{11} & (\mathbf{P}\mathbf{v})^T \\ \mathbf{P}\mathbf{v} & \mathbf{A}' \end{pmatrix},$$

where $\mathbf{v^T} = \{a_{21}, a_{31}, \cdots, a_{n1}\}$ and $\mathbf{P}$ must satisfy $(\mathbf{P}\mathbf{v})^T = \{k, 0, 0, \cdots\}$. Then

$$\mathbf{P}\mathbf{v} = \mathbf{v} - 2\mathbf{u}(\mathbf{u}^T\mathbf{v}) = k\mathbf{e}, \tag{7.6}$$

with $\mathbf{e^T} = \{1, 0, 0, \dots 0\}$. 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

$$(\mathbf{P}\mathbf{v})^T \mathbf{P}\mathbf{v} = k^2 = \mathbf{v}^T\mathbf{v} = |\mathbf{v}|^2 = \sum_{i=2}^{n} a_{i1}^2,$$

which determines the constant $k = \pm v$. Now we can rewrite Eq. (7.6) 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

$$2(\mathbf{u}^T\mathbf{v})^2 = (v^2 \pm a_{21}v), \tag{7.7}$$

which finally determines

$$\mathbf{u} = \frac{\mathbf{v} - k\mathbf{e}}{2(\mathbf{u}^T\mathbf{v})}.$$

In solving Eq. (7.7) 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. It is not so difficult to implement Householder's algorithm, as demonstrated by the following code.

http://folk.uio.no/compphys/programs/chapter07/cpp/householder.cpp

```
  /*
  ** The function
  **           householder()
  ** perform a Housholder reduction of a real symmetric matrix
  ** a[][]. On output a[][] is replaced by the orthogonal matrix
  ** effecting the transformation. d[] returns the diagonal elements
  ** of the tri-diagonal matrix, and e[] the off-diagonal elements,
  ** with e[0] = 0.
  */
void householder(double **a, int n, double *d, double *e)
{
  register int l,k,j,i;
  double      scale,hh,h,g,f;

  for(i = n - 1; i > 0; i--) {
    l = i-1;
    h = scale= 0.0;
    if(l > 0) {
      for(k = 0; k <= l; k++)
        scale += fabs(a[i][k]);
        if(scale == 0.0)       // skip transformation
          e[i] = a[i][l];
        else {
        for(k = 0; k <= l; k++) {
          a[i][k] /= scale;   // used scaled a's for transformation
          h     += a[i][k]*a[i][k];
        }
        f     = a[i][l];
        g     = (f >= 0.0 ? -sqrt(h) : sqrt(h));
        e[i]  = scale*g;
        h    -= f * g;
        a[i][l] = f - g;
        f     = 0.0;
        for(j = 0;j <= l;j++) {
          a[j][i] = a[i][j]/h; // can be omitted if eigenvector not wanted
          g     = 0.0;
          for(k = 0; k <= j; k++) {
            g += a[j][k]*a[i][k];
          }
          for(k = j+1; k <= l; k++)
            g += a[k][j]*a[i][k];
          e[j]=g/h;
          f += e[j]*a[i][j];
        }
        hh=f/(h+h);
        for(j = 0; j <= l;j++) {
          f = a[i][j];
          e[j]=g=e[j]-hh*f;
          for(k = 0; k <= j; k++)
            a[j][k] -= (f*e[k]+g*a[i][k]);
        }
      } // end k-loop
    } // end if-loop for l > 1
    else {
      e[i]=a[i][l];
    }
    d[i]=h;
  } // end i-loop
  d[0] = 0.0;
  e[0] = 0.0;

      /* Contents of this loop can be omitted if eigenvectors not
  ** wanted except for statement d[i]=a[i][i];
      */

  for(i = 0; i < n; i++) {
    l = i-1;
    if(d[i]) {
      for(j = 0; j <= l; j++) {
        g= 0.0;
        for(k = 0; k <= l; k++) {
          g += a[i][k] * a[k][j];
        }
        for (k = 0; k <= l; k++) {
          a[k][j] -= g * a[k][i];
        }
      }
    }
    d[i]  = a[i][i];
    a[i][i] = 1.0;
    for(j = 0; j <= l; j++) {
      a[j][i]=a[i][j] = 0.0;
    }
  }
} // End: function householder()
```

7.5.2 Diagonalization of a Tridiagonal Matrix via Francis' Algorithm

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[1].

Before we discuss the algorithms, we note that the eigenvalues of a tridiagonal matrix can be obtained using the characteristic polynomial

$$P(\lambda) = det(\lambda \mathbf{I} - \mathbf{A}) = \prod_{i=1}^{n} (\lambda_i - \lambda),$$

with the matrix

$$\mathbf{A} - \lambda \mathbf{I} = \left(det \begin{array}{ccccccc} d_1 - \lambda & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & d_2 - \lambda & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & d_3 - \lambda & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & d_{N_{\mathrm{step}}-2} - \lambda & e_{N_{\mathrm{step}}-1} \\ 0 & \dots & \dots & \dots & \dots & e_{N_{\mathrm{step}}-1} & d_{N_{\mathrm{step}}-1} - \lambda \end{array} \right)$$

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 householder(double **a, int n, double d[], double e[])
         void francis(double d[], double[], int n, double **z)
Fortran: CALL householder(a, n, d, e)
         CALL francis(d, e, n, z)
```

The last step through the function *francis()* involves several technical details. Let us describe the basic idea in terms of a four-dimensional example. For more details, see Ref. [28], 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

[1] This section is not complete it will be finished end of fall 2009.

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.

7.6 Power Methods

We assume $\hat{A}$ can be diagonalized. Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the n eigenvalues (counted with multiplicity) of $\hat{A}$ and let $v_1, v_2, \ldots, v_n$ be the corresponding eigenvectors. We assume that λ_1 is the dominant eigenvalue, so that $|\lambda_1| > |\lambda_j|$ for $j > 1$.

The initial vector b_0 can be written:

$$b_0 = c_1 v_1 + c_2 v_2 + \cdots + c_m v_m.$$

If b_0 is chosen randomly (with uniform probability), then c_1 âĽ’ă 0 with probability 1. Now,

$$\begin{aligned} A^k b_0 &= c_1 A^k v_1 + c_2 A^k v_2 + \cdots + c_m A^k v_m \\ &= c_1 \lambda_1^k v_1 + c_2 \lambda_2^k v_2 + \cdots + c_m \lambda_m^k v_m \\ &= c_1 \lambda_1^k \left(v_1 + \tfrac{c_2}{c_1} \left(\tfrac{\lambda_2}{\lambda_1}\right)^k v_2 + \cdots + \tfrac{c_m}{c_1} \left(\tfrac{\lambda_m}{\lambda_1}\right)^k v_m \right). \end{aligned}$$

The expression within parentheses converges to v_1 because $|\lambda_j/\lambda_1| < 1$ for $j > 1$. On the other hand, we have

$$b_k = \frac{A^k b_0}{\|A^k b_0\|}.$$

Therefore, b_k converges to (a multiple of) the eigenvector v_1. The convergence is geometric, with ratio

$$\left|\frac{\lambda_2}{\lambda_1}\right|,$$

where λ_2 denotes the second dominant eigenvalue. Thus, the method converges slowly if there is an eigenvalue close in magnitude to the dominant eigenvalue.

Under the assumptions:

- A has an eigenvalue that is strictly greater in magnitude than its other eigenvalues
- The starting vector b_0 has a nonzero component in the direction of an eigenvector associated with the dominant eigenvalue.

then:

- A subsequence of (b_k) converges to an eigenvector associated with the dominant eigenvalue

Note that the sequence (b_k) does not necessarily converge. It can be shown that $b_k = e^{i\phi_k} v_1 + r_k$ where: v_1 is an eigenvector associated with the dominant eigenvalue, and $\|r_k\| \to 0$. The presence of the term $e^{i\phi_k}$ implies that (b_k) does not converge unless $e^{i\phi_k} = 1$. Under the two assumptions listed above, the sequence (μ_k) defined by $\mu_k = \frac{b_k^* A b_k}{b_k^* b_k}$ converges to the dominant eigenvalue.

Power iteration is not used very much because it can find only the dominant eigenvalue.

The algorithm is however very useful in some specific case. For instance, Google uses it to calculate the page rank of documents in their search engine. For matrices that are well-conditioned and as sparse as the web matrix, the power iteration method can be more efficient than other methods of finding the dominant eigenvector.

Some of the more advanced eigenvalue algorithms can be understood as variations of the power iteration. For instance, the inverse iteration method applies power iteration to the matrix $\hat{A}^{-1}$. Other algorithms look at the whole subspace generated by the vectors b_k. This subspace is known as the Krylov subspace. It can be computed by Arnoldi iteration or Lanczos iteration. The latter is method of choice for diagonalizing symmetric matrices with huge dimensionalities. We discuss the Lanczos algorithm in the next section.

7.7 Iterative methods: Lanczos' algorithm

The Lanczos algorithm is applied to symmetric eigenvalue problems. The basic features with a real symmetric matrix (and normally huge $n > 10^6$ and sparse) $\hat{A}$ of dimension $n \times n$ are

- The Lanczos' algorithm generates a sequence of real tridiagonal matrices T_k of dimension $k \times k$ with $k \leq n$, with the property that the extremal eigenvalues of T_k are progressively better estimates of $\hat{A}$' extremal eigenvalues.
- The method converges to the extremal eigenvalues.
- The similarity transformation is

$$\hat{T} = \hat{Q}^T \hat{A} \hat{Q},$$

with the first vector $\hat{Q}\hat{e}_1 = \hat{q}_1$.

We are going to solve iteratively

$$\hat{T} = \hat{Q}^T \hat{A} \hat{Q},$$

with the first vector $\hat{Q}\hat{e}_1 = \hat{q}_1$. We can then write out the matrix $\hat{Q}$ in terms of its column vectors

$$\hat{Q} = [\hat{q}_1 \hat{q}_2 \dots \hat{q}_n].$$

The matrix

$$\hat{T} = \hat{Q}^T \hat{A} \hat{Q},$$

can be written as

$$\hat{T} = \begin{pmatrix} \alpha_1 & \beta_1 & 0 & \dots & \dots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & 0 & \dots & 0 \\ 0 & \beta_2 & \alpha_3 & \beta_3 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & 0 \\ \dots & & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ 0 & \dots & \dots & 0 & \beta_{n-1} & \alpha_n \end{pmatrix}$$

Using the fact that $\hat{Q}\hat{Q}^T = \hat{I}$, we can rewrite

$$\hat{T} = \hat{Q}^T \hat{A}\hat{Q},$$

as

$$\hat{Q}\hat{T} = \hat{A}\hat{Q},$$

and if we equate columns (recall from the previous slide)

$$\hat{T} = \begin{pmatrix} \alpha_1 & \beta_1 & 0 & \dots & \dots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & 0 & \dots & 0 \\ 0 & \beta_2 & \alpha_3 & \beta_3 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & 0 \\ \dots & & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ 0 & \dots & \dots & 0 & \beta_{n-1} & \alpha_n \end{pmatrix}$$

we obtain

$$\hat{A}\hat{q}_k = \beta_{k-1}\hat{q}_{k-1} + \alpha_k\hat{q}_k + \beta_k\hat{q}_{k+1}.$$

We have thus

$$\hat{A}\hat{q}_k = \beta_{k-1}\hat{q}_{k-1} + \alpha_k\hat{q}_k + \beta_k\hat{q}_{k+1},$$

with $\beta_0\hat{q}_0 = 0$ for $k = 1 : n-1$. Remember that the vectors $\hat{q}_k$ are orthornormal and this implies

$$\alpha_k = \hat{q}_k^T \hat{A}\hat{q}_k,$$

and these vectors are called Lanczos vectors. We have thus

$$\hat{A}\hat{q}_k = \beta_{k-1}\hat{q}_{k-1} + \alpha_k\hat{q}_k + \beta_k\hat{q}_{k+1},$$

with $\beta_0\hat{q}_0 = 0$ for $k = 1 : n-1$ and

$$\alpha_k = \hat{q}_k^T \hat{A}\hat{q}_k.$$

If

$$\hat{r}_k = (\hat{A} - \alpha_k\hat{I})\hat{q}_k - \beta_{k-1}\hat{q}_{k-1},$$

is non-zero, then

$$\hat{q}_{k+1} = \hat{r}_k/\beta_k,$$

with $\beta_k = \pm||\hat{r}_k||_2$. These steps can then be written in terms of the following simple algorithm:

```
r_0 = q_1; beta_0=1; q_0=0; int k = 0;
while (beta_k != 0)
   q_{k+1} = r_k/beta_k
   k = k+1
   alpha_k = q_k^T A q_k
   r_k = (A-alpha_k I) q_k -beta_{k-1}q_{k-1}
   beta_k = || r_k||_2
end while
```

7.8 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

$$f''=\frac{f(x+h)-2f(x)+f(x-h)}{h^2}+O(h^2),$$

where h is our step. Next we define minimum and maximum values for the variable x, $R_{\min}$ and $R_{\max}$, respectively. With a given number of steps, N_{step}, we then define the step h as

$$h=\frac{R_{\max}-R_{\min}}{N_{\mathrm{step}}}.$$

If we now define an arbitrary value of x as

$$x_i=R_{\min}+ih \qquad i=1,2,\dots,N_{\mathrm{step}}-1$$

we can rewrite the Schrödinger equation for x_i as

$$-\frac{u(x_k+h)-2u(x_k)+u(x_k-h)}{h^2}+x_k^2u(x_k)=2Eu(x_k),$$

or in a more compact way

$$-\frac{u_{k+1}-2u_k+u_{k-1}}{h^2}+x_k^2u_k=-\frac{u_{k+1}-2u_k+u_{k-1}}{h^2}+V_ku_k=2Eu_k,$$

where $u_k = u(x_k)$, $u_{k\pm 1} = u(x_k \pm h)$ 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

$$d_k u_k + e_{k-1} u_{k-1} + e_{k+1} u_{k+1} = 2E u_k,$$

where u_k is unknown. Since we have $N_{\mathrm{step}} - 1$ values of k we can write the latter equation as a matrix eigenvalue problem

$$\begin{pmatrix} d_1 & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & d_2 & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & d_3 & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & d_{N_{\mathrm{step}}-2} & e_{N_{\mathrm{step}}-1} \\ 0 & \dots & \dots & \dots & \dots & e_{N_{\mathrm{step}}-1} & d_{N_{\mathrm{step}}-1} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{N_{\mathrm{step}}-1} \end{pmatrix} = 2E \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{N_{\mathrm{step}}-1} \end{pmatrix} \tag{7.8}$$

or if we wish to be more detailed, we can write the tridiagonal matrix as

$$\begin{pmatrix} \frac{2}{h^2}+V_1 & -\frac{1}{h^2} & 0 & 0 & \dots & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2}+V_2 & -\frac{1}{h^2} & 0 & \dots & 0 & 0 \\ 0 & -\frac{1}{h^2} & \frac{2}{h^2}+V_3 & -\frac{1}{h^2} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \frac{2}{h^2}+V_{N_{\mathrm{step}}-2} & -\frac{1}{h^2} \\ 0 & \dots & \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2}+V_{N_{\mathrm{step}}-1} \end{pmatrix} \tag{7.9}$$

This is a matrix problem with a tridiagonal matrix of dimension $N_{\mathrm{step}} - 1 \times N_{\mathrm{step}} - 1$ and will thus yield $N_{\mathrm{step}} - 1$ eigenvalues. It is important to notice that we do not set up a matrix of dimension $N_{\mathrm{step}} \times N_{\mathrm{step}}$ since we can fix the value of the wave function at $k = N_{\mathrm{step}}$. 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 9.

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.

7.8.1 *Numerical solution of the Schrödinger equation by diagonalization*

The algorithm for solving Eq. (7.8) may take the following form

- Define values for N_{step}, R_{min} and R_{max}. These values define in turn the step size h. Typical values for R_{max} and R_{min} could be 10 and -10 respectively for the lowest-lying states. The number of mesh points N_{step} could be in the range 100 to some thousands. You can check the stability of the results as functions of $N_{\mathrm{step}}-1$ and R_{max} and R_{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_{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 $N_{\mathrm{step}}-1$, 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 $n = N_{\mathrm{step}}-1$ and a matrix z of dimension $N_{\mathrm{step}}-1 \times N_{\mathrm{step}}-1$ 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 5.

7.8.2 *Program example and results for the one-dimensional harmonic oscillator*

We present here a program example which encodes the above algorithm.

http://folk.uio.no/compphys/programs/chapter07/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 $1,3,5,7,9$.

Table 7.1 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

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 $\approx N^3$. 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 3, where we dealt with interpolation and extrapolation, we discussed also 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. (7.9) 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`.

Table 7.2 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

```
! 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
      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
```

7.9 Exercises

7.1. The aim of this problem is to solve Schrödinger's equation for two electrons in a three-dimensional harmonic oscillator well with and without a repulsive Coulomb interaction. Your task is to solve this equation by reformulating it in a discretized form as an eigenvalue equation to be solved with Jacobi's method. To achieve this you will have to write your own code which implements Jacobi's method.

Electrons confined in small areas in semiconductors, so-called quantum dots, form a hot research area in modern solid-state physics, with applications spanning from such diverse fields as quantum nano-medicine to the contruction of quantum gates.

Here we will assume that these electrons move in a three-dimensional harmonic oscillator potential (they are confined by for example quadrupole fields) and repel each other via the static Colulomb interaction. We assume spherical symmetry.

We are first interested in the solution of the radial part of Schrödinger's equation for one electron. This equation reads

$$-\frac{\hbar^2}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr}-\frac{l(l+1)}{r^2}\right)R(r)+V(r)R(r)=ER(r).$$

In our case $V(r)$ is the harmonic oscillator potential $(1/2)kr^2$ with $k=m\omega^2$ and E is the energy of the harmonic oscillator in three dimensions. The oscillator frequency is ω and the energies are

$$E_{nl}=\hbar\omega\left(2n+l+\frac{3}{2}\right),$$

with $n=0,1,2,\dots$ and $l=0,1,2,\dots$.

Since we have made a transformation to spherical coordinates it means that $r\in[0,\infty)$. The quantum number l is the orbital momentum of the electron. Then we substitute $R(r)=(1/r)u(r)$ and obtain

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r)+\left(V(r)+\frac{l(l+1)}{r^2}\frac{\hbar^2}{2m}\right)u(r)=Eu(r).$$

The boundary conditions are $u(0)=0$ and $u(\infty)=0$.

We introduce a dimensionless variable $\rho=(1/\alpha)r$ where α is a constant with dimension length and get

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(\rho)+\left(V(\rho)+\frac{l(l+1)}{\rho^2}\frac{\hbar^2}{2m\alpha^2}\right)u(\rho)=Eu(\rho).$$

We will set in this project $l=0$. Inserting $V(\rho)=(1/2)k\alpha^2\rho^2$ we end up with

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(\rho)+\frac{k}{2}\alpha^2\rho^2u(\rho)=Eu(\rho).$$

We multiply thereafter with $2m\alpha^2/\hbar^2$ on both sides and obtain

$$-\frac{d^2}{d\rho^2}u(\rho)+\frac{mk}{\hbar^2}\alpha^4\rho^2u(\rho)=\frac{2m\alpha^2}{\hbar^2}Eu(\rho).$$

The constant α can now be fixed so that

$$\frac{mk}{\hbar^2}\alpha^4=1,$$

or

$$\alpha=\left(\frac{\hbar^2}{mk}\right)^{1/4}.$$

Defining

$$\lambda=\frac{2m\alpha^2}{\hbar^2}E,$$

we can rewrite Schrödinger's equation as

$$-\frac{d^2}{d\rho^2}u(\rho)+\rho^2u(\rho)=\lambda u(\rho).$$

This is the first equation to solve numerically. In three dimensions the eigenvalues for $l=0$ are $\lambda_0=3,\lambda_1=7,\lambda_2=11,\dots.$

We use the by now standard expression for the second derivative of a function u

$$u''=\frac{u(\rho+h)-2u(\rho)+u(\rho-h)}{h^2}+O(h^2),$$

where h is our step. Next we define minimum and maximum values for the variable ρ, $\rho_{\mathrm{min}}=0$ and ρ_{max}, respectively. You need to check your results for the energies against different values ρ_{max}, since we cannot set $\rho_{\mathrm{max}}=\infty$.

With a given number of steps, n_{step}, we then define the step h as

$$h=\frac{\rho_{\mathrm{max}}-\rho_{\mathrm{min}}}{n_{\mathrm{step}}}.$$

Define an arbitrary value of ρ as

$$\rho_i=\rho_{\mathrm{min}}+ih \qquad i=0,1,2,\dots,n_{\mathrm{step}}$$

we can rewrite the Schrödinger equation for ρ_i as

$$-\frac{u(\rho_i+h)-2u(\rho_i)+u(\rho_i-h)}{h^2}+\rho_i^2u(\rho_i)=\lambda u(\rho_i),$$

or in a more compact way

$$-\frac{u_{i+1}-2u_i+u_{i-1}}{h^2}+\rho_i^2u_i=-\frac{u_{i+1}-2u_i+u_{i-1}}{h^2}+V_iu_i=\lambda u_i,$$

where $V_i=\rho_i^2$ is the harmonic oscillator potential. Define first the diagonal matrix element

$$d_i=\frac{2}{h^2}+V_i,$$

and the non-diagonal matrix element

$$e_i = -\frac{1}{h^2}.$$

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

$$d_i u_i + e_{i-1} u_{i-1} + e_{i+1} u_{i+1} = \lambda u_i,$$

where u_i is unknown. We can write the latter equation as a matrix eigenvalue problem

$$\left(\begin{array}{ccccccc} d_1 & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & d_2 & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & d_3 & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & d_{n_{\text{step}}-2} & e_{n_{\text{step}}-1} \\ 0 & \dots & \dots & \dots & \dots & e_{n_{\text{step}}-1} & d_{n_{\text{step}}} \end{array}\right) \left(\begin{array}{c} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{n_{\text{step}}-1} \end{array}\right) = \lambda \left(\begin{array}{c} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{n_{\text{step}}-1} \end{array}\right) \tag{7.10}$$

or if we wish to be more detailed, we can write the tridiagonal matrix as

$$\left(\begin{array}{ccccccc} \frac{2}{h^2}+V_1 & -\frac{1}{h^2} & 0 & 0 & \dots & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2}+V_2 & -\frac{1}{h^2} & 0 & \dots & 0 & 0 \\ 0 & -\frac{1}{h^2} & \frac{2}{h^2}+V_3 & -\frac{1}{h^2} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \frac{2}{h^2}+V_{n_{\text{step}}-2} & -\frac{1}{h^2} \\ 0 & \dots & \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2}+V_{n_{\text{step}}-1} \end{array}\right) \tag{7.11}$$

Recall that the solutions are known via the boundary conditions at $i = n_{\text{step}}$ and at the other end point, that is for ρ_0. The solution is zero in both cases.

a) Your task here is to write a function which implements Jacobi's rotation algorithm in order to solve Eq. (7.10).
We Define the quantities $\tan\theta = t = s/c$, with $s = \sin\theta$ and $c = \cos\theta$ and

$$\cot 2\theta = \tau = \frac{a_{ll} - a_{kk}}{2a_{kl}}.$$

We can then define the angle θ so that the non-diagonal matrix elements of the transformed matrix a_{kl} become non-zero and we obtain the quadratic equation (using $\cot 2\theta = 1/2(\cot\theta - \tan\theta)$

$$t^2 + 2\tau t - 1 = 0,$$

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 $s = tc$. Explain why we should choose t to be the smaller of the roots. Show that these choice ensures that $|\theta| \leq \pi/4$) and has the effect of minimizing the difference between the matrices $\mathbf{B}$ and $\mathbf{A}$ since

$$||\mathbf{B}-\mathbf{A}||_F^2 = 4(1-c)\sum_{i=1,i\neq k,l}^{n}(a_{ik}^2 + a_{il}^2) + \frac{2a_{kl}^2}{c^2}.$$

b) How many points n_{step} do you need in order to get the lowest three eigenvalues with four leading digits? Remember to check the eigenvalues for the dependency on the choice of $\rho_{\max}$.
How many similarity transformations are needed before you reach a result where all non-diagonal matrix elements are essentially zero? Try to estimate the number of transformations and extract a behavior as function of the dimensionality of the matrix.
You can check your results against the code based on Householder's algorithm, *tqli* in the file lib.cpp.
Comment your results (here you could for example compute the time needed for both algorithms for a given dimensionality of the matrix).

c) We will now study two electrons in a harmonic oscillator well which also interact via a repulsive Coulomb interaction. Let us start with the single-electron equation written as

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r)+\frac{1}{2}kr^2u(r)=E^{(1)}u(r),$$

where $E^{(1)}$ stands for the energy with one electron only. For two electrons with no repulsive Coulomb interaction, we have the following Schrödinger equation

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dr_1^2}-\frac{\hbar^2}{2m}\frac{d^2}{dr_2^2}+\frac{1}{2}kr_1^2+\frac{1}{2}kr_2^2\right)u(r_1,r_2)=E^{(2)}u(r_1,r_2).$$

Note that we deal with a two-electron wave function $u(r_1,r_2)$ and two-electron energy $E^{(2)}$. With no interaction this can be written out as the product of two single-electron wave functions, that is we have a solution on closed form.
We introduce the relative coordinate $\mathbf{r}=\mathbf{r}_1-\mathbf{r}_2$ and the center-of-mass coordinate $\mathbf{R}=1/2(\mathbf{r}_1+\mathbf{r}_2)$. With these new coordinates, the radial Schrödinger equation reads

$$\left(-\frac{\hbar^2}{m}\frac{d^2}{dr^2}-\frac{\hbar^2}{4m}\frac{d^2}{dR^2}+\frac{1}{4}kr^2+kR^2\right)u(r,R)=E^{(2)}u(r,R).$$

The equations for r and R can be separated via the ansatz for the wave function $u(r,R)=\psi(r)\phi(R)$ and the energy is given by the sum of the relative energy E_r and the center-of-mass energy E_R, that is

$$E^{(2)}=E_r+E_R.$$

We add then the repulsive Coulomb interaction between two electrons, namely a term

$$V(r_1,r_2)=\frac{\beta e^2}{|\mathbf{r}_1-\mathbf{r}_2|}=\frac{\beta e^2}{r},$$

with $\beta e^2=1.44$ eVnm.
Adding this term, the r-dependent Schrödinger equation becomes

$$\left(-\frac{\hbar^2}{m}\frac{d^2}{dr^2}+\frac{1}{4}kr^2+\frac{\beta e^2}{r}\right)\psi(r)=E_r\psi(r).$$

This equation is similar to the one we had previously in (a) and we introduce again a dimensionless variable $\rho=r/\alpha$. Repeating the same steps as in (a), we arrive at

$$-\frac{d^2}{d\rho^2}\psi(\rho)+\frac{mk}{\hbar^2}\alpha^4\rho^2\psi(\rho)+\frac{m\alpha\beta e^2}{\rho\hbar^2}\psi(\rho)=\frac{m\alpha^2}{\hbar^2}E_r\psi(\rho).$$

We want to manipulate this equation further to make it as similar to that in (a) as possible. We define $k_r = 1/4k$ The constant α is then again fixed so that

$$\frac{mk_r}{\hbar^2}\alpha^4 = 1,$$

or

$$\alpha = \left(\frac{\hbar^2}{mk_r}\right)^{1/4}.$$

Defining

$$\lambda = \frac{m\alpha^2}{\hbar^2}E,$$

we can rewrite Schrödinger's equation as

$$-\frac{d^2}{d\rho^2}\psi(\rho) + \rho^2\psi(\rho) + \frac{\gamma}{\rho} = \lambda\psi(\rho),$$

with

$$\gamma = \frac{m\alpha\beta e^2}{\hbar^2}.$$

We treat γ as a parameter which reflects the strength of the oscillator potential.
Here we will study the cases $\gamma = 0$, $\gamma = 0.5$, $\gamma = 1$, $\gamma = 2$ and $\gamma = 4$. for the ground state only, that is the lowest-lying state.
For $\gamma = 0$ you should get a result which corresponds to the relative energy of a non-interacting system. The way we have written the equations means you get the same as in (a) for $\gamma = 0$. Make sure your results are stable as functions of ρ_{max} and the number of steps.
We are only interested in the ground state with $l = 0$. We omit the center-of-mass energy.
You can reuse the code you wrote for (a), but you need to change the potential from ρ^2 to $\rho^2 + \gamma/\rho$.
Comment the results for the lowest state (ground state) as function of varying strengths of γ.
For specific oscillator frequencies, the above equation has analytic answers, see the article by M. Taut, Phys. Rev. A 48, 3561 - 3566 (1993). The article can be retrieved from the following web address `http://prola.aps.org/abstract/PRA/v48/i5/p3561_1`.

d) In this exercise we want to plot the wave function for two electrons as functions of the relative coordinate r and different values of γ. For $\gamma = 0$ your wave function should correspond to that of a harmonic oscillator. Varying γ, the shape of the wave function will change.
We are only interested in the wave function for the ground state with $l = 0$ and omit again the center-of-mass motion.
You can choose between two approaches; the first is to use the existing *tqli* function. Here the eigenvectors are obtained from the matrix $z[i][j]$, where the index j refers to eigenvalue j. The index i points to the value of the wave function in position ρ_j. That is, $u^{(\lambda_j)}(\rho_i) = z[i][j]$. The eigenvectors are normalized. Plot then the normalized wave functions for different values of γ and comment the results.
The other alternative is to add a piece to your Jacobi routine which also returns the eigenvectors. This is the more difficult part. You will need to normalize the eigenvectors.

Part III
Differential Equations

Here we discuss both ordinary differential equations and partial differential equations, with a focus on finite difference schemes. We specialize also in a dedicated chapter on numerical solutions of the time-dependent Schrödinger equation.

Chapter 8
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 [40]*

Abstract This chapter aims at giving an overview on some of the most used methods to solve ordinary differential equations. Several examples of applications to physical systems are discussed, from the classical pendulum to the physics of Neutron stars.

8.1 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 [41], Butcher [42] and Hairer, Nørsett and Wanner [43].

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 10.

[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.

8.2 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
$$\frac{dy}{dt} = f(t,y).$$
This equation is of first order and f is an arbitrary function. A second-order equation goes typically like
$$\frac{d^2y}{dt^2} = f(t,\frac{dy}{dt},y).$$
A well-known second-order equation is Newton's second law
$$m\frac{d^2x}{dt^2} = -kx, \tag{8.1}$$
where k is the force constant. ODE depend only on one variable, whereas
- partial differential equations like the time-dependent Schrödinger equation
$$i\hbar\frac{\partial\psi(\mathbf{x},t)}{\partial t} = \frac{\hbar^2}{2m}\left(\frac{\partial^2\psi(\mathbf{r},t)}{\partial x^2}+\frac{\partial^2\psi(\mathbf{r},t)}{\partial y^2}+\frac{\partial^2\psi(\mathbf{r},t)}{\partial z^2}\right)+V(\mathbf{x})\psi(\mathbf{x},t),$$
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.,
$$\frac{dy}{dt} = g^3(t)y(t),$$
is an example of a linear equation, while
$$\frac{dy}{dt} = g^3(t)y(t) - g(t)y^2(t),$$
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. (8.1). If we define the position $x(t) = y^{(1)}(t)$ and the velocity $v(t) = y^{(2)}(t)$ as its derivative

$$\frac{dy^{(1)}(t)}{dt} = \frac{dx(t)}{dt} = y^{(2)}(t),$$

we can rewrite Newton's second law as two coupled first-order differential equations

$$m\frac{dy^{(2)}(t)}{dt} = -kx(t) = -ky^{(1)}(t), \tag{8.2}$$

and

$$\frac{dy^{(1)}(t)}{dt} = y^{(2)}(t). \tag{8.3}$$

8.3 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

$$y_0 = y(t = t_0).$$

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

$$h = \frac{b-a}{N}.$$

With this step and the derivative of y we can construct the next value of the function y at

$$y_1 = y(t_1 = t_0 + h),$$

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

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h\Delta(t_i, y_i(t_i)) + O(h^{p+1}),$$

where $O(h^{p+1})$ represents the truncation error. To determine Δ, we Taylor expand our function y

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h\left(y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}\right) + O(h^{p+1}), \tag{8.4}$$

where we will associate the derivatives in the parenthesis with

$$\Delta(t_i, y_i(t_i)) = (y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}). \tag{8.5}$$

We define

$$y'(t_i) = f(t_i, y_i)$$

and if we truncate Δ at the first derivative, we have

$$y_{i+1} = y(t_i) + hf(t_i, y_i) + O(h^2), \tag{8.6}$$

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 how a solution may look like. As an example, consider Newton's equation rewritten in Eqs. (8.2) and (8.3). We define $y_0 = y^{(1)}(t=0)$ an $v_0 = y^{(2)}(t=0)$. The first steps in Newton's equations are then

$$y_1^{(1)} = y_0 + hv_0 + O(h^2)$$

and

$$y_1^{(2)} = v_0 - hy_0k/m + O(h^2).$$

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)

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1}^{(2)} + O(h^2)$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2).$$

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

$$\Delta(t_i,y_i(t_i)) = f(t_i) + \frac{h}{2}\frac{df(t_i,y_i)}{dt} + O(h^3).$$

The second derivative can be rewritten as

$$y'' = f' = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f$$

and we can rewrite Eq. (8.4) as

$$y_{i+1} = y(t=t_i+h) = y(t_i) + hf(t_i) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y} f\right) + O(h^3),$$

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

$$y_{i+1} = y(t=t_i+h) = y(t_i) + h(f(t_i,y_i) + \dots f^{(p-1)}(t_i,y_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}).$$

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.

8.3.1 Improvements of 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

$$y_{n+1}^{(1)} = y_n^{(1)} + \frac{h}{2}\left(y_{n+1}^{(2)} + y_n^{(2)}\right) + O(h^2)$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2),$$

yielding

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_n^{(2)} + \frac{h^2}{2}a_n + O(h^3)$$

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

$$y_{n+1/2}^{(2)} = y_{n-1/2}^{(2)} + ha_n + O(h^2),$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2).$$

Note that this method needs the calculation of $y_{1/2}^{(2)}$. This is done using for example Euler's method

$$y_{1/2}^{(2)} = y_0^{(2)} + \frac{h}{2}a_0 + O(h^2).$$

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

$$y^{(2)}_{n+1} = y^{(2)}_{n} + ha_{n+1/2} + O(h^2), \tag{8.7}$$

and

$$y^{(1)}_{n+1} = y^{(1)}_{n} + hy^{(2)}_{n+1/2} + O(h^2). \tag{8.8}$$

8.3.2 Verlet and Leapfrog algorithms

Another set of popular algorithms, which are both numerically stable and easy to implement are the Verlet and Leapfrog algorithms. These algorithms are much used in so-called Molecular Dynamics applications, see for example Refs. [44, 45]. Consider again a second-order differential equation like Newton's second law, whose one-dimensional version reads

$$m\frac{d^2x}{dt^2} = F(x,t),$$

which we rewrite in terms of two coupled differential equations

$$\frac{dx}{dt} = v(x,t) \qquad \text{and} \qquad \frac{dv}{dt} = F(x,t)/m = a(x,t).$$

If we now perform a Taylor expansion

$$x(t+h) = x(t) + hx^{(1)}(t) + \frac{h^2}{2}x^{(2)}(t) + O(h^3).$$

In our case the second derivative is know via Newton's second law, namely $x^{(2)}(t) = a(x,t)$. If we add to the above equation the corresponding Taylor expansion for $x(t-h)$, we obtain, using the discretized expressions

$$x(t_i \pm h) = x_{i\pm 1} \qquad \text{and} \qquad x_i = x(t_i),$$

$$x_{i+1} = 2x_i - x_{i-1} + h^2 x^{(2)}_i + O(h^4).$$

We note that the truncation error goes like $O(h^4)$ since all the odd terms cancel when we add the two Taylor expansions. We see also that the velocity is not directly included in the equation since the function $x^{(2)} = a(x,t)$ is supposed to be known. If we need the velocity however, we can compute it using the well-known formula

$$x^{(1)}_i = \frac{x_{i+1} - x_{i-1}}{2h} + O(h^2).$$

We note that the velocity has a truncation error which goes like $O(h^2)$. In for example so-called Molecular dynamics calculations, since the acceleration is normally known via Newton's second law, there is seldomly a need for computing the velocity. The above sets of equations for the position $x(t)$ and the velocity defines the Verlet formula. The Leapfrog algorithm is also easily derived.

We can rewrite the above Taylor expansion for $x(t+h)$ as

$$x(t+h) = x(t) + h\left(x^{(1)}(t) + \frac{h}{2}x^{(2)}(t)\right) + O(h^3).$$

Noting that

$$x^{(1)}(t+h/2) = \left(x^{(1)}(t) + \frac{h}{2}x^{(2)}(t)\right) + O(h^2),$$

we obtain

$$x(t+h) = x(t) + h + x^{(1)}(t+h/2) + O(h^3),$$

which needs to be combined with

$$x^{(1)}(t+h/2) = x^{(1)}(t-h/2) + hx^{(2)}(t) + O(h^2).$$

Again, there is a lower truncation error in h for the velocity. Furthermore, the positions and the velocities are evaluated at different time steps. If one needs $x^{(1)}(t)$, this can be computed using

$$x^{(1)}(t) = \left(x^{(1)}(t \mp h/2) \pm \frac{h}{2}x^{(2)}(t)\right) + O(h^2).$$

The initial conditions can be handled in similar ways and the inaccuracy which arises between $x^{(1)}(0)$ and $x^{(1)}(h/2)$ is normally ignored. Summarizing, the popular Leapfrog algorithm implies the evaluation of position and velocity at different time steps. The final algorithm is given by the following steps

$$x^{(1)}(t+h/2) = x^{(1)}(t-h/2) + hx^{(2)}(t) + O(h^2),$$

which is used in

$$x(t+h) = x(t) + h + x^{(1)}(t+h/2) + O(h^3),$$

and finally

$$x^{(1)}(t+h) = x^{(1)}(t+h/2) + \frac{h}{2}x^{(2)}(t+h) + O(h^2),$$

8.3.3 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 [24] 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.

8.4 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

$$\frac{dy}{dt} = f(t, y),$$

and

$$y(t) = \int f(t, y) dt,$$

and

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y) dt.$$

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

$$\int_{t_i}^{t_{i+1}} f(t, y) dt \approx h f(t_{i+1/2}, y_{i+1/2}) + O(h^3).$$

This means in turn that we have

$$y_{i+1} = y_i + h f(t_{i+1/2}, y_{i+1/2}) + O(h^3).$$

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

$$y_{(i+1/2)} = y_i + \frac{h}{2}\frac{dy}{dt} = y(t_i) + \frac{h}{2}f(t_i, y_i).$$

This means that we can define the following algorithm for the second-order Runge-Kutta method, RK2.

$$k_1 = hf(t_i, y_i),$$

$$k_2 = hf(t_{i+1/2}, y_i + k_1/2),$$

with the final value

$$y_{i+1} \approx y_i + k_2 + O(h^3).$$

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 Simpson's 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

$$\int_{t_i}^{t_{i+1}} f(t, y)dt \approx \frac{h}{6}\left[f(t_i, y_i) + 4f(t_{i+1/2}, y_{i+1/2}) + f(t_{i+1}, y_{i+1})\right] + O(h^5).$$

This means in turn that we have

$$y_{i+1} = y_i + \frac{h}{6}\left[f(t_i, y_i) + 4f(t_{i+1/2}, y_{i+1/2}) + f(t_{i+1}, y_{i+1})\right] + O(h^5).$$

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
$$k_1 = hf(t_i, y_i), \tag{8.9}$$
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
$$k_2 = hf(t_i + h/2, y_i + k_1/2). \tag{8.10}$$
3. The improved slope at the midpoint is used to further improve the slope of $y_{i+1/2}$ by computing

$$k_3 = hf(t_i + h/2, y_i + k_2/2). \tag{8.11}$$

4. With the latter slope we can in turn predict the value of y_{i+1} via the computation of

$$k_4 = hf(t_i + h, y_i + k_3). \tag{8.12}$$

5. The final algorithm becomes then

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \tag{8.13}$$

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. 8.1 gives a geometrical interpretation of the fourth-order Runge-Kutta method.

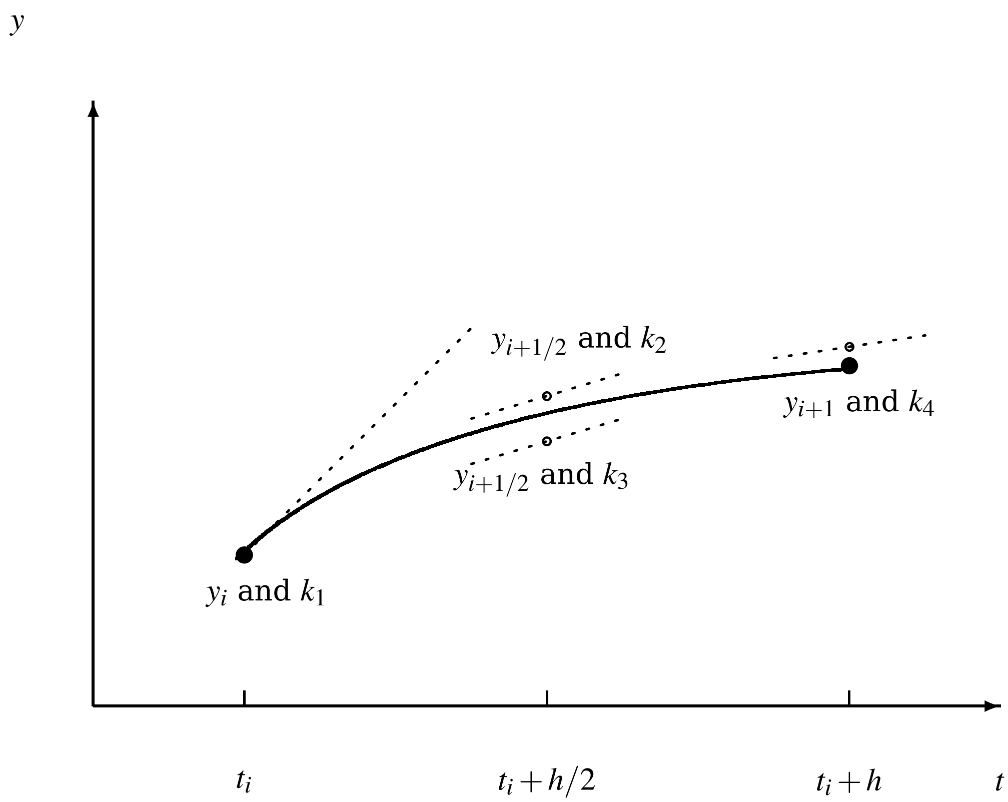


Fig. 8.1 Geometrical interpretation of the fourth-order Runge-Kutta method. The derivative is evaluated at four points, once at the intial point, twice at the trial midpoint and once at the trial endpoint. These four derivatives constitute one Runge-Kutta step resulting in the final value for $y_{i+1} = y_i + 1/6(k_1 + 2k_2 + 2k_3 + k_4)$.

8.5 Adaptive Runge-Kutta and multistep methods

In case the function to integrate varies slowly or fast in different integration domains, adaptive methods are normally used. One strategy is always to decrease the step size. As we have seen earlier, this leads to more computations and may eventually even lead to the loss of numerical precision. An alternative is to use higher-order Runge-Kutta methods for example. However, this leads again to more cycles, furthermore, there is no guarantee that higher-order leads to an improved error, see for example the discussions in Ref. [42]

Assume the exact result is $\tilde{y}$ and that we are using a Runge-Kutta method of order M. Suppose we run two calculations, one with a step length h (which we will label y_1) and one with step length $h/2$ (labelled y_2). The exact solution in terms of y_1 is

$$\tilde{y} = y_1 + Ch^{M+1} + O(h^{M+2}),$$

where C is some constant and

$$\tilde{y} = y_2 + 2C(h/2)^{M+1} + O(h^{M+2}).$$

Note that we need to perform two calculations in the last equation, one for each interval defined by $h/2$.calculate two halves in the last equation. The difference between the two solutions is then

$$|y_1 - y_2| = Ch^{M+1}(1 - \frac{1}{2^M}),$$

from which we can define the constant C as

$$C = \frac{|y_1 - y_2|}{(1-2^{-M})h^{M+1}}. \tag{8.14}$$

We rewrite then the exact solution in terms of a quantity ε

$$\tilde{y} = y_2 + \varepsilon + O((h)^{M+2}),$$

with

$$\varepsilon = \frac{|y_1 - y_2|}{2^M - 1}.$$

If we employ our fourth-order Runge-Kutta scheme, we have

$$\tilde{y} = y_2 + \varepsilon + O(h^6),$$

with

$$\varepsilon = \frac{|y_1 - y_2|}{15}.$$

The estimate is one order higher than the original Runge-Kutta method to fourth order. But this method is normally rather inefficient since it requires a lot of computations. We solve typically the equation three times at each time step. However, we can compare the estimate ε with some by us given accuracy ξ say for example $\xi = 10^{-8}$. We can then ask the following question: what is, with a given y_j and t_j, the largest possible step size $\tilde{h}$ that leads to an error below ξ? We want

$$C\tilde{h}^{M+1} \le \xi,$$

which leads to, using Eq. (8.14),

$$\left(\frac{\tilde{h}}{h}\right)^{M+1} \frac{|y_1 - y_2|}{(1-2^{-M})} \le \xi,$$

meaning that we can define this optimal step length as

$$\tilde{h} = h\left(\frac{\xi}{\varepsilon}\right)^{1/(M+1)}.$$

Using this equation, we can design the following algorithm:

- If the two answers are close, use the current value for the step length h.
- If $\varepsilon > \xi$ we need to decrease the step size in the next time step.
- If $\varepsilon < \xi$ we need to increase the step size in the next time step.

At each step, two different approximations for the solution are made and compared. If the two answers are in close agreement, the approximation is accepted. If the two answers do not agree to a specified accuracy, the step size is reduced. If the answers agree to more significant digits than required, the step size is increased. Even though this algorithm is rather simple to implement, it requires unnecessarily many computations.

It is possible to reduce the number of operations by combining Runge-Kutta algorithms of different orders. A much used algorithm is the so-called Runge-Kutta-Fehlberg algorithm which uses a combination of fourth and fifth order Runge-Kutta methods, normally abbreviated to RKF45. Without going into much details, the philosophy of such methods consists in evaluating the function f such that the function values can be used for both the fourth order and the fifth order method, avoiding thereby additional computations. The RKF45 method requires at each step the computations of the following six values

$$k_1 = hf(t_k, y_k),$$

$$k_2 = hf(t_k + \frac{1}{4}h, y_k + \frac{1}{4}k_1),$$

$$k_3 = hf(t_k + \frac{3}{8}h, y_k + \frac{3}{32}k_1 + \frac{9}{32}k_2),$$

$$k_4 = hf(t_k + \frac{12}{13}h, y_k + \frac{1932}{2197}k_1 + \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3),$$

$$k_5 = hf(t_k + h, y_k + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 + \frac{845}{4104}k_4),$$

and

$$k_6 = hf(t_k + \frac{1}{2}h, y_k - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_2 + \frac{1859}{4104}k_4 - + \frac{11}{40}k_5).$$

Then an approximation to the solution of the ordinary differential equation is made using a Runge-Kutta method of order four:

$$y_{k+1} = y_k + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4101}k_4 - \frac{1}{5}k_5,$$

where the four function values k_1 , k_3 , k_4 , and k_5 are used. Notice that k_2 is not used here. A better value for the solution is determined using a Runge-Kutta method of order five as follows

$$z_{k+1} = y_k + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6.$$

The optimal time step αh is then determined by

$$\alpha = \left(\frac{\xi h}{2|z_{k+1} - y_{k+1}|}\right)^{1/4},$$

with ξ our defined tolerance. For more details behind the derivation of this method, see for example Ref. [42].

8.6 Physics examples

8.6.1 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. 8.2. If the spring is not compressed or stretched too far, the force on the block at a given position x is

$$F = -kx.$$

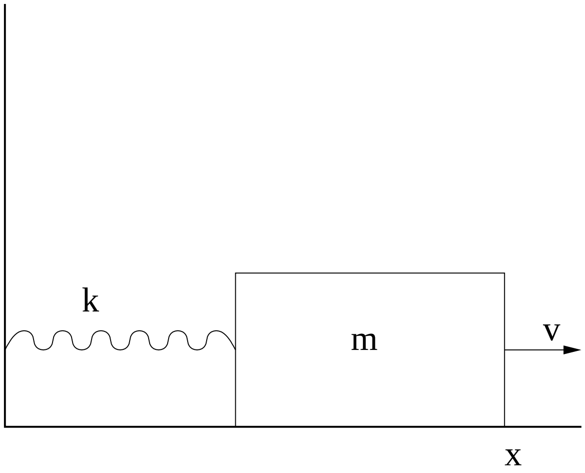


Fig. 8.2 Block tied to a wall with a spring tension acting on it.

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

$$m\frac{d^2x}{dt^2} = -kx,$$

or we could rephrase it as

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x = -\omega_0^2 x, \tag{8.15}$$

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

$$\frac{dx(t)}{dt} = v(t),$$

and

$$\frac{dv(t)}{dt} = -\omega_0^2 x(t).$$

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

$$x(t+T) = x(t),$$

with T the period defined as

$$T = \frac{2\pi}{\omega_0} = \frac{2\pi}{\sqrt{k/m}}.$$

Observe that T depends only on k/m and not on the amplitude of the solution or the constant v.

In addition to the periodicity test, the total energy has also to be conserved.

Suppose we choose the initial conditions

$$x(t=0) = 1 \text{ m} \qquad v(t=0) = 0 \text{ m/s},$$

meaning that block is at rest at $t = 0$ but with a potential energy

$$E_0 = \frac{1}{2}kx(t=0)^2 = \frac{1}{2}k.$$

The total energy at any time t has however to be conserved, meaning that our solution has to fulfill the condition

$$E_0 = \frac{1}{2}kx(t)^2 + \frac{1}{2}mv(t)^2.$$

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.

8.6.1.1 Program to solve the differential equations for a sliding block

The program which implements the above algorithm is presented here, with a corresponding

http://folk.uio.no/mhjensen/compphys/programs/chapter08/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. 8.3 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.

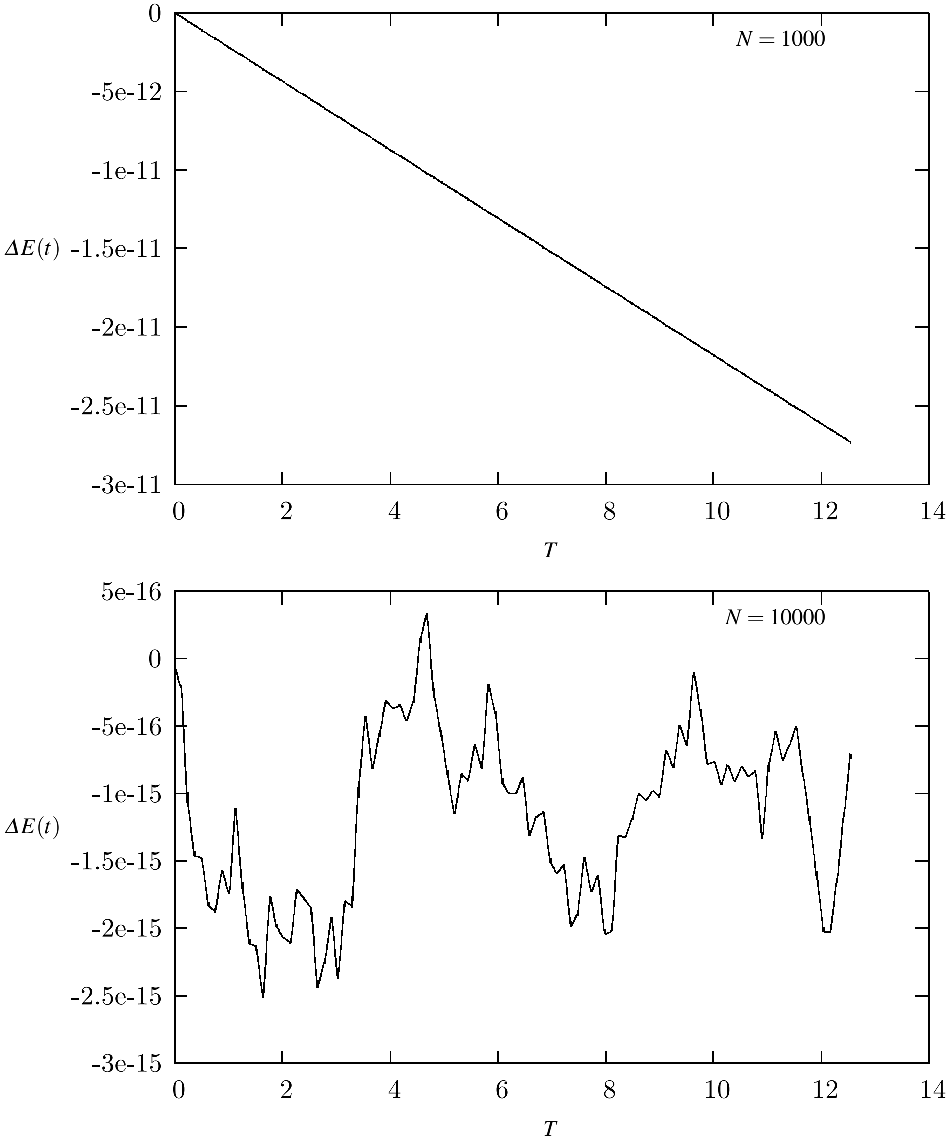


Fig. 8.3 Plot of $\Delta E(t) = E_0 - E_{\mathrm{computed}}$ for $N = 1000$ and $N = 10000$ time steps up to two periods. The initial position $x_0 = 1$ m and initial velocity $v_0 = 0$ m/s. The mass and spring tension are set to $k = m = 1$.

8.6.2 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. (8.15) to

$$\frac{d^2x}{dt^2} = -\omega_0^2 x - \nu \frac{dx}{dt}, \tag{8.16}$$

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

$$F(t) = Bcos(\omega t),$$

and we rewrite Eq. (8.16) as

$$\frac{d^2x}{dt^2} = -\omega_0^2 x - \nu \frac{dx}{dt} + F(t). \tag{8.17}$$

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 drop V, we rewrite Eq. (8.17) as

$$L\frac{d^2Q}{dt^2} + \frac{Q}{C} + R\frac{dQ}{dt} = V(t). \tag{8.18}$$

The circuit is shown in Fig. 8.4.

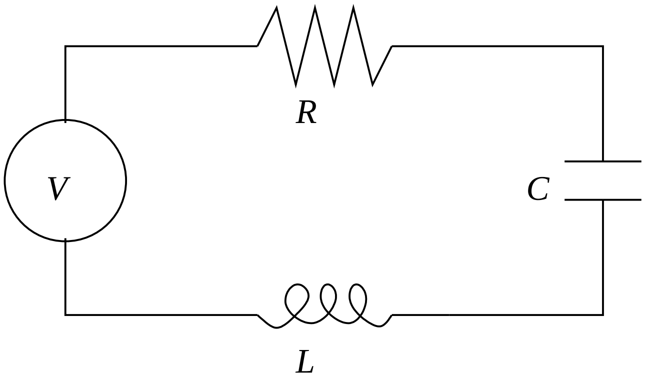


Fig. 8.4 Simple RLC circuit with a voltage source V.

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

$$V(t) = IR + LdI/dt + Q/C,$$

and using

$$I = \frac{dQ}{dt},$$

we arrive at Eq. (8.18).

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!

8.6.3 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. 8.5.

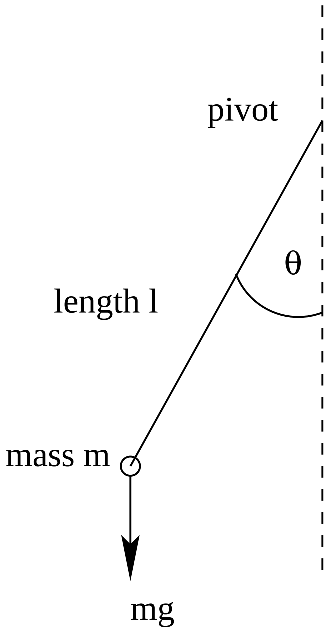


Fig. 8.5 A simple pendulum.

The angular equation of motion of the pendulum is again given by Newton's equation, but now as a nonlinear differential equation

$$ml\frac{d^2\theta}{dt^2} + mg\sin(\theta) = 0,$$

with an angular velocity and acceleration given by

$$v = l\frac{d\theta}{dt},$$

and

$$a = l\frac{d^2\theta}{dt^2}.$$

For small angles, we can use the approximation

$$\sin(\theta) \approx \theta.$$

and rewrite the above differential equation as

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\theta,$$

which is exactly of the same form as Eq. (8.15). We can thus check our solutions for small values of θ against an analytical solution. The period is now

$$T = \frac{2\pi}{\sqrt{l/g}}.$$

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

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mg\sin(\theta) = 0,$$

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

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mg\sin(\theta) = A\cos(\omega t), \tag{8.19}$$

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

$$\omega_0 = \sqrt{g/l},$$

the so-called natural frequency and the new dimensionless quantities

$$\hat{t} = \omega_0 t,$$

with the dimensionless driving frequency

$$\hat{\omega} = \frac{\omega}{\omega_0},$$

and introducing the quantity Q, called the *quality factor*,

$$Q = \frac{mg}{\omega_0 \nu},$$

and the dimensionless amplitude

$$\hat{A} = \frac{A}{mg}$$

we can rewrite Eq. (8.19) as

$$\frac{d^2\theta}{d\hat{t}^2} + \frac{1}{Q}\frac{d\theta}{d\hat{t}} + \sin(\theta) = \hat{A}\cos(\hat{\omega}\hat{t}).$$

This equation can in turn be recast in terms of two coupled first-order differential equations as follows

$$\frac{d\theta}{d\hat{t}} = \hat{v},$$

and

$$\frac{d\hat{v}}{d\hat{t}} = -\frac{\hat{v}}{Q} - \sin(\theta) + \hat{A}\cos(\hat{\omega}\hat{t}).$$

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.

8.7 Physics Project: the pendulum

8.7.1 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

$$\frac{d\theta}{d\hat{t}} = \hat{v},$$

and

$$\frac{d\hat{v}}{d\hat{t}} = -\frac{\hat{v}}{Q} - \theta + \hat{A}\cos(\hat{\omega}\hat{t}).$$

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

$$\begin{aligned} \theta(t) &= \left[\theta_0 - \frac{\hat{A}(1-\hat{\omega}^2)}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}\right] e^{-\tau/2Q} cos(\sqrt{1-\frac{1}{4Q^2}}\tau) \\ &+ \left[\hat{v}_0 + \frac{\theta_0}{2Q} - \frac{\hat{A}(1-3\hat{\omega}^2)/2Q}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}\right] e^{-\tau/2Q} sin(\sqrt{1-\frac{1}{4Q^2}}\tau) + \frac{\hat{A}(1-\hat{\omega}^2)cos(\hat{\omega}\tau)+\frac{\hat{\omega}}{Q}sin(\hat{\omega}\tau)}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}, \end{aligned}$$

and

$$\begin{aligned} \hat{v}(t) &= \left[\hat{v}_0 - \frac{\hat{A}\hat{\omega}^2/Q}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}\right] e^{-\tau/2Q} cos(\sqrt{1-\frac{1}{4Q^2}}\tau) \\ &- \left[\theta_0 + \frac{\hat{v}_0}{2Q} - \frac{\hat{A}[(1-\hat{\omega}^2)-\hat{\omega}^2/Q^2]}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}\right] e^{-\tau/2Q} sin(\sqrt{1-\frac{1}{4Q^2}}\tau) + \frac{\hat{\omega}\hat{A}[-(1-\hat{\omega}^2)sin(\hat{\omega}\tau)+\frac{\hat{\omega}}{Q}cos(\hat{\omega}\tau)]}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}, \end{aligned}$$

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

$$\theta(t) = \frac{\hat{A}(1-\hat{\omega}^2)cos(\hat{\omega}\tau) + \frac{\hat{\omega}}{Q}sin(\hat{\omega}\tau)}{(1-\hat{\omega}^2)^2 + \hat{\omega}^2/Q^2},$$

and

$$\hat{v}(t) = \frac{\hat{\omega}\hat{A}[-(1-\hat{\omega}^2)sin(\hat{\omega}\tau) + \frac{\hat{\omega}}{Q}cos(\hat{\omega}\tau)]}{(1-\hat{\omega}^2)^2 + \hat{\omega}^2/Q^2},$$

tracing the closed phase-space curve

$$\left(\frac{\theta}{\tilde{A}}\right)^2 + \left(\frac{\hat{v}}{\hat{\omega}\tilde{A}}\right)^2 = 1$$

with

$$\tilde{A} = \frac{\hat{A}}{\sqrt{(1-\hat{\omega}^2)^2 + \hat{\omega}^2/Q^2}}.$$

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. 8.6 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. (8.6), Euler-Richardson's method in Eqs. (8.7)-(8.8) 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 present results obtained with the fourth-order Runge-Kutta method.

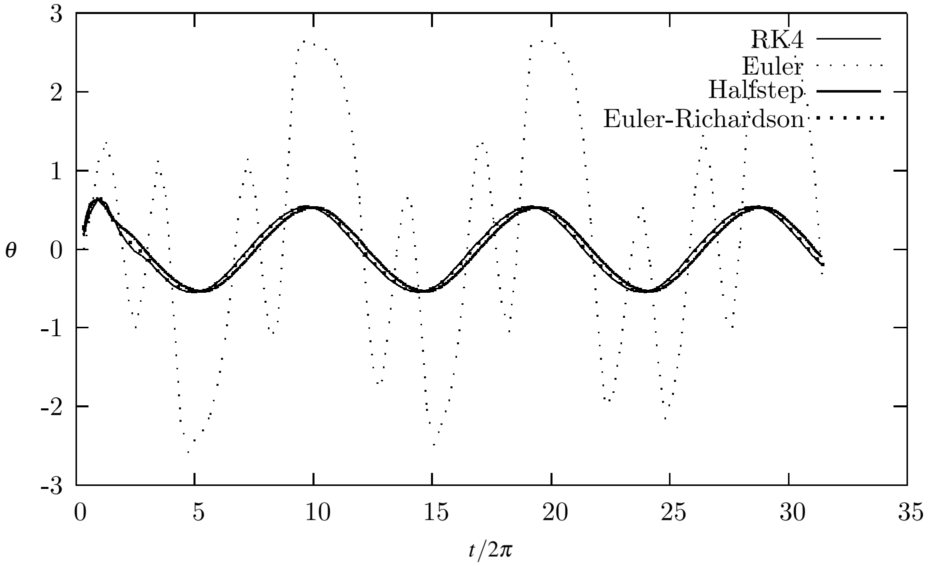


Fig. 8.6 Plot of θ as function of time τ with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 0.5$. The mass and length of the pendulum are set equal to 1. The initial velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.01$. Four different methods have been used to solve the equations, Euler's method from Eq. (8.6), the half-step method, Euler-Richardson's method in Eqs. (8.7)-(8.8) and finally the fourth-order Runge-Kutta scheme RK4. Only $N = 100$ integration points have been used for a time interval $t \in [0, 10\pi]$.

The corresponding phase space plot is shown in Fig. 8.7, for the same parameters as in Fig. 8.6. 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. 8.6. 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.

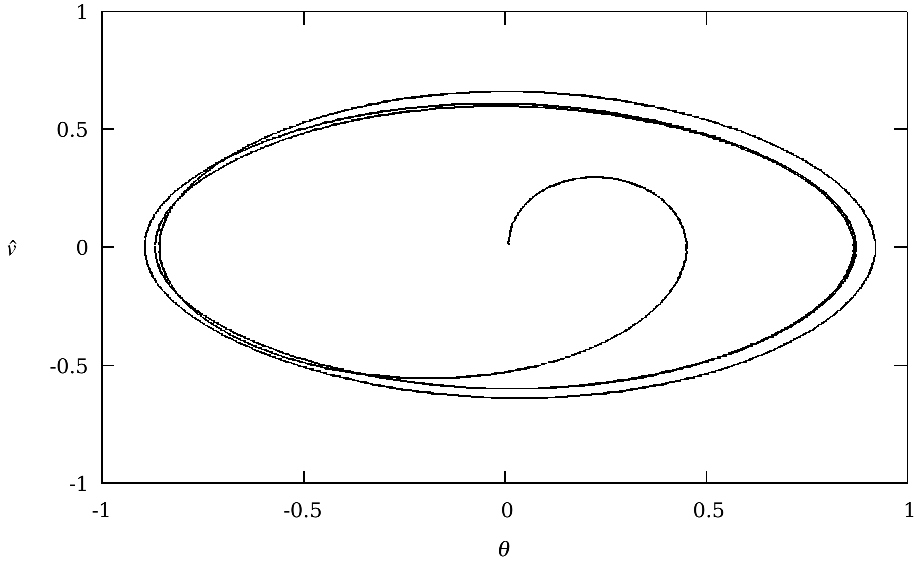


Fig. 8.7 Phase-space curve of a linear damped pendulum with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 0.5$. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.01$.

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. 8.8. The corresponding phase-space curve is shown in Fig. 8.9. 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 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. 8.10 that θ as function of time exhibits a rather different behavior from Fig. 8.8, even though the initial conditions and all other parameters except $\hat{A}$ are the same. The phase-space curve is shown in Fig. 8.11.

We will explore these topics in more detail in Exercise 8.2 below, where we extend our discussion to the phenomena of period doubling and its link to chaotic motion.

8.7.2 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

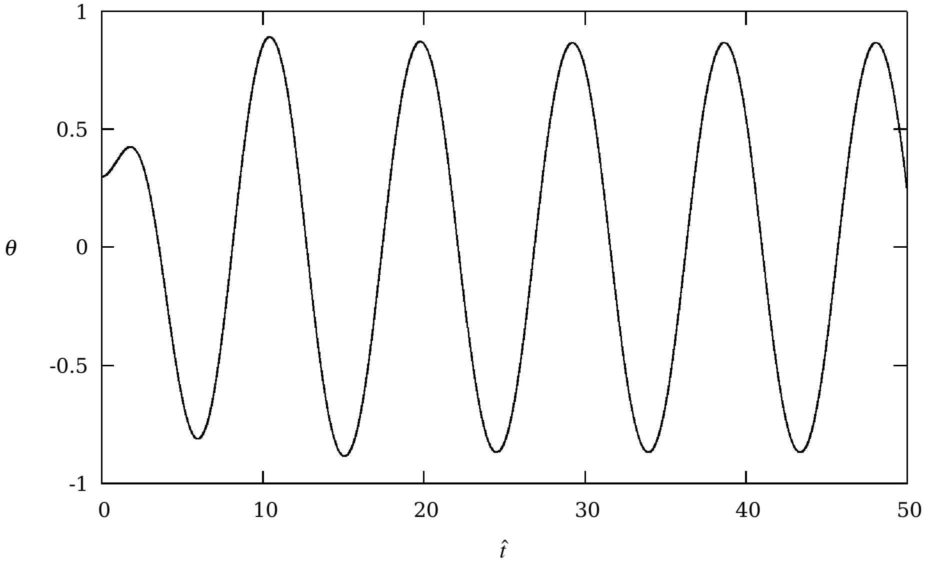


Fig. 8.8 Plot of θ as function of time τ with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 0.5$. The mass of the pendulum is set equal to 1 kg and its length to 1 m. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.3$.

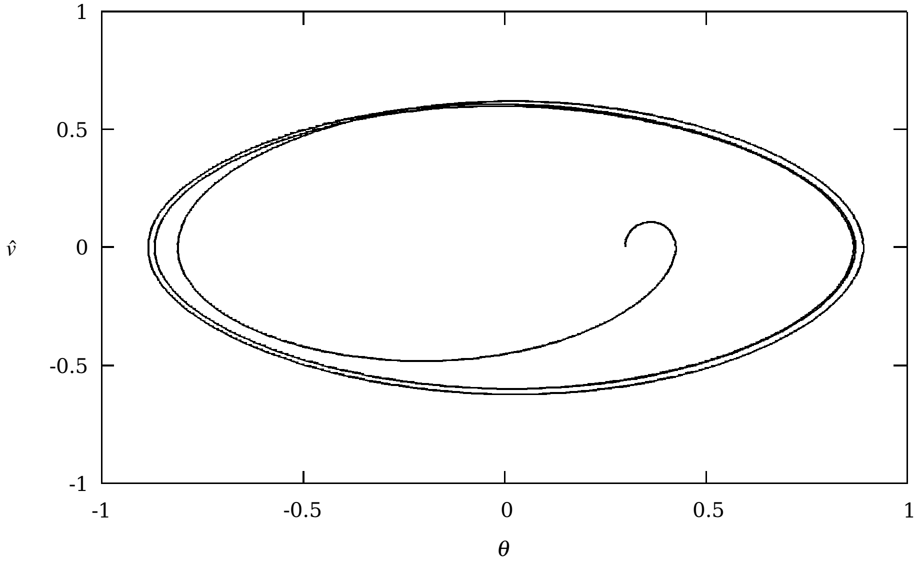


Fig. 8.9 Phase-space curve with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 0.5$. The mass of the pendulum is set equal to 1 kg and its length $l = 1$ m.. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.3$.

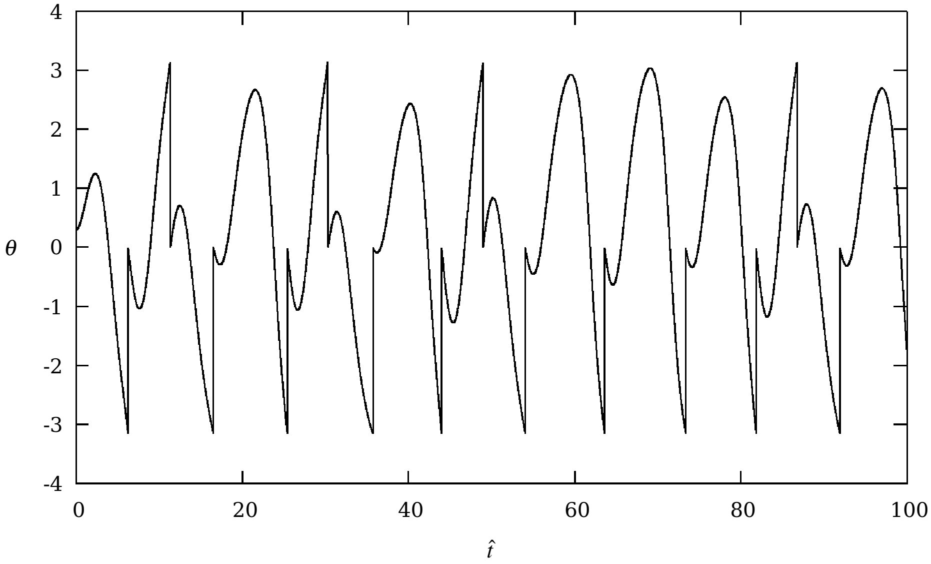


Fig. 8.10 Plot of θ as function of time τ with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 1.35$. The mass of the pendulum is set equal to 1 kg and its length to 1 m. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.3$. Every time θ passes the value $\pm\pi$ we reset its value to swing between $\theta \in [-\pi, pi]$. This gives the vertical jumps in amplitude.

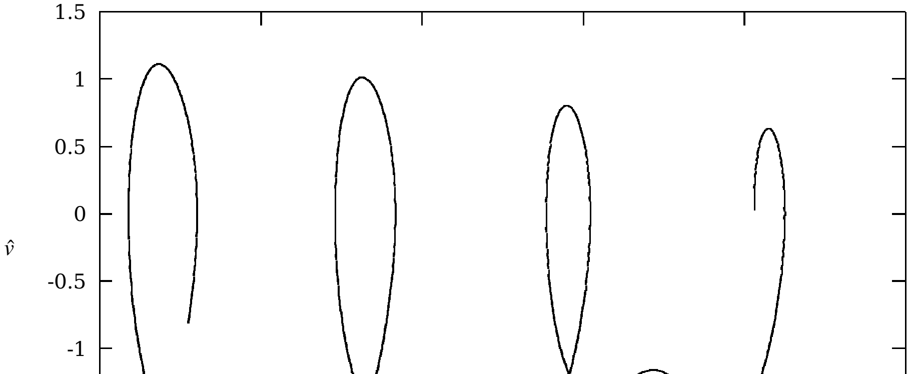


Fig. 8.11 Phase-space curve after 10 periods with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 1.35$. The mass of the pendulum is set equal to 1 kg and its length $l = 1$ m. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.3$.

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 a class where we define various methods for solving ordinary and coupled first order differential equations. This is done 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/chapter08/program2.cpp.

http://folk.uio.no/mhjensen/compphys/programs/chapter08/cpp/program2.cpp

```
  #include <stdio.h> 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
```

8.8 Exercises

8.1. 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 $0.033 \text{ s} \leq P \leq 4.0 \text{ s}$. 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. [46–48] 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 $< \rho < 10^{15}$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

$$F_{Grav} = -\frac{Gm}{r^2}\rho/c^2,$$

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

$$m(r) = \frac{4\pi}{c^2}\int_0^r \rho(r')r'^2 dr'$$

which gives rise to a differential equation for mass and density

$$\frac{dm}{dr} = 4\pi r^2\rho(r)/c^2.$$

When the star is in equilibrium we have

$$\frac{dP}{dr} = -\frac{Gm(r)}{r^2}\rho(r)/c^2.$$

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.

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

$\hat{r} = r/R_0$, 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

$$\frac{dM_0\hat{m}}{dR_0\hat{r}} = 4\pi R_0^2\hat{r}^2\rho_s\hat{\rho},$$

yielding

$$\frac{d\hat{m}}{d\hat{r}} = 4\pi R_0^3\hat{r}^2\rho_s\hat{\rho}/M_0.$$

If these equations should be dimensionless we must demand that

$$4\pi R_0^3\rho_s/M_0 = 1.$$

Correspondingly, we have for the pressure equation

$$\frac{d\rho_s\hat{P}}{dR_0\hat{r}} = -GM_0\frac{\hat{m}\rho_s\hat{\rho}}{R_0^2\hat{r}^2}$$

and since this equation should also be dimensionless, we will have

$$GM_0/R_0 = 1.$$

This means that the constants R_0 and M_0 which will render the equations dimensionless are given by

$$R_0 = \frac{1}{\sqrt{\rho_s G 4\pi}},$$

and

$$M_0 = \frac{4\pi\rho_s}{(\sqrt{\rho_s G 4\pi})^3}.$$

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

$$\frac{d\hat{P}}{d\hat{r}} = -\frac{\hat{m}\hat{\rho}}{\hat{r}^2}, \qquad \frac{d\hat{m}}{d\hat{r}} = \hat{r}^2\hat{\rho}.$$

In the solution of our problem, we will assume that the mass-energy density is given by a simple parametrization from Bethe and Johnson [49]. 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

$$\rho(n) = 236 \times n^{2.54} + n m_n, \tag{8.20}$$

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

$$P = -\frac{\partial E}{\partial V}, \tag{8.21}$$

where E is the energy in units of MeV/c^2 we have

$$P(n) = n\frac{\partial \rho(n)}{\partial n} - \rho(n) = 363.44 \times n^{2.54}.$$

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

$$P_{new} = \frac{dP}{dr} + P_{old},$$

this comes as a function of r. However, having obtained the new pressure, you will need to use Eq. (8.1) 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. (8.1) and (8.20).
5. Write then a function which sets up the derivatives

$$-\frac{\hat{m}\hat{\rho}}{\hat{r}^2}, \qquad \hat{r}^2\hat{\rho}.$$

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

$$\frac{d\hat{P}}{d\hat{r}} = -\frac{(\hat{P}+\hat{\rho})(\hat{r}^3\hat{P}+\hat{m})}{\hat{r}^2-2\hat{m}\hat{r}},$$

   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.

8.2.

$$ml\frac{d^2\theta}{dt^2} + mgsin(\theta) = 0,$$

with an angular velocity and acceleration given by

$$v = l\frac{d\theta}{dt},$$

and

$$a = l\frac{d^2\theta}{dt^2}.$$

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

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = 0, \tag{8.22}$$

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

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = Asin(\omega t), \tag{8.23}$$

with A and ω two constants representing the amplitude and the angular frequency respectively. The latter is called the driving frequency.

1. Rewrite Eqs. (8.22) and (8.23) as dimensionless equations.
2. Write then a code which solves Eq. (8.22) 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.
3. Now we switch to Eq. (8.23) 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.

4. 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.
5. 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.

8.3. We assume that the orbit of Earth around the Sun is co-planar, and we take this to be the xy-plane. Using Newton's second law of motion we get the following equations

$$\frac{d^2x}{dt^2} = \frac{F_{G,x}}{M_{\mathrm{Earth}}},$$

and

$$\frac{d^2y}{dt^2} = \frac{F_{G,y}}{M_{\mathrm{Earth}}},$$

where $F_{G,x}$ and $F_{G,y}$ are the x and y components of the gravitational force.

a) Rewrite the above second-order ordinary differential equations as a set of coupled first order differential equations. Write also these equations in terms of dimensionless variables. As an alternative to the usage of dimensionless variables, you could also use so-called astronomical units (AU as abbreviation). If you choose the latter set of units, one astronomical unit of length, known as 1 AU, is the average distance between the Sun and Earth, that is 1 AU = 1.5×10^{11} m. It can also be convenient to use years instead of seconds since years match better the solar system. The mass of the Sun is $M_{\mathrm{sun}} = M_{\odot} = 2 \times 10^{30}$ kg. The mass of Earth is $M_{\mathrm{Earth}} = 6 \times 10^{24}$ kg. The mass of other planets like Jupiter is $M_{\mathrm{Jupiter}} = 1.9 \times 10^{27}$ kg and its distance to the Sun is 5.20 AU. Similar numbers for Mars are $M_{\mathrm{Mars}} = 6.6 \times 10^{23}$ kg and 1.52 AU, for Venus $M_{\mathrm{Venus}} = 4.9 \times 10^{24}$ kg and 0.72 AU, for Saturn are $M_{\mathrm{Saturn}} = 5.5 \times 10^{26}$ kg and 9.54 AU, for Mercury are $M_{\mathrm{Mercury}} = 2.4 \times 10^{23}$ kg and 0.39 AU, for Uranus are $M_{\mathrm{Uranus}} = 8.8 \times 10^{25}$ kg and 19.19 AU, for Neptun are $M_{\mathrm{Neptun}} = 1.03 \times 10^{26}$ kg and 30.06 AU and for Pluto are $M_{\mathrm{Pluto}} = 1.31 \times 10^{22}$ kg and 39.53 AU. Pluto is no longer considered a planet, but we add it here for historical reasons.

   Finally, mass units can be obtained by using the fact that Earth's orbit is almost circular around the Sun. For circular motion we know that the force must obey the following relation

$$F_G = \frac{M_{\mathrm{Earth}} v^2}{r} = \frac{GM_{\odot} M_{\mathrm{Earth}}}{r^2},$$

   where v is the velocity of Earth. The latter equation can be used to show that

$$v^2 r = GM_{\odot} = 4\pi^2 \mathrm{AU}^3/\mathrm{yr}^2.$$

   Discretize the above differential equations and set up an algorithm for solving these equations using the so-called Euler-Cromer.

b) Write then a program which solves the above differential equations for the Earth-Sun system using the Euler-Cromer method. Find out which initial value for the velocity that gives a circular orbit and test the stability of your algorithm as function of different time steps Δt. Find a possible maximum value Δt for which the Euler-Cromer method does not yield stable results. Make a plot of the results you obtain for the position of Earth (plot the x and y values) orbiting the Sun.

Check also for the case of a circular orbit that both the kinetic and the potential energies are constants. Check also that the angular momentum is a constant. Explain why these quantities are conserved.

c) Modify your code by implementing the fourth-order Runge-Kutta method and compare the stability of your results by repeating the steps in b). Compare the stability of the two methods, in particular as functions of the needed step length Δt. Comment your results.

d) Kepler's second law states that the line joining a planet to the Sun sweeps out equal areas in equal times. Modify your code so that you can verify Kepler's second law for the case of an elliptical orbit. Compare both the Runge-Kutta method and the Euler-Cromer method and check that the total energy and angular momentum are conserved. Why are these quantities conserved? A convenient choice of starting values are an initial position of 1 AU and an initial velocity of 5 AU/yr.

e) Consider then a planet which begins at a distance of 1 AU from the sun. Find out by trial and error what the initial velocity must be in order for the planet to escape from the sun. Can you find an exact answer?

f) We will now study the three-body problem, still with the Sun kept fixed at the center but including Jupiter (the most massive planet in the solar system, having a mass that is approximately 1000 times smaller than that of the Sun) together with Earth. This leads us to a three-body problem. Without Jupiter, Earth's motion is stable and unchanging with time. The aim here is to find out how much Jupiter alters Earth's motion.

The program you have developed can easily be modified by simply adding the magnitude of the force betweem Earth and Jupiter.

This force is given again by

$$F_{\mathrm{Earth-Jupiter}} = \frac{GM_{\mathrm{Jupiter}}M_{\mathrm{Earth}}}{r^2_{\mathrm{Earth-Jupiter}}},$$

where M_{Jupiter} is the mass of the sun and M_{Earth} is the mass of Earth. The gravitational constant is G and $r_{\mathrm{Earth-Jupiter}}$ is the distance between Earth and Jupiter.

We assume again that the orbits of the two planets are co-planar, and we take this to be the xy-plane. Modify your first-order differential equations in order to accomodate both the motion of Earth and Jupiter by taking into account the distance in x and y between Earth and Jupiter. Set up the algorithm and plot the positions of Earth and Jupiter using the fourth-order Runge-Kutta method. Include an adaptive solver to your Runge-Kutta method, using for example the adaptive scheme proposed by Fehlberg.

Discuss the stability of the solutions using the standard Runge-Kutta4 solver and the adaptive scheme.

Repeat the calculations by increasing the mass of Jupiter by a factor of 10 and 1000 and plot the position of Earth. Study again the stability of the standard and the adaptive Runge-Kutta solvers.

g) Finally, using your optimal Runge-Kutta solver, we carry out a real three-body calculation where all three systems, Earth, Jupiter and the Sun are in motion. To do this, choose the center-of-mass position of the three-body system as the origin rather than the position of the sun. Give the sun an initial velocity which makes the total momentum of the system exactly zero (the center-of-mass will remain fixed). Compare these results with those from the previous exercise and comment your results. Extend your program to include all planets in the solar system (if you have time, you can also include the various moons, but it is not required) and discuss your results. Try to find data for the initial positions and velocities for all planets.

h) The perihelion precession of Mercury. This part is optional but gives you an additional 30% on the final score!

An important test of the general theory of relativity was comparing its prediction for the perihelion precession of Mercury to the observed value. The observed value of the perihelion precession, when all classical effects (such as the perturbation of the orbit due to gravitational attraction from the other planets) are subtracted, is 43″ (43 arc seconds) per century.

Closed elliptical orbits are a special feature of the Newtonian $1/r^2$ force. In general, any correction to the pure $1/r^2$ behaviour will lead to an orbit which is not closed, i.e. after one complete orbit around the Sun, the planet will not be at exactly the same position as it started. If the correction is small, then each orbit around the Sun will be almost the same as the classical ellipse, and the orbit can be thought of as an ellipse whose orientation in space slowly rotates. In other words, the perihelion of the ellipse slowly precesses around the Sun.

You will now study the orbit of Mercury around the Sun, adding a general relativistic correction to the Newtonian gravitational force, so that the force becomes

$$F_G = \frac{GM_{\text{Sun}}M_{\text{Mercury}}}{r^2}\left[1+\frac{3l^2}{r^2c^2}\right]$$

where M_{Mercury} is the mass of Mercury, r is the distance between Mercury and the Sun, $l = |r \times v|$ is the magnitude of Mercury's orbital angular momentum per unit mass, and c is the speed of light in vacuum. Run a simulation over one century of Mercury's orbit around the Sun with no other planets present, starting with Mercury at perihelion on the x axis. Check then the value of the perihelion angle θ_{p}, using

$$\tan\theta_{\text{p}} = \frac{y_{\text{p}}}{x_{\text{p}}}$$

where x_{p} (y_{p}) is the x (y) position of Mercury at perihelion, i.e. at the point where Mercury is at its closest to the Sun. You may use that the speed of Mercury at perihelion is $12.44\,\text{AU/yr}$, and that the distance to the Sun at perihelion is $0.3075\,\text{AU}$. You need to make sure that the time resolution used in your simulation is sufficient, for example by checking that the perihelion precession you get with a pure Newtonian force is at least a few orders of magnitude smaller than the observed perihelion precession of Mercury. Can the observed perihelion precession of Mercury be explained by the general theory of relativity?

8.4. In this exercise we will implement a molecular dynamics (MD) code to model the behavior of a system of Argon atoms, and use this model to study statistical properties of the system. In all calculations, we will use so-called MD units. These assume that all the particles in a simulation are identical, so the masses and LJ parameters can be factored out of the equations. You will need to insert $A = \bar{A}A_0$ for every variable quantity A in equations 8.25-8.30 above. For example, for velocity, $v = \bar{v}\frac{L_0}{t_0}$. The time step Δt must also be treated this way.

Quantity	Conversion factor	Value
Length	$L_0 = \sigma$	3.405 Å
Time	$t_0 = \sigma\sqrt{m/\varepsilon}$	$2.1569 \cdot 10^3$ fs
Force	$F_0 = m\sigma/t_0^2 = \varepsilon/\sigma$	$3.0303 \cdot 10^{-1}$ eV/Å
Energy	$E_0 = \varepsilon$	$1.0318 \cdot 10^{-2}$ eV
Temperature	$T_0 = \varepsilon/k_{\text{B}}$	119.74 K

Table 8.1 Conversion factors A_0 from MD units for variable quantities.

In case you want to convert between your internal MD units and other units during input and output, the actual values of the conversion factors are listed in table 8.1. These are calculated using the argon mass, lattice constant and LJ parameters: $m = 39.948$ amu, $a = 5.260$

Å (solid argon), $\sigma = 3.405$ Å, $\varepsilon = 1.0318 \cdot 10^{-2}$ eV. Another common practice is putting $E_0 = 4\varepsilon$, affecting the conversion factors F_0, T_0 and t_0.

Normally distributed random numbers are obtained by performing a Box-Muller transform on uniformly distributed numbers. Let u and v be uniform numbers in the interval $(-1,1)$. These numbers will only be accepted for the transformation if $s = u^2 + v^2$ is in the interval $(0,1)$. In that case, we obtain two normally distributed numbers n_1 and n_2 by multiplying u and v with a constant,

$$n_1 = Su, \qquad n_2 = Sv, \qquad S = \sqrt{\frac{-\ln s}{s}}. \tag{8.24}$$

n_1 and n_2 will have standard deviations of 1, but multiplying all generated numbers with a constant will give a distribution with that constant as the standard deviation.

a) Write a program that generates an $N_c \times N_c \times N_c$ unit cell face centered cubic lattice of argon atoms. If you use an object oriented programming language, each atom and/or the entire lattice should be objects of a class.
For easy testing of the lattice arranger and the later MD implementation, you should already visualize your atoms. VMD is a visualization program with a simple output format and pretty graphics. It can be downloaded and run from your home area, and a description of its output format can be found in the appendix.
b) Consider first free particles with initial independent Maxwell-Boltzmann distributed velocities. These correspond to normally distributed values with standard deviation $\sqrt{k_\mathrm{B}T/m}$ for the desired temperature T. In a system of N atoms, all $3N$ velocity components v are set using

$$v = \sqrt{k_\mathrm{B}T/m}\xi \tag{8.25}$$

where ξ is a normally distributed number with mean 0 and standard deviation 1. Remove any initial total linear momentum from the system.
Integrate the dynamical equation (N2L) using the symplectic and numerically stable velocity Verlet algorithm. For each particle i, the steps are as follows (currently setting $U_i = 0$):

$$\mathbf{v}_i(t+\Delta t/2) = \mathbf{v}_i(t) + \frac{\mathbf{F}_i(t)}{2m}\Delta t \tag{8.26}$$
$$\mathbf{r}_i(t+\Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t+\Delta t/2)\Delta t \tag{8.27}$$
$$\mathbf{F}_i(t+\Delta t) = -\nabla_i U_i(\{\mathbf{r}\}(t+\Delta t)) \tag{8.28}$$
$$\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i(t+\Delta t/2) + \frac{\mathbf{F}_i(t+\Delta t)}{2m}\Delta t \tag{8.29}$$

The particles will now spread out into space. We are only interested in bulk atoms in a material, so the next step is implementing periodic boundary conditions. Every time the position of a particle is updated, the program must check if it has gone though one of the sides.
c) Create a function for calculating the force between all particles. Use the Lennard-Jones potential, which has the following form:

$$U_{ij}(r_{ij}) = 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \tag{8.30}$$

where $r_{ij} = r_i - r_j$. Differentiate the expression analytically for finding the force. Summing up the potential energy between all particles can also be useful.
Use the minimum image convention when calculating the distance between particles. E.g. the distance between atoms/atom replicae i and j in the x direction becomes $\min_\delta(x_i - x_j + \delta L)$

where $\delta \in \{-1,0,1\}$ and L is the length of the simulation box in the x direction. This limits the interaction range to half the system size, which is more than enough for our potential. You should now have a working MD program for simulating bulk argon in its solid, liquid and gas phases.

d) You probably notice that the force calculations are the most time consuming part of your program. The number of force terms is $\frac{1}{2}N(N-1)$ for each time step, which gives a workload scaling $\propto N^2$. We want to improve this by neglecting force terms for particles far apart. The LJ interaction is short ranged and can be neglected for distances over $r_{\text{cut}} \approx 3\varepsilon$. A simple and efficient way of achieving this is by implementing Verlet lists.

   Create arrays specifying the neighbours of all particles and a function to update this list e.g. every 10th timestep. An atom i needs only to keep track of neighbour atoms with a lower index j. The force loops can now iterate over all atoms i and atoms j in the neighbour list of i. Compare the time usage of the program with and without Verlet lists for different system sizes.

e) An MD simulation of bulk material enables the measurement of macroscopic quantities. The ergodic hypothesis states that the time a system has one particular value of an observable A is proportional to the phase space volume where A has this value. This applies to systems in equilibrium studied for a long period of time. As a result, the time average and ensemble average of a variable are equal. If we average over long enough periods of time, we can predict equilibrium properties of real materials.

   According to the central limit theorem, the velocity distribution of the particles will eventually evolve into a Maxwell-Boltzmann-distribution whatever the initial condition. Switch to initializing the velocities with uniformly distributed random numbers in the interval $[-v, v]$, for a reasonable v. Investigate the velocity distribution after equilibration, e.g. by dumping the velocities to a file and using the Matlab `hist()` function. Roughly how much time does it take for the velocities to reach a MB-distribution?

   The easiest quantity to calculate is the total energy of the system. Sum up the kinetic and potential energies of all your argon atoms. Output the total energy for each time step of a simulation. The energy should be conserved, but some fluctuations are inevitable as the dynamics are discretized. How does the size of the fluctuations depend on the time step Δt?

   The temperature of a MD system is non-trivial to calculate for general potential forms. The simplest estimate assumes equilibrium between the translational and potential degrees of freedom. According to the equipartition principle, the total kinetic energy is

$$E_k = \frac{3}{2} N k_{\text{B}} T \tag{8.31}$$

   where N is the number of atoms and T is our estimate for the system temperature.

   Invert the equation and measure the temperature for each time step. Don't forget to equilibriate the system first. What mean temperature does the system settle on, and how does this compare to the initial temperature? How does the temperature fluctuations vary with the system size?

   There are several ways of measuring the pressure P of a many-atom system. The method we will use is derived from the virial equation for the pressure. In a volume V with particle density $\rho = N/V$, the average pressure is

$$P = \rho k_{\text{B}} T + \frac{1}{3V} \langle \sum_{i<j} \mathbf{F}_{ij} \cdot \mathbf{r}_{ij} \rangle \tag{8.32}$$

   where the sum runs over all interacting particle pairs. The vector products should be computed and summed up inside the force loops for efficiency.

f) In order to simulate the canonical ensemble, interactions with an external heat bath must be taken into account. Many methods have been suggested in order to achieve this, all with their pros and cons. Requirements for a good thermostat are:

- Keeping the system temperature around the heat bath temperature
- Sampling the phase space corresponding to the canonical ensemble
- Tunability
- Preservation of dynamics

The method closest to fullfilling these requirements which is in widespread use is the Nosé-Hoover thermostat, which is somewhat complicated to implement. We will focus on simpler methods. They will require negligble CPU time and should be applied for each time step.
Many thermostats work by rescaling the velocities of all atoms by multiplying them with a factor γ. The Berendsen thermostat uses

$$\gamma = \sqrt{1 + \frac{\Delta t}{\tau}\left(\frac{T_{\text{bath}}}{T} - 1\right)} \tag{8.33}$$

with τ as the relaxation time, tuning the coupling to the heat bath. Though it satisfies Fouriers law of heat transfer (the transfered heat between two bodies is proportional to their temperature difference) it does a poor job at sampling the canonical ensemble.
Implement the Berendsen thermostat as a function in your code. $\tau = \Delta t$ will keep the (estimated) temperature exactly constant. It should be put to 10-20 times this value.
The Andersen thermostat simulates (hard) collisions between atoms inside the system and in the heat bath. Atoms which collide will gain a new normally distributed velocity with standard deviation $\sqrt{k_{\text{B}}T_{\text{bath}}/m}$. For all atoms, a random uniformly distributed number in the interval $[0,1]$ is generated. If this number is less than $\frac{\Delta t}{\tau}$, the atom is assigned a new velocity. In this case, τ is treated as a collision time, and should have about the same value as the τ in the Berendsen thermostat. The Andersen thermostat is very useful when equilibrating systems, but disturbs the dynamics of e.g. lattice vibrations.
Implement the Andersen thermostat, and compare $T(t)$ graphs for simulations using the different methods. Again, be aware that our T is just an approximation to the real temperature. Differences can also be seen in the dynamics.

g) In a volume with PBC and atoms consituting a fluid, self-diffusion can be simulated. We are to measure the self-diffusion constant D for liquid argon. This is achieved by finding the mean square displacement of all atoms after a given time,

$$\langle r^2(t)\rangle = \frac{1}{N}\sum_{i=1}^{N}(r(t) - r_{\text{initial}}). \tag{8.34}$$

From diffusion theory, we know that $\langle r^2(t)\rangle = 6Dt$ for a random walk in three dimensions, which is a good approximation to the motion of an atom in a fluid. Plot the mean square displacement as a function of time and extract the diffusion constant. Investigate the effect of temperature by finding D for some temperatures in the liquid phase of argon. Remember that you are measuring the total distance travelled by the atoms, which must be continous when an atom is displaced though a PBC boundary.

h) A radial distribution function $g(r)$, also called a pair correlation function, is a tool for characterizing the microscopic structure of a fluid. It is interpreted as the radial probability for finding another atom a distance r from an arbitrary atom, or equivalently, the atomic density in a spherical shell of radius r around an atom. It is commonly normalized by dividing it with the average particle density so that $\lim_{r\to\infty} g(r) = 1$.

Estimate $g(r)$ for $r \in (0, \frac{L}{2}]$ in your argon system. The easiest way is to divide the distance interval into bins, loop over all pairs of particles and count how many distances belong in each bin. Time-averaging the function gives a better description of the system's general behaviour. Plot $g(r)$ for temperatures where the system is in solid and liquid phases. Does it appear as expected? How would the exact $g(r)$ look for a perfect crystal?

Chapter 9
Two point boundary value problems

Abstract When differential 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.

9.1 Introduction

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 7 and 10 focus on finite difference methods. Chapter 7 solves the finite difference problem as an eigenvalue problem for a one variable differential equation while in chapter 10 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. [50] 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. [51, 52]. 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 7 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.

9.2 Shooting methods

In many physics applications we encounter differential equations like

$$\frac{d^2y}{dx^2}+k^2(x)y=F(x);\ a\leq x\leq b, \tag{9.1}$$

with boundary conditions

$$y(a)=\alpha,\ y(b)=\beta. \tag{9.2}$$

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-order 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

$$\frac{y_{i+1}+y_{i-1}-2y_i}{h^2}+O(h^2)+k_i^2y_i=F_i.$$

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

$$y_{i+1}\approx -y_{i-1}+y_i\left(2-h^2k_i^2+h^2F_i\right). \tag{9.3}$$

Starting at $i=1$ we have after one step

$$y_2\approx -y_0+y_1\left(2-h^2k_1^2+h^2F_1\right).$$

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

$$y_i'\approx\frac{y_{i+1}-y_i}{h}.$$

This means that we have $y_1\approx y_0+hy_0'$.

9.2.1 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

$$y(x+h) = y(x) + hy^{(1)}(x) + \frac{h^2}{2!}y^{(2)}(x) + \frac{h^3}{3!}y^{(3)}(x) + \frac{h^4}{4!}y^{(4)}(x) + \cdots$$

Here $y^{(n)}(x)$ is a shorthand notation for the nth derivative $d^n y/dx^n$. 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

$$y^{(2)}(x) = \frac{y(x+h) + y(x-h) - 2y(x)}{h^2} - \frac{h^2}{12}y^{(4)}(x) + O(h^6).$$

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

$$\frac{d^2(k^2 y(x))}{dx^2} \approx \frac{\left(k^2(x+h)y(x+h) + k^2(x)y(x)\right) + \left(k^2(x-h)y(x-h) + k^2(x)y(x)\right)}{h^2}.$$

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}

$$y_{i+1} = \frac{2\left(1 - \frac{5}{12}h^2 k_i^2\right)y_i - \left(1 + \frac{1}{12}h^2 k_{i-1}^2\right)y_{i-1}}{1 + \frac{h^2}{12}k_{i+1}^2} + O(h^6),$$

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

$$y_{i+1} = \frac{2\left(1 - \frac{5}{12}h^2 k_i^2\right)y_i - \left(1 + \frac{1}{12}h^2 k_{i-1}^2\right)y_{i-1}}{1 + \frac{h^2}{12}k_{i+1}^2} + \frac{h^2}{12}\left(F_{i+1} + F_{i-1} - 2F_i\right) + O(h^6).$$

Starting at $i = 1$ results in, using the boundary condition $y_0 = 0$,

$$y_2 = \frac{2\left(1 - \frac{5}{12}h^2 k_1 y_1\right) - \left(1 + \frac{1}{12}h^2 k_0^2 y_0\right)}{1 + \frac{h^2}{12}k_2^2} + \frac{h^2}{12}\left(F_2 + F_0 - 2F_1\right) + O(h^6).$$

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.

9.2.2 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

$$F = T sin(\theta + \Delta\theta) - T sin(\theta) = T sin(\theta_{i+1}) - T sin(\theta_i).$$

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

$$y'' + \lambda y = 0;\ y(0) = 0,\ y(L) = 0.$$

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 10.

If $\lambda > 0$ the above wave equation has a solution of the form

$$y(x) = Acos(\alpha x) + Bsin(\alpha x),$$

and imposing the boundary conditions results in an infinite sequence of solutions of the form

$$y_n(x) = sin(\frac{n\pi x}{L}),\ n = 1,2,3,\ldots$$

with eigenvalues

$$\lambda_n = \frac{n^2\pi^2}{L^2},\ n = 1,2,3,\ldots$$

For $\lambda = 0$ we have

$$y(x) = Ax + B,$$

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

$$\omega_n = \sqrt{\frac{\lambda_n T}{\rho}} = \frac{n\pi}{L}\sqrt{\frac{T}{\rho}},\ n = 1,2,3,\ldots,$$

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

$$y_2 \approx -y_0 + y_1\left(2 - h^2k_1^2 + h\right) = 4h\pi cos(4\pi x_0)\left(2 - 16h^2\pi^2\right) = 4\pi\left(2 - 16h^2\pi^2\right).$$

If we split our interval $x \in [0,1]$ into 10 equally spaced points we arrive at the results displayed in Table 9.1. 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 increasing discrepancy we see towards the endpoints.

Table 9.1 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

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 4.

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 type, see Eq. (14.6), (all physical costants are set equal to one, that is $m = c = \hbar = k = 1$)

$$-\frac{1}{2}\frac{d^2y}{dx^2} + \frac{1}{2}x^2y = \varepsilon y;\ -\infty < x < \infty,$$

with boundary conditions $y(-\infty) = y(\infty) = 0$. For the lowest lying state, the eigenvalue is $\varepsilon = 1/2$ and the eigenfunction is

Table 9.2 Integrated and exact solution of the differential equation $-y'' + x^2 y = 2\varepsilon y$ 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

$$y(x) = \left(\frac{1}{\pi}\right)^{1/4} \exp(-x^2/2).$$

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 9.2 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.

9.2.3 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

$$\widehat{\mathbf{H}}\psi(r) = (\widehat{\mathbf{T}} + \widehat{\mathbf{V}})\psi(r) = E\psi(r). \tag{9.4}$$

In detail this gives

$$\left(-\frac{\hbar^2}{2m}\nabla^2 + V(r)\right)\psi(r) = E\psi(r). \tag{9.5}$$

The eigenfunction in spherical coordinates takes the form

$$\psi(r) = R(r)Y_l^m(\theta,\phi), \tag{9.6}$$

and the radial part $R(r)$ is a solution to

$$-\frac{\hbar^2}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr} - \frac{l(l+1)}{r^2}\right)R(r) + V(r)R(r) = ER(r). \tag{9.7}$$

Then we substitute $R(r) = (1/r)u(r)$ and obtain

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r) + \left(V(r) + \frac{l(l+1)}{r^2}\frac{\hbar^2}{2m}\right)u(r) = Eu(r). \tag{9.8}$$

We introduce a dimensionless variable $\rho = (1/\alpha)r$ where α is a constant with dimension length and get

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(\rho) + \left(V(\rho) + \frac{l(l+1)}{\rho^2}\frac{\hbar^2}{2m\alpha^2}\right)u(\rho) = Eu(\rho). \tag{9.9}$$

In our case we are interested in attractive potentials

$$V(r) = -V_0 f(r), \tag{9.10}$$

where $V_0 > 0$ and analyze bound states where $E < 0$. The final equation can be written as

$$\frac{d^2}{d\rho^2}u(\rho) + k(\rho)u(\rho) = 0, \tag{9.11}$$

where

$$\begin{aligned} k(\rho) &= \gamma\left(f(\rho) - \frac{1}{\gamma}\frac{l(l+1)}{\rho^2} - \varepsilon\right) \\ \gamma &= \frac{2m\alpha^2 V_0}{\hbar^2} \\ \varepsilon &= \frac{|E|}{V_0} \end{aligned} \tag{9.12}$$

9.2.3.1 Schrödinger equation for a spherical box potential

Let us now specify the spherical symmetric potential to

$$f(r) = \begin{cases} 1 \\ -0 \end{cases} \text{for} \quad \begin{matrix} r \le a \\ r > a \end{matrix} \tag{9.13}$$

and choose $\alpha = a$. Then

$$k(\rho) = \gamma\begin{cases} 1 - \varepsilon - \frac{1}{\gamma}\frac{l(l+1)}{\rho^2} \\ -\varepsilon - -\frac{1}{\gamma}\frac{l(l+1)}{\rho^2} \end{cases} \text{for} \quad \begin{matrix} r \le a \\ r > a \end{matrix} \tag{9.14}$$

The eigenfunctions in Eq. (9.5) are subject to conditions which limit the possible solutions. Of importance for the present example is that $u(r)$ must be finite everywhere and $\int|u(r)|^2 d\tau$ 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 $r \longrightarrow \infty$.

9.2.3.2 Analysis of $u(\rho)$ at $\rho=0$

For small ρ Eq. (9.11) reduces to

$$\frac{d^2}{d\rho^2}u(\rho)-\frac{l(l+1)}{\rho^2}u(\rho)=0, \tag{9.15}$$

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

$$u(\rho)=\rho^{l+1} \quad \text{for small } \rho \tag{9.16}$$

9.2.3.3 Analysis of $u(\rho)$ for $\rho \longrightarrow \infty$

For large ρ Eq. (9.11) reduces to

$$\frac{d^2}{d\rho^2}u(\rho)-\gamma\varepsilon u(\rho)=0 \quad \gamma>0, \tag{9.17}$$

with solutions $u(\rho)=\exp(\pm\gamma\varepsilon\rho)$ and the condition for large ρ means that our numerical solution must satisfy

$$u(\rho)=e^{-\gamma\varepsilon\rho} \quad \text{for large } \rho \tag{9.18}$$

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.

9.3 Numerical procedure, shooting and matching

The eigenvalue problem in Eq. (9.11) 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 $u(\rho)=e^{-\gamma\varepsilon\rho}$. However, due to numerical inaccuracy the solution will contain small admixtures of the undesireable exponential growing function $u(\rho)=e^{+\gamma\varepsilon\rho}$. 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

$$\frac{\frac{d}{d\rho}u^{<}(\rho)}{u^{<}(\rho)} = \frac{\frac{d}{d\rho}u^{>}(\rho)}{u^{>}(\rho)} \quad \text{at} \quad \rho = \rho_m. \tag{9.19}$$

We can modify this expression by normalizing the function $u^{<}u^{<}(\rho_m) = Cu^{>}u^{>}(\rho_m)$. Then Eq. (9.19) becomes

$$\frac{d}{d\rho}u^{<}(\rho) = \frac{d}{d\rho}u^{>}(\rho) \quad \text{at} \quad \rho = \rho_m \tag{9.20}$$

For an arbitary value of the eigenvalue Eq. (9.19) will not be satisfied. Thus the numerical procedure will be to iterate for different eigenvalues until Eq. (9.20) is satisfied.

We can calculate the first order derivatives by

$$\begin{aligned} \frac{d}{d\rho}u^{<}(\rho_m) &\approx \frac{u^{<}(\rho_m) - u^{<}(\rho_m - h)}{h} \\ \frac{d}{d\rho}u^{>}(\rho_m) &\approx \frac{u^{>}(\rho_m) - u^{>}(\rho_m + h)}{h} \end{aligned} \tag{9.21}$$

Thus the criterium for a proper eigenfunction will be

$$f = u^{>}(\rho_m + h) - u^{<}(\rho_m - h) = 0. \tag{9.22}$$

9.3.1 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 4.

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 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 4 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. (9.22)

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.
//

double 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 (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.

9.4 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

$$-u(x)'' = f(x), \qquad x \in (0,1), \qquad u(0) = u(1) = 0, \tag{9.23}$$

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. [52, 53] 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 $y \ne x$ 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 $x = y - \varepsilon$ to $x = y + \varepsilon$, with ε as an infinitesmally small number. We obtain then

$$\frac{dG}{dx}|_{x=y+\varepsilon} - \frac{dG}{dx}|_{x=y-\varepsilon} = 1.$$

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 [52, 53]

$$W = \frac{du_>}{dx} u_< - \frac{du_<}{dx} u_>,$$

and using

$$\frac{dG}{dx}|_{x=y+\varepsilon} - \frac{dG}{dx}|_{x=y-\varepsilon} = 1,$$

and one can then show that the Green's function reads

$$G(x,y) = u_<(x_<) u_>(x_>), \tag{9.24}$$

where $x_<$ is defined for $x = y - \varepsilon$ and $x_> = y + \varepsilon$. Using the definition of the Green's function in Eq. (9.24) we can now solve Eq. (9.23) for $x \in [a,b]$ using

$$u(x)=u^{>}(x)\int_a^x u^{<}(x')f(x')dx'+u^{<}(x)\int_x^b u^{>}(x')f(x')dr' \tag{9.25}$$

The algorithm for solving Eq. (9.23) 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.

A possible algorithm could be phrased as follows:

- Compute the solution of the homogeneous part of Eq. (9.23) 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 5.
- Compute thereafter the inward integral of the Green's function approach. Adding these two integrals gives the resulting wave function of Eq. (9.25).

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()
```

9.5 Exercises

9.1. 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 6. Which method would you prefer?

9.2. 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

$$-\frac{\hbar^2}{2m}\frac{d^2u(r)}{dr^2} + V(r)u(r) = Eu(r), \tag{9.26}$$

with

$$m = 2\frac{m_p m_n}{m_p + m_n}, \tag{9.27}$$

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

$$V(r) = \begin{cases} 0 & r > a \\ -V_0 & 0 < r \le a \\ \infty & r \le 0 \end{cases}, \tag{9.28}$$

displayed in Fig 9.1.

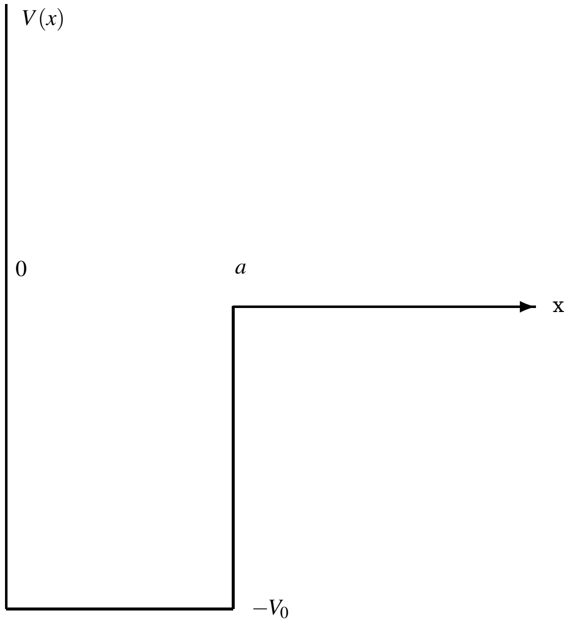


Fig. 9.1 Example of a finite box potential with value $-V_0$ in $0 < x \le a$, infinitely large for $x \le 0$ and zero else.

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)

$$\frac{d^2u(r)}{dr^2}+\frac{m}{\hbar^2}(V_0+E)u(r)=0 \qquad r<a, \tag{9.29}$$

and

$$\frac{d^2u(r)}{dr^2}+\frac{m}{\hbar^2}Eu(r)=0 \qquad r>a. \tag{9.30}$$

1. 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

   $$u(r)=Asin(kr) \qquad r<a, \tag{9.31}$$

   and

   $$u(r)=B\exp(-\beta r) \qquad r>a, \tag{9.32}$$

   where A and B are constants. We have also defined

   $$k=\sqrt{m(V_0-|E|)}/\hbar, \tag{9.33}$$

   and

   $$\beta=\sqrt{m|E|}/\hbar. \tag{9.34}$$

   Show then, using the continuity requirement on the wave function that at $r=a$ you obtain the transcendental equation

   $$kcot(ka)=-\beta. \tag{9.35}$$

2. Insert values of $V_0=60$ MeV and $a=1.45$ fm (1 fm = 10^{-15} m) and make a plot of Eq. (9.35) 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. (9.35) is satisfied, obtain a numerical value for E using for example Newton-Raphson's method or similar methods, see chapter 4. 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?
3. 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.
4. 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

   $$V(r)=V_a\frac{e^{-ax}}{x}+V_b\frac{e^{-bx}}{x}+V_c\frac{e^{-cx}}{x} \tag{9.36}$$

   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.

Chapter 10
Partial Differential Equations

Abstract Partial differential equations play an important role in our modelling of physical processes, from diffusion of heat to our understanding of Tsunamis. In this chapter we present some of the basic methods using finite difference methods.

10.1 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

$$\frac{\partial^2 u}{\partial x^2} = A\frac{\partial^2 u}{\partial t^2}, \tag{10.1}$$

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

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = A\frac{\partial^2 u}{\partial t^2},$$

In Chapter 12 we will see 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

$$\frac{\partial^2 u}{\partial x^2} = A\frac{\partial u}{\partial t}, \tag{10.2}$$

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. (10.1) except that we have set $A = 0$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \tag{10.3}$$

or if we have a finite electric charge represented by a charge density $\rho(\mathbf{x})$ we have the familiar Poisson equation

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=-4\pi\rho(\mathbf{x}). \tag{10.4}$$

Other famous partial differential equations are the Helmholtz (or eigenvalue) equation, here specialized to two dimensions only

$$-\frac{\partial^2 u}{\partial x^2}-\frac{\partial^2 u}{\partial y^2}=\lambda u, \tag{10.5}$$

the linear transport equation (in $2+1$ dimensions) familiar from Brownian motion as well

$$\frac{\partial u}{\partial x}+\frac{\partial u}{\partial x}+\frac{\partial u}{\partial y}=0, \tag{10.6}$$

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. [54] 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. [50, 55].

As in the previous chapters we will focus mainly on widely used algorithms for solutions of partial differential equations. A text like Evans' [54] 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 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. [50,52,55,56].

A general partial differential equation with two given dimensions reads

$$A(x,y)\frac{\partial^2 u}{\partial x^2}+B(x,y)\frac{\partial^2 u}{\partial x\partial y}+C(x,y)\frac{\partial^2 u}{\partial y^2}=F(x,y,u,\frac{\partial u}{\partial x},\frac{\partial u}{\partial y}),$$

and if we set

$$B = C = 0,$$

we recover the $1+1$-dimensional diffusion equation which is an example of a so-called parabolic partial differential equation. With

$$B = 0, \qquad AC < 0$$

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. (10.3) 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 possible implementations.

10.2 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 6. 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. [57]. 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

$$\frac{\kappa}{C\rho}\nabla^2 T(\mathbf{x},t) = \frac{\partial T(\mathbf{x},t)}{\partial t}$$

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.,

$$\frac{\kappa}{C\rho(\mathbf{x},t)}\nabla^2 T(\mathbf{x},t) = \frac{\partial T(\mathbf{x},t)}{\partial t}.$$

Setting all constants equal to the diffusion constant D, i.e.,

$$D = \frac{C\rho}{\kappa},$$

we arrive at

$$\nabla^2 T(\mathbf{x},t) = D\frac{\partial T(\mathbf{x},t)}{\partial t}.$$

Specializing to the $1+1$-dimensional case we have

$$\frac{\partial^2 T(x,t)}{\partial x^2} = D\frac{\partial T(x,t)}{\partial t}.$$

We note that the dimension of D is time/length^2. Introducing the dimensionless variables $\alpha\hat{x} = x$ we get

$$\frac{\partial^2 T(x,t)}{\alpha^2\partial\hat{x}^2} = D\frac{\partial T(x,t)}{\partial t},$$

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

$$\frac{\partial^2 T(\hat{x},\hat{t})}{\partial\hat{x}^2} = \frac{\partial T(\hat{x},\hat{t})}{\partial\hat{t}}.$$

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 $T(\hat{x},\hat{t}) \to u(x,t)$.

10.2.1 Explicit Scheme

In one dimension we have the following equation

$$\nabla^2 u(x,t) = \frac{\partial u(x,t)}{\partial t},$$

or

$$u_{xx} = u_t,$$

with initial conditions, i.e., the conditions at $t = 0$,

$$u(x,0) = g(x) \qquad 0 < x < L$$

with $L = 1$ the length of the x-region of interest. The boundary conditions are

$$u(0,t) = a(t) \qquad t \geq 0,$$

and

$$u(L,t) = b(t) \qquad t \geq 0,$$

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

$$\Delta x = \frac{1}{n+1}$$

and the time step length Δt. The position after i steps and time at time-step j are now given by

$$\begin{array}{l} t_j = j\Delta t \qquad j \geq 0 \\ x_i = i\Delta x \; 0 \leq i \leq n+1 \end{array}$$

If we then use standard approximations for the derivatives we obtain

$$u_t \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t) - u(x_i,t_j)}{\Delta t}$$

with a local approximation error $O(\Delta t)$ and

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2},$$

or

$$u_{xx} \approx \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2},$$

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

$$u_t \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t},$$

and

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}.$$

The one-dimensional diffusion equation can then be rewritten in its discretized version as

$$\frac{u_{i,j+1} - u_{i,j}}{\Delta t} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}.$$

Defining $\alpha = \Delta t/\Delta x^2$ results in the explicit scheme

$$u_{i,j+1} = \alpha u_{i-1,j} + (1-2\alpha)u_{i,j} + \alpha u_{i+1,j}. \tag{10.7}$$

Since all the discretized initial values

$$u_{i,0} = g(x_i),$$

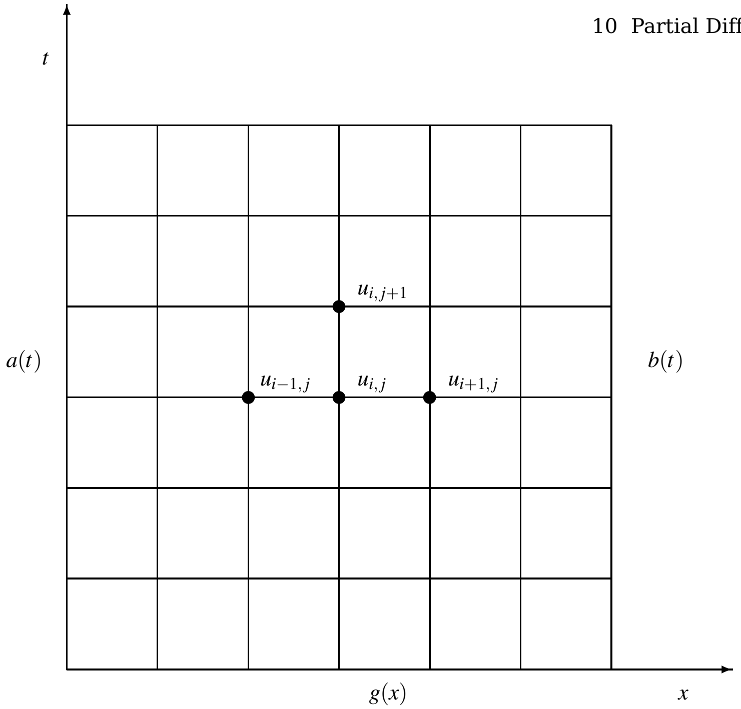


Fig. 10.1 Discretization of the integration area used in the solution of the $1+1$-dimensional diffusion equation. This discretization is often called calculational molecule.

are known, then after one time-step the only unknown quantity is $u_{i,1}$ which is given by

$$u_{i,1} = \alpha u_{i-1,0} + (1-2\alpha)u_{i,0} + \alpha u_{i+1,0} = \alpha g(x_{i-1}) + (1-2\alpha)g(x_i) + \alpha g(x_{i+1}).$$

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. (10.7). The procedure is depicted in Fig. 10.1.

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 $t_j = j\Delta t$

$$V_j = \begin{bmatrix} u_{1,j} \\ u_{2,j} \\ \dots \\ u_{n,j} \end{bmatrix}.$$

This results in a matrix-vector multiplication

$$V_{j+1} = \hat{A}V_j$$

with the matrix $\hat{A}$ given by

$$\hat{A} = \begin{bmatrix} 1-2\alpha & \alpha & 0 & 0\dots \\ \alpha & 1-2\alpha & \alpha & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & \alpha & 1-2\alpha \end{bmatrix}$$

which means we can rewrite the original partial differential equation as a set of matrix-vector multiplications

$$V_{j+1} = \hat{A}V_j = \cdots = \hat{A}^{j+1}V_0,$$

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
// Armadillo notation for vectors
   u(0) = unew(0) = u(n) = unew(n) = 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 integration
  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

$$\Delta t/\Delta x^2 \leq 1/2.$$

This means that if $\Delta x = 0.01$ (a rather frequent choice), then $\Delta t = 5\times 10^{-5}$. 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

$$\rho(\hat{A}) < 1, \tag{10.8}$$

see for example chapter 10 of Ref. [28] or chapter 4 of [23] for proofs. The spectral radius is defined as

$$\rho(\hat{A}) = \max\left\{|\lambda| : \det(\hat{A} - \lambda\hat{I}) = 0\right\},$$

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. (10.8) is always satisfied.

We can obtain closed-form 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

$$\hat{B} = \begin{bmatrix} 2 & -1 & 0 & 0 & \dots \\ -1 & 2 & -1 & 0 & \dots \\ \dots & \dots & \dots & \dots & -1 \\ 0 & 0 & \dots & -1 & 2 \end{bmatrix}.$$

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

$$(\hat{B}\hat{x})_i = \mu_i x_i,$$

resulting in

$$(\hat{B}\hat{x})_i = \sum_{j=1}^{n} \left(2\delta_{ij} - \delta_{i+1j} - \delta_{i-1j}\right) x_j = 2x_i - x_{i+1} - x_{i-1} = \mu_i x_i.$$

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

$$2\sin(i\theta) - \sin((i+1)\theta) - \sin((i-1)\theta) = \mu_i \sin(i\theta),$$

or

$$2(1-\cos(\theta))\sin(i\theta) = \mu_i \sin(i\theta),$$

which is nothing but

$$2(1-\cos(\theta))x_i = \mu_i x_i,$$

with eigenvalues $\mu_i = 2 - 2\cos(\theta)$.

Our requirement in Eq. (10.8) results in

$$-1 < 1 - \alpha 2(1-\cos(\theta)) < 1,$$

which is satisfied only if $\alpha < (1-\cos(\theta))^{-1}$ resulting in $\alpha \leq 1/2$ or $\Delta t/\Delta x^2 \leq 1/2$.

A more general tridiagonal matrix

$$\hat{A} = \begin{bmatrix} a & b & 0 & 0 & \dots \\ c & a & b & 0 & \dots \\ \dots & \dots & \dots & \dots & b \\ 0 & 0 & \dots & c & a \end{bmatrix},$$

has eigenvalues $\mu_i = a + s\sqrt{bc}\cos(i\pi/n+1)$ with $i = 1:n$, see for example Ref. [56] for a derivation using a finite difference scheme.

10.2.2 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

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t}.$$

However, there is nothing which hinders us from using the backward formula

$$u_t \approx \frac{u(x_i, t_j) - u(x_i, t_j - \Delta t)}{\Delta t},$$

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

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j - \Delta t)}{2\Delta t},$$

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

$$u_{xx} \approx \frac{u(x_i+\Delta x,t_j)-2u(x_i,t_j)+u(x_i-\Delta x,t_j)}{\Delta x^2},$$

and define again $\alpha = \Delta t/\Delta x^2$. We obtain now

$$u_{i,j-1} = -\alpha u_{i-1,j} + (1-2\alpha)u_{i,j} - \alpha u_{i+1,j}.$$

Here $u_{i,j-1}$ is the only unknown quantity. Defining the matrix $\hat{A}$

$$\hat{A} = \begin{bmatrix} 1+2\alpha & -\alpha & 0 & 0 & \dots \\ -\alpha & 1+2\alpha & -\alpha & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & -\alpha \\ 0 & 0 & \dots & -\alpha & 1+2\alpha \end{bmatrix},$$

we can reformulate again the problem as a matrix-vector multiplication

$$\hat{A}V_j = V_{j-1}$$

meaning that we can rewrite the problem as

$$V_j = \hat{A}^{-1}V_{j-1} = \hat{A}^{-1}\left(\hat{A}^{-1}V_{j-2}\right) = \dots = \hat{A}^{-j}V_0.$$

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 6. 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. 10.2.

10.2.2.1 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 6.

```
// parts of the function for backward Euler
void backward_euler(int n, int tsteps, double delta_x, double alpha)
{
  double a, b, c;
  vec u(n+1); // This is u of Au = y
  vec y(n+1); // Right side of matrix equation Au=y, the solution at a previous step

  // Initial conditions
  for (int i = 1; i < n; i++) {
    y(i) = u(i) = func(delta_x*i);
  }
  // Boundary conditions (zero here)
  y(n) = u(n) = u(0) = y(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,
    // see chapter 6
    tridag(a, b, c, y, u, n+1);
    // boundary conditions
    u(0) = 0;
    u(n) = 0;
    // replace previous time solution with new
    for (int i = 0; i <= n; i++) {
  y(i) = u(i);
    }
    // You may consider printing the solution at regular time intervals
    .... // print statements
  } // end time iteration
  ...
}
```

10.2.3 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

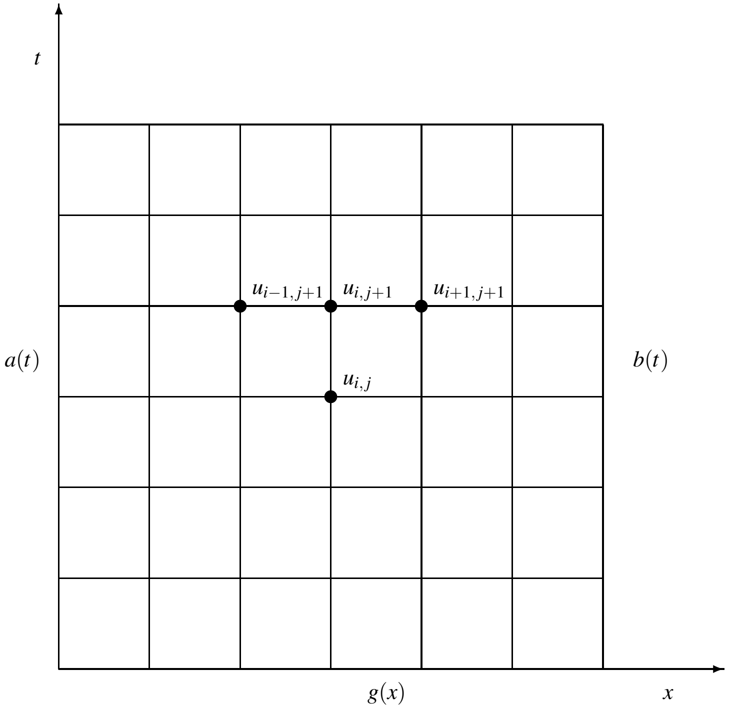


Fig. 10.2 Calculational molecule for the implicit scheme.

$$\frac{\theta}{\Delta x^2}\left(u_{i-1,j}-2u_{i,j}+u_{i+1,j}\right)+\frac{1-\theta}{\Delta x^2}\left(u_{i+1,j-1}-2u_{i,j-1}+u_{i-1,j-1}\right)=\frac{1}{\Delta t}\left(u_{i,j}-u_{i,j-1}\right), \tag{10.9}$$

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.

To derive the Crank-Nicolson equation, we start with the forward Euler scheme and Taylor expand $u(x,t+\Delta t)$, $u(x+\Delta x,t)$ and $u(x-\Delta x,t)$

$$\begin{aligned}
u(x+\Delta x,t) &= u(x,t)+\tfrac{\partial u(x,t)}{\partial x}\Delta x+\tfrac{\partial^2 u(x,t)}{2\partial x^2}\Delta x^2+\mathscr{O}(\Delta x^3), \qquad (10.10)\\
u(x-\Delta x,t) &= u(x,t)-\tfrac{\partial u(x,t)}{\partial x}\Delta x+\tfrac{\partial^2 u(x,t)}{2\partial x^2}\Delta x^2+\mathscr{O}(\Delta x^3),\\
u(x,t+\Delta t) &= u(x,t)+\tfrac{\partial u(x,t)}{\partial t}\Delta t+\mathscr{O}(\Delta t^2).
\end{aligned}$$

With these Taylor expansions the approximations for the derivatives takes the form

$$\begin{aligned}
\left[\tfrac{\partial u(x,t)}{\partial t}\right]_{\text{approx}} &= \tfrac{\partial u(x,t)}{\partial t}+\mathscr{O}(\Delta t), \qquad (10.11)\\
\left[\tfrac{\partial^2 u(x,t)}{\partial x^2}\right]_{\text{approx}} &= \tfrac{\partial^2 u(x,t)}{\partial x^2}+\mathscr{O}(\Delta x^2).
\end{aligned}$$

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 $t' = t+\Delta t/2$.

$$\begin{aligned}
u(x+\Delta x,t+\Delta t) &= u(x,t')+\tfrac{\partial u(x,t')}{\partial x}\Delta x+\tfrac{\partial u(x,t')}{\partial t}\tfrac{\Delta t}{2}+\tfrac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2+\tfrac{\partial^2 u(x,t')}{2\partial t^2}\tfrac{\Delta t^2}{4}+\\
&\qquad \tfrac{\partial^2 u(x,t')}{\partial x\partial t}\tfrac{\Delta t}{2}\Delta x+\mathscr{O}(\Delta t^3)\\
u(x-\Delta x,t+\Delta t) &= u(x,t')-\tfrac{\partial u(x,t')}{\partial x}\Delta x+\tfrac{\partial u(x,t')}{\partial t}\tfrac{\Delta t}{2}+\tfrac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2+\tfrac{\partial^2 u(x,t')}{2\partial t^2}\tfrac{\Delta t^2}{4}-\\
&\qquad \tfrac{\partial^2 u(x,t')}{\partial x\partial t}\tfrac{\Delta t}{2}\Delta x+\mathscr{O}(\Delta t^3)\\
u(x+\Delta x,t) &= u(x,t')+\tfrac{\partial u(x,t')}{\partial x}\Delta x-\tfrac{\partial u(x,t')}{\partial t}\tfrac{\Delta t}{2}+\tfrac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2+\tfrac{\partial^2 u(x,t')}{2\partial t^2}\tfrac{\Delta t^2}{4}-\\
&\qquad \tfrac{\partial^2 u(x,t')}{\partial x\partial t}\tfrac{\Delta t}{2}\Delta x+\mathscr{O}(\Delta t^3)\\
u(x-\Delta x,t) &= u(x,t')-\tfrac{\partial u(x,t')}{\partial x}\Delta x-\tfrac{\partial u(x,t')}{\partial t}\tfrac{\Delta t}{2}+\tfrac{\partial^2 u(x,t')}{2\partial x^2}\Delta x^2+\tfrac{\partial^2 u(x,t')}{2\partial t^2}\tfrac{\Delta t^2}{4}+\\
&\qquad \tfrac{\partial^2 u(x,t')}{\partial x\partial t}\tfrac{\Delta t}{2}\Delta x+\mathscr{O}(\Delta t^3)\\
u(x,t+\Delta t) &= u(x,t')+\tfrac{\partial u(x,t')}{\partial t}\tfrac{\Delta_t}{2}+\tfrac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2+\mathscr{O}(\Delta t^3)\\
u(x,t) &= u(x,t')-\tfrac{\partial u(x,t')}{\partial t}\tfrac{\Delta t}{2}+\tfrac{\partial^2 u(x,t')}{2\partial t^2}\Delta t^2+\mathscr{O}(\Delta t^3)
\end{aligned}$$

We now insert these expansions in the approximations for the derivatives to find

$$\begin{aligned}
\left[\tfrac{\partial u(x,t')}{\partial t}\right]_{\text{approx}} &= \tfrac{\partial u(x,t')}{\partial t}+\mathscr{O}(\Delta t^2), \qquad (10.12)\\
\left[\tfrac{\partial^2 u(x,t')}{\partial x^2}\right]_{\text{approx}} &= \tfrac{\partial^2 u(x,t')}{\partial x^2}+\mathscr{O}(\Delta x^2).
\end{aligned}$$

Bringing all these equations together results in Eq. (10.9) with $\theta = 1/2$.

The following table summarizes the three methods.

Using our previous definition of $\alpha = \Delta t/\Delta x^2$ we can rewrite Eq. (10.9) as

Scheme:	*Truncation Error:*	*Stability requirements:*
Crank-Nicolson	$\mathscr{O}(\Delta x^2)$ and $\mathscr{O}(\Delta t^2)$	Stable for all Δt and Δx.
Backward Euler	$\mathscr{O}(\Delta x^2)$ and $\mathscr{O}(\Delta t)$	Stable for all Δt and Δx.
Forward Euler	$\mathscr{O}(\Delta x^2)$ and $\mathscr{O}(\Delta t)$	$\Delta t \leq \frac{1}{2}\Delta x^2$

Table 10.1 Comparison of the different schemes.

$$-\alpha u_{i-1,j} + (2+2\alpha)u_{i,j} - \alpha u_{i+1,j} = \alpha u_{i-1,j-1} + (2-2\alpha)u_{i,j-1} + \alpha u_{i+1,j-1},$$

or in matrix-vector form as

$$\left(2\hat{I} + \alpha\hat{B}\right)V_j = \left(2\hat{I} - \alpha\hat{B}\right)V_{j-1},$$

where the vector V_j is the same as defined in the implicit case while the matrix $\hat{B}$ is

$$\hat{B} = \begin{bmatrix} 2 & -1 & 0 & 0 & \dots \\ -1 & 2 & -1 & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & -1 \\ 0 & 0 & \dots & -1 & 2 \end{bmatrix}.$$

We can rewrite the Crank-Nicolson scheme as follows

$$V_j = \left(2\hat{I} + \alpha\hat{B}\right)^{-1}\left(2\hat{I} - \alpha\hat{B}\right)V_{j-1}.$$

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

$$\rho\left(\left(2\hat{I} + \alpha\hat{B}\right)^{-1}\left(2\hat{I} - \alpha\hat{B}\right)\right) < 1,$$

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. 10.3.

10.2.3.1 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 n, int tsteps, double delta_x, double alpha)
{
  double a, b, c;
  vec u(n+1); // This is u in Au = r
  vec r(n+1); // Right side of matrix equation Au=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 < n; i++) {
  r(i) = alpha*u(i-1) + (2 - 2*alpha)*u(i) + alpha*u(i+1);
    }
    r(0) = 0;
    r(n) = 0;
    // Then solve the tridiagonal matrix
    tridag(a, b, c, r, u, xsteps+1);
    u(0) = 0;
    u(n) = 0;
    // Eventual print statements etc
    ....
}
```

10.2.4 Solution for the One-dimensional Diffusion Equation

It cannot be repeated enough, it is always useful to find cases where one can compare the numerical results and the developed algorithms and codes with closed-form solutions. The above case is also particularly simple. We have the following partial differential equation

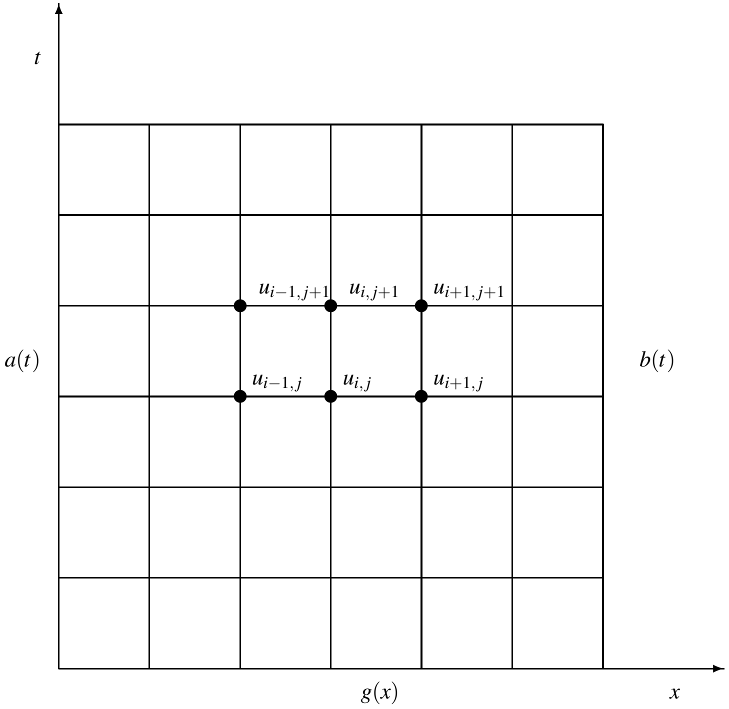


Fig. 10.3 Calculational molecule for the Crank-Nicolson scheme.

$$\nabla^2 u(x,t) = \frac{\partial u(x,t)}{\partial t},$$

with initial conditions

$$u(x,0) = g(x) \qquad 0 < x < L.$$

The boundary conditions are

$$u(0,t) = 0 \qquad t \geq 0, \qquad\qquad u(L,t) = 0 \qquad t \geq 0,$$

We assume that we have solutions of the form (separation of variable)

$$u(x,t) = F(x)G(t).$$

which inserted in the partial differential equation results in

$$\frac{F''}{F} = \frac{G'}{G},$$

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,

$$F'' + \lambda^2 F = 0; \qquad\qquad G' = -\lambda^2 G,$$

with general solutions

$$F(x) = A\sin(\lambda x) + B\cos(\lambda x); \qquad\qquad G(t) = Ce^{-\lambda^2 t}.$$

To satisfy the boundary conditions we require $B = 0$ and $\lambda = n\pi/L$. One solution is therefore found to be

$$u(x,t) = A_n \sin(n\pi x/L)e^{-n^2\pi^2 t/L^2}.$$

But there are 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

$$u(x,t) = \sum_{n=1}^{\infty} A_n \sin(n\pi x/L)e^{-n^2\pi^2 t/L^2}.$$

The coefficient A_n is in turn determined from the initial condition. We require

$$u(x,0) = g(x) = \sum_{n=1}^{\infty} A_n \sin(n\pi x/L).$$

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)

$$A_n = \frac{2}{L}\int_0^L g(x)\sin(n\pi x/L)\mathrm{d}x.$$

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. [52].

10.2.5 Explict scheme for the diffusion equation in two dimensions

We end this section by setting up an explicit scheme for the diffusion equation in two spatial coordinates. Here we assume that we are dealing with dimensionless quantities. The implict scheme is discussed in section 10.3. The $2+1$-dimensional diffusion equation, with the diffusion constant $D=1$, is given by

$$\frac{\partial u}{\partial t} = \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right),$$

where we have $u = u(x,y,t)$. We assume that we have a square lattice of length L with equally many mesh points in the x and y directions.

We discretize again position and time using now

$$u_{xx} \approx \frac{u(x+h,y,t) - 2u(x,y,t) + u(x-h,y,t)}{h^2},$$

which we rewrite as, in its discretized version,

$$u_{xx} \approx \frac{u^l_{i+1,j} - 2u^l_{i,j} + u^l_{i-1,j}}{h^2},$$

where $x_i = x_0 + ih$, $y_j = y_0 + jh$ and $t_l = t_0 + l\Delta t$, with $h = L/(n+1)$ and Δt the time step. We have defined our domain to start $x(y) = 0$ and end at $X(y) = L$. The second derivative with respect to y reads

$$u_{yy} \approx \frac{u^l_{i,j+1} - 2u^l_{i,j} + u^l_{i,j-1}}{h^2}.$$

We use again the so-called forward-going Euler formula for the first derivative in time. In its discretized form we have

$$u_t \approx \frac{u^{l+1}_{i,j} - u^l_{i,j}}{\Delta t},$$

resulting in

$$u^{l+1}_{i,j} = u^l_{i,j} + \alpha\left[u^l_{i+1,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1} - 4u^l_{i,j}\right],$$

where the left hand side, with the solution at the new time step, is the only unknown term, since starting with $t = t_0$, the right hand side is entirely determined by the boundary and initial conditions. We have $\alpha = \Delta t/h^2$. This scheme can be implemented using essentially the same approach as we used in Eq. (10.7). To find the constraints on Δt and h is left as an exercise.

10.3 Laplace’s and Poisson’s Equations

Laplace’s equation reads

$$\nabla^2 u(\mathbf{x}) = u_{xx} + u_{yy} = 0.$$

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 to be rectangular or following polar coordinates r,θ as well. Here we choose equal steps lengths in the x and the y directions. We set

$$h = \Delta x = \Delta y = \frac{L}{n+1},$$

where L is the length of the sides and we have $n+1$ points in both directions.

The discretized version reads

$$u_{xx} \approx \frac{u(x+h,y) - 2u(x,y) + u(x-h,y)}{h^2},$$

and

$$u_{yy} \approx \frac{u(x,y+h) - 2u(x,y) + u(x,y-h)}{h^2},$$

which we rewrite as

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2},$$

and

$$u_{yy} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2},$$

which gives when inserted in Laplace's equation

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right]. \qquad (10.13)$$

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

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right] + \frac{h^2}{4}\rho_{i,j}. \qquad (10.14)$$

The boundary condtions read

$$u_{i,0} = g_{i,0} \qquad 0 \le i \le n+1,$$

$$u_{i,L} = g_{i,0} \qquad 0 \le i \le n+1,$$

$$u_{0,j} = g_{0,j} \qquad 0 \le j \le n+1,$$

and

$$u_{L,j} = g_{L,j} \qquad 0 \le j \le n+1.$$

The calculational molecule for the Laplace operator of Eq. (10.13) is shown in Fig. 10.4.

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. [52] for proofs. However, solving these equations using for example the LU decomposition techniques discussed in chapter 6 becomes inefficient since the matrices are sparse. The relaxation techniques discussed below are more efficient.

10.3.1 Scheme for solving Laplace's (Poisson's) equation

We rewrite Eq. (10.14

$$4u_{i,j} = \left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right] - h^2\rho_{i,j} = \Delta_{ij} - \tilde{\rho}_{ij}, \qquad (10.15)$$

where we have defined

$$\Delta_{ij} = \left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right],$$

and

$$\tilde{\rho}_{ij} = h^2\rho_{i,j}.$$

In order to illustrate how we can transform the last equations into a linear algebra problem of the type $\mathbf{Ax} = \mathbf{w}$, with $\mathbf{A}$ a matrix and $\mathbf{x}$ and $\mathbf{w}$ unknown and known vectors respectively, let us also for the sake of simplicity assume that the number of points $n = 3$. We assume also that $u(x,y) = g(x,y)$ on the border $\delta\Omega$. Our calculational molecule becomes then We can now spell out the four equations which define the four unknown values of the function arising from the four inner points, using the labeling of Fig. 10.5.

The inner values of the function u are then given by

$$\begin{aligned}
4u_{11} - u_{21} - u_{01} - u_{12} - u_{10} &= -\tilde{\rho}_{11} \\
4u_{12} - u_{02} - u_{22} - u_{13} - u_{11} &= -\tilde{\rho}_{12} \\
4u_{21} - u_{11} - u_{31} - u_{22} - u_{20} &= -\tilde{\rho}_{21} \\
4u_{22} - u_{12} - u_{32} - u_{23} - u_{21} &= -\tilde{\rho}_{22}.
\end{aligned}$$

If we isolate on the left-hand side the unknown quantities u_{11}, u_{12}, u_{21} and u_{22}, that is the inner points not constrained by the boundary conditions, we can rewrite the above equations as a matrix $\mathbf{A}$ times an unknown vector $\mathbf{x}$, that is

$$\mathbf{Ax} = \mathbf{b},$$

or in more detail

$$\begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \begin{bmatrix} u_{11} \\ u_{12} \\ u_{21} \\ u_{22} \end{bmatrix} = \begin{bmatrix} u_{01} + u_{10} - \tilde{\rho}_{11} \\ u_{13} + u_{02} - \tilde{\rho}_{12} \\ u_{31} + u_{20} - \tilde{\rho}_{21} \\ u_{32} + u_{23} - \tilde{\rho}_{22} \end{bmatrix}.$$

The right hand side is constrained by the values at the boundary plus the known function $\tilde{\rho}$. For a two-dimensional equation it is easy to convince oneself that for larger sets of mesh points, we will not have more than five function values for every row of the above matrix. For a problem with $n+1$ mesh points, our matrix $\mathbf{A} \in \mathbb{R}^{(n+1)\times(n+1)}$ leads to $(n-1)\times(n-1)$ unknown

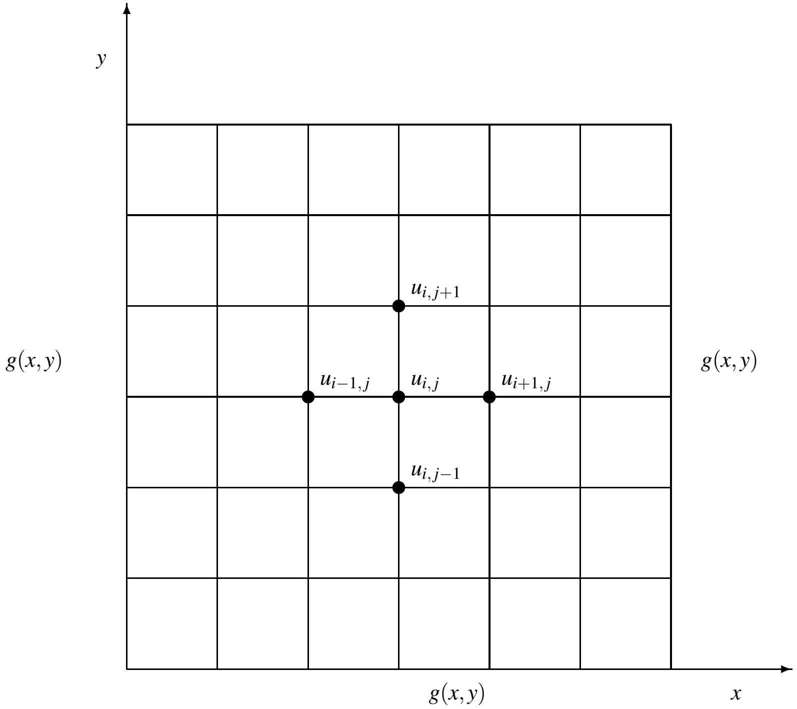


Fig. 10.4 Five-point calculational molecule for the Laplace operator of Eq. (10.13). The border $\delta\Omega$ defines the boundary condition $u(x,y) = g(x,y)$.

function values u_{ij}. This means that, if we fix the endpoints for the two-dimensional case (with a square lattice) at $i(j)=0$ and $i(j)=n+1$, we have to solve the equations for $1 \geq i(j) len$.

Since the matrix is rather sparse but is not on a tridiagonal form, elimination methods like the LU decomposition discussed in chapter 6, are not very practical. Rather, iterative schemes like Jacobi's method or the Gauss-Seidel method discussed in the same chapter, are preferred. The above matrix is also always diagonally dominant, a necessary condition for these iterative solvers to converge.

In setting up for example Jacobi's method, it is useful to rewrite the matrix **A** as

$$\mathbf{A} = \mathbf{D} + \mathbf{U} + \mathbf{L},$$

with **D** being a diagonal matrix with 4 as the only value, Ṵ is an upper triangular matrix and **L** a lower triangular matrix. In our case we have

$$\mathbf{D} = \begin{bmatrix} 4\,0\,0\,0 \\ 0\,4\,0\,0 \\ 0\,0\,4\,0 \\ 0\,0\,0\,4 \end{bmatrix},$$

and

$$\mathbf{L} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

We assume now that we have an estimate for the unknown functions u_{11}, u_{12}, u_{21} and u_{22}. We will call this the zeroth value and label it as $u_{11}^{(0)}$, $u_{12}^{(0)}$, $u_{21}^{(0)}$ and $u_{22}^{(0)}$. We can then set up an iterative scheme where the next solution is defined in terms of the previous one as

$$\begin{aligned} u_{11}^{(1)} &= \tfrac{1}{4}(b_1 - u_{12}^{(0)} - u_{21}^{(0)}) \\ u_{12}^{(1)} &= \tfrac{1}{4}(b_2 - u_{11}^{(0)} - u_{22}^{(0)}) \\ u_{21}^{(1)} &= \tfrac{1}{4}(b_3 - u_{11}^{(0)} - u_{22}^{(0)}) \\ u_{22}^{(1)} &= \tfrac{1}{4}(b_4 - u_{12}^{(0)} - u_{21}^{(0)}), \end{aligned}$$

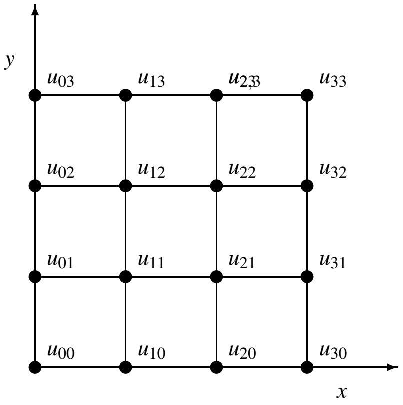


Fig. 10.5 Explicit molecule for $n=3$. The border $\delta\Omega$ defines the boundary condition $u(x,y)=g(x,y)$.

where we have defined the vector

$$\mathbf{b} = \begin{bmatrix} u_{01} + u_{10} - \tilde{\rho}_{11} \\ u_{13} + u_{02} - \tilde{\rho}_{12} \\ u_{31} + u_{20} - \tilde{\rho}_{21} \\ u_{32} + u_{23} - \tilde{\rho}_{22} \end{bmatrix}.$$

We can rewrite the equations in a more compact form in terms of the matrices **D**, **L** and **U** as, after $r+1$ iterations,

$$\mathbf{x}^{(r+1)} = \mathbf{D}^{-1}\left(\mathbf{b} - (\mathbf{L}+\mathbf{U})\mathbf{x}^{(r)}\right), \tag{10.16}$$

where the unknown functions are now defined in terms of

$$\mathbf{x} = \begin{bmatrix} u_{11} \\ u_{12} \\ u_{21} \\ u_{22} \end{bmatrix}.$$

If we wish to implement Gauss-Seidel's algorithm, see our discussion in chapter 6, the set of equations to solve are then given by

$$\mathbf{x}^{(r+1)} = -(\mathbf{D}+\mathbf{L})^{-1}\left(\mathbf{b} - \mathbf{U}\mathbf{x}^{(r)}\right), \tag{10.17}$$

or alternatively as

$$\mathbf{x}^{(r+1)} = \mathbf{D}^{-1}\left(\mathbf{b} - \mathbf{L}\mathbf{x}^{(r+1)} - \mathbf{U}\mathbf{x}^{(r)}\right).$$

In the next subsection we discuss an actual implementation of the Jacobi algorithm.

10.3.2 Jacobi Algorithm for solving Laplace's Equation

It is thus fairly straightforward to extend this equation to the three-dimensional case. Whether we solve Eq. (10.13) or Eq. (10.14), 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. (10.13) 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 based on the Jacobi method or the Gauss-Seidel method or the relaxation methods discussed in chapter 6.

Implementing Jacobi's method is 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. (10.13) or Eq. (10.14) in order to obtain a new solution $u_{i,j}^{(1)}$. Most likely this solution will not be a solution to Eq. (10.13). 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. (10.13) 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 to the next step.

4. Update the new value of u for the given iteration
5. Go to step 2

A simple example may help in understanding this method. We consider a condensator with parallel plates separated at a distance L resulting in for example the voltage differences $u(x,0) = 200sin(2\pi x/L)$ and $u(x,1) = -200sin(2\pi x/L)$. 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 C++ program which solves iteratively Eq. (10.13) using Jacobi's method. Only the part which computes Eq. (10.13) is included here.

```
....
// We define the step size for a square lattice with n+1 points
     double h = (xmax-xmin)/(n+1);
     double L = xmax-xmin; // The length of the lattice
// We allocate space for the vector u and the temporary vector to
// be upgraded in every iteration
     mat u( n+1, n+1); // using Armadillo to define matrices
     mat u_temp( n+1, n+1); // This is the temporary value
     u = 0. // This is also our initial guess for all unknown values
// We need to set up the boundary conditions. Specify for various cases
   .....
// The iteration algorithm starts here
     iterations = 0;
     while( (iterations <= max_iter) && ( diff > 0.00001) ){
       u_temp = u; diff = 0.;
       for (j = 1; j<= n,j++){
          for(l = 1; l <= n; l++){
            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 += fabs(u_temp(i,j)-u(i,j));
          }
       }
       iterations++;
       diff /= pow((n),2.0);
     }  // end while loop
```

The important part of the algorithm is applied in the function which sets up the two-dimensional Laplace equation. There we have a 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 a given maximum. We need also to provide a convergence tolerance. In the above program example we have fixed this to be 0.00001. Depending on the type of applications one may have to change both the number of maximum iterations and the tolerance.

While the Jacobi iteration scheme is very simple and parallelizable, it has a slow convergence rate, which often renders it impractical for any "real world" applications. However, the algorithm is easy to parallelize.

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].

10.3.3 Jacobi's algorithm extended to the diffusion equation in two dimensions

In our previous section we discussed the extension of the diffusion equation to two dimensions, using an explicit scheme. Let us know implememt the implicit scheme and show how we can extend the previous algorithm for solving Laplace's or Poisson's equations to the diffusion equation as well. As the reader will notice, this simply implies a slight redefinition of the vector **b** defined in Eq. (10.16).

To see this, let us first set up the diffusion in two spatial dimensions, with boundary and initial conditions. The $2+1$-dimensional diffusion equation (with dimensionless variables) reads for a function $u = u(x,y,t)$

$$\frac{\partial u}{\partial t} = D\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right).$$

We assume that we have a square lattice of length L with equally many mesh points in the x and y directions. Setting the diffusion constant $D = 1$ and using the shorthand notation $u_{xx} = \partial^2 u/\partial x^2$ etc for the second derivatives and $u_t = \partial u/\partial t$ for the time derivative, we have, with a given set of boundary and initial conditions,

$$\begin{array}{rr} u_t = u_{xx} + u_{yy} & x \in (0,L), t > 0 \\ u(x,0) = g(x) & x \in (0,L) \\ u(0,y,t) = u(L,y,t) = u(x,0,t) = u(x,L,t)0 & t > 0 \end{array}$$

We discretize again position and time, and use the following approximation for the second derivatives

$$u_{xx} \approx \frac{u(x+h,y,t) - 2u(x,y,t) + u(x-h,y,t)}{h^2},$$

which we rewrite as, in its discretized version,

$$u_{xx} \approx \frac{u^l_{i+1,j} - 2u^l_{i,j} + u^l_{i-1,j}}{h^2},$$

where $x_i = x_0 + ih$, $y_j = y_0 + jh$ and $t_l = t_0 + l\Delta t$, with $h = L/(n+1)$ and Δt the time step. The second derivative with respect to y reads

$$u_{yy} \approx \frac{u^l_{i,j+1} - 2u^l_{i,j} + u^l_{i,j-1}}{h^2}.$$

We use now the so-called backward going Euler formula for the first derivative in time. In its discretized form we have

$$u_t \approx \frac{u^l_{i,j} - u^{l-1}_{i,j}}{\Delta t},$$

resulting in

$$u^l_{i,j} + 4\alpha u^l_{i,j} - \alpha\left[u^l_{i+1,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1}\right] = u^{l-1}_{i,j},$$

where the right hand side is the only known term, since starting with $t = t_0$, the right hand side is entirely determined by the boundary and initial conditions. We have $\alpha = \Delta t/h^2$. For future time steps, only the boundary values are determined and we need to solve the equations for the interior part in an iterative way similar to what was done for Laplace's or Poisson's equations. To see this, we rewrite the previous equation as

$$u^l_{i,j} = \frac{1}{1+4\alpha}\left[\alpha(u^l_{i+1,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1}) + u^{l-1}_{i,j}\right],$$

or in a more compact form as

$$u^l_{i,j} = \frac{1}{1+4\alpha}\left[\alpha\Delta^l_{ij} + u^{l-1}_{i,j}\right], \tag{10.18}$$

with $\Delta^l_{ij} = \left[u^l_{i,j+1} + u^l_{i,j-1} + u^l_{i+1,j} + u^l_{i-1,j}\right]$. This equation has essentially the same structure as Eq. (10.15), except that the function ρ_{ij} is replaced by the solution at a previous time step $l-1$. Furthermore, the diagonal matrix elements are now given by $1+4\alpha$, while the non-zero non-diagonal matrix elements equal α. This matrix is also positive definite, meaning in turn that iterative schemes like the Jacobi or the Gauss-Seidel methods will converge to the desired solution after a given number of iterations.

We leave it as an exercise to implement the Jacobi and Gauss-Seidel algorithms for Eq. (10.18).

10.4 Wave Equation in two Dimensions

The $1+1$-dimensional wave equation reads

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial t^2},$$

with $u = u(x,t)$ and we have assumed that we operate with dimensionless variables. Possible boundary and initial conditions with $L=1$ are

$$\begin{array}{cc} u_{xx} = u_{tt} & x \in (0,1), t > 0 \\ u(x,0) = g(x) & x \in (0,1) \\ u(0,t) = u(1,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = 0 & x \in (0,1) \end{array}.$$

We discretize again time and position,

$$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2},$$

and

$$u_{tt} \approx \frac{u(x,t+\Delta t) - 2u(x,t) + u(x,t-\Delta t)}{\Delta t^2},$$

which we rewrite as

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2},$$

and

$$u_{tt} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta t^2},$$

resulting in

$$u_{i,j+1} = 2u_{i,j} - u_{i,j-1} + \frac{\Delta t^2}{\Delta x^2}\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right). \tag{10.19}$$

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

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j - \Delta t)}{2\Delta t},$$

and at $t = 0$ it reduces to

$$u_t \approx \frac{u_{i,+1} - u_{i,-1}}{2\Delta t} = 0,$$

implying that $u_{i,+1} = u_{i,-1}$. If we insert this condition in Eq. (10.19) we arrive at a special formula for the first time step

$$u_{i,1} = u_{i,0} + \frac{\Delta t^2}{2\Delta x^2}\left(u_{i+1,0} - 2u_{i,0} + u_{i-1,0}\right). \qquad (10.20)$$

We need seemingly two different equations, one for the first time step given by Eq. (10.20) and one for all other time-steps given by Eq. (10.19). However, it suffices to use Eq. (10.19) for all times as long as we provide $u(i,-1)$ using

$$u_{i,-1} = u_{i,0} + \frac{\Delta t^2}{2\Delta x^2}\left(u_{i+1,0} - 2u_{i,0} + u_{i-1,0}\right),$$

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

$$\begin{array}{ll} t_l = l\Delta t & l \geq 0 \\ x_i = i\Delta x & 0 \leq i \leq n_x \\ y_j = j\Delta y & 0 \leq j \leq n_y \end{array},$$

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

$$\begin{array}{cc} u_{xx} + u_{yy} = u_{tt} & x,y \in (0,1), t > 0 \\ u(x,y,0) = g(x,y) & x,y \in (0,1) \\ u(0,0,t) = u(1,1,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = 0 & x,y \in (0,1) \end{array}.$$

We have now the following discretized partial derivatives

$$u_{xx} \approx \frac{u^l_{i+1,j} - 2u^l_{i,j} + u^l_{i-1,j}}{h^2},$$

and

$$u_{yy} \approx \frac{u^l_{i,j+1} - 2u^l_{i,j} + u^l_{i,j-1}}{h^2},$$

and

$$u_{tt} \approx \frac{u^{l+1}_{i,j} - 2u^l_{i,j} + u^{l-1}_{i,j}}{\Delta t^2},$$

which we merge into the discretized $2+1$-dimensional wave equation as

$$u^{l+1}_{i,j} = 2u^l_{i,j} - u^{l-1}_{i,j} + \frac{\Delta t^2}{h^2}\left(u^l_{i+1,j} - 4u^l_{i,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1}\right), \qquad (10.21)$$

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

$$u^{-1}_{i,j} = u^0_{i,j} + \frac{\Delta t}{2h^2}\left(u^0_{i+1,j} - 4u^0_{i,j} + u^0_{i-1,j} + u^0_{i,j+1} + u^0_{i,j-1}\right),$$

in our setup of the initial conditions.

10.4.1 Closed-form Solution

We develop here the closed-form solution for the $2+1$ dimensional wave equation with the following boundary and initial conditions

$$\begin{array}{cc} c^2(u_{xx}+u_{yy}) = u_{tt} & x,y \in (0,L), t>0 \\ u(x,y,0) = f(x,y) & x,y \in (0,L) \\ u(0,0,t) = u(L,L,t) = 0 & t>0 \\ \partial u/\partial t|_{t=0} = g(x,y) & x,y \in (0,L) \end{array}.$$

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

$$G_{mn}(t) = B_{mn}\cos(\lambda_{mn}t) + D_{mn}\sin(\lambda_{mn}t),$$

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 D_{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

$$B_{mn} = \frac{2}{L}\int_0^L \int_0^L dxdyf(x,y)\sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L}),$$

and

$$D_{mn} = \frac{2}{L}\int_0^L \int_0^L dxdyg(x,y)\sin(\frac{m\pi x}{L})\sin(\frac{n\pi y}{L}).$$

Inserting the particular functional forms of $f(x,y)$ and $g(x,y)$ one obtains the final closed-form expressions.

10.5 Exercises

10.1. Consider the two-dimensional wave equation for a vibrating membrane given by the following initial and boundary conditions

$$\begin{cases} u_{xx} + u_{yy} = u_{tt} & x,y \in (0,1), t > 0 \\ u(x,y,0) = sin(x)cos(y) & x,y \in (0,1) \\ u(0,0,t) = u(1,1,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = 0 & x,y \in (0,1) \end{cases}.$$

1. Find the closed-form solution for this equation using the technique of separation of variables.
2. Write down the algorithm for solving this equation and set up a program to solve the discretized wave equation. Compare your results with the closed-form solution. Use a quadratic grid.
3. 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.

4. Develop an algorithm for solving the new wave equation and write a program which implements it.

10.2. In this project will first study the simple two-dimensional wave equation and compare our numerical solution with closed-form results. Thereafter we introduce a simple model for a tsunami.

Consider first the two-dimensional wave equation for a vibrating square membrane given by the following initial and boundary conditions

$$\begin{cases} \lambda\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right) = \frac{\partial^2 u}{\partial t^2} & x,y \in [0,1], t \geq 0 \\ u(x,y,0) = sin(\pi x)sin(2\pi y) & x,y \in (0,1) \\ u = 0 \text{ boundary} & t \geq 0 \\ \partial u/\partial t|_{t=0} = 0 & x,y \in (0,1) \end{cases}.$$

The boundary is defined by $x = 0$, $x = 1$, $y = 0$ and $y = 1$.

- Find the closed-form solution for this equation using the technique of separation of variables.
- Write down the algorithm for the explicit method for solving this equation and set up a program to solve the discretized wave equation. Describe in particular how you treat the boundary conditions and initial conditions. Compare your results with the closed-form solution. Use a quadratic grid.

  Check your results as function of the number of mesh points and in particular against the stability condition

  $$\Delta t \leq \frac{1}{\sqrt{\lambda}}\left(\frac{1}{\Delta x^2}+\frac{1}{\Delta y^2}\right)^{-1/2}$$

  where Δt, Δx and Δy are the chosen step lengths. In our case $\Delta x = \Delta y$. It can be useful to make animations of the results.

  An example of a simple code which solves this problem using the explicit scheme is listed here.

```
int main ( int argc, char * argv[] )
{
  .....
  // Various declarations, not all are included
  .....
  // n is the number of mesh points for x and y (square lattice)
  // m is the number of integration points in time
  int n, m
  double tstep = (tfinal-tinitial)/m;
  double h = 1.0/(n+ 1.0);
  double alpha = tstep*tstep/(h*h)
  // We define the solution u at an explicit time step l
  // using Armadillo to define matrices
  mat u( n+1, n+1), u_last( n+1, n+1), u_next( n+1, n+1);
  // We declare also vectors that hold the position in the x and y directions
  vec x(n+1), y(n+1);
  for ( int i = 0; i < n+1 ; i++ ) {
   x(i) = i*h;
   y(i) = x(i);
  }
  u_last = 0.0;
  // initializing the function (using the initial conditions)
  for ( int i = 1; i < n; i++ ) { // setting initial step
   for ( int j = 1; j < n; j++ ) {
    u_last(i,j) = sin(PI*x(i))*sin(2*PI*y(j));
   }
  }
  u = 0.0; u_next =0.0; // includes also the boundary, set for all times
  for ( int i = 1; i < n; i++ ) { // setting first step using the initial derivative
   for ( int j = 1; j < n; j++ ) {
    u(i,j) = u_last(i,j) - alpha*0.5*
  (4*u_last(i,j) - u_last(i+1,j) - u_last(i-1,j) - u_last(i,j+1) - u_last(i,j-1));
   }

  // iterating in time
  double t = tinitial;
  while ( t < tfinal ) {
   t = t + tstep;
   for ( int i = 1; i < n; i++ ) { // computing next step
    for ( int j = 1; j < n ; j++ ) {
  u_next(i,j) = 2*u(i,j) - u_last(i,j) -
   alpha*(4*u(i,j) - u(i+1,j) - u(i-1,j) - u(i,j+1) - u(i,j-1));
    }
   }
   // Update then the new function value
   for ( int i = 1; i < n; i++ ) {
    for ( int j = 1; j < n; j++ ) {
  u_last(i,j) = u(i,j);
  u(i,j) = u_next(i,j);
    }
   }
  // One may consider to print to file the results after a given # of time steps
  .... print statements
  }
  return 0;
}
```

We modify now the wave equation in order to consider a $2+1$ dimensional wave equation with a position dependent velocity, given by

$$\frac{\partial^2 u}{\partial t^2} = \nabla\cdot(\lambda(x,y)\nabla u).$$

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. We will model λ as

$$\lambda = gH(x,y),$$

with g being the acceleration of gravity and $H(x,y)$ is the still water depth.

The function $H(x,y)$ 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 as discussed in point d) below. We have assumed that the vertical motion is negligible and that we deal with long wavelenghts $\tilde{\lambda}$ compared with the depth of the sea H, that is $\tilde{\lambda}/H \gg 1$. We will also neglect Coriolis effects.

You can discretize

$$\nabla \cdot (\lambda(x,y)\nabla u) = \frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right),$$

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

$$\frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right) \approx \frac{1}{\Delta y}\left(\lambda_{i,j+1/2}\left[\frac{u^l_{i,j+1} - u^l_{i,j}}{\Delta y}\right] - \lambda_{i,j-1/2}\left[\frac{u^l_{i,j} - u^l_{i,j-1}}{\Delta y}\right]\right).$$

- Show 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.

We assume that we can approximate the coastline with a quadratic grid. As boundary condition at the coastline we will employ

$$\frac{\partial u}{\partial n} = \nabla u \cdot \mathbf{n} = 0,$$

where $\partial u/\partial n$ is the derivative in the direction normal to the boundary.

We are going to model the impact of an earthquake on sea water. This is normally modelled via an elevation of the sea bottom. We will assume that the movement of the sea bottom is very rapid compared with the period of the propagating waves. This means that we can approximate the bottom elevation with an initial surface elevation. The initial conditions are then given by (with L the length of the grid)

$$u(x,y,0) = f(x,y) \qquad x,y \in (0,L),$$

and

$$\partial u/\partial t|_{t=0} = 0 \qquad x,y \in (0,L).$$

We will approximate the initial elevation with the function

$$f(x,y) = A_0 \exp\left(-\left[\frac{x-x_c}{\sigma_x}\right]^2 - \left[\frac{y-y_c}{\sigma_y}\right]^2\right),$$

where A_0 is the elevation of the surface and is typically $1-2$ m. The variables σ_x and σ_y represent the extensions of the surface elevation. In this project we will let $\sigma_x = 80$ km and $\sigma_y = 200$ km. The 2004 tsunami had extensions of approximately 200 and 1000 km, respectively.

The variables x_c and y_c represent the epicentre of the earthquake.

We need also to model the sea bottom and the function $\lambda(x,y) = gH(x,y)$. We assume that we can model the sea bottom with a water depth of 5000 m and a surface elevation of 2 m. The sea bottom towards one of the coastlines has a shape with an inclination of $\theta = 1$ degree and depth where the earthquake takes place of 5000 m. This gives the following model for $\lambda(x,y) = gH(x,y) = gH(x)$ with $H_0 = 5000$ m

```
for ( int i = 0; i < (2*n+1); i++ ) {
 if ( (i-1)*(h/2.0) < X_0 ) {
  lambda[i] = G*H_0; // lambda depends only on x
 } else {
  lambda[i] = G*(H_0 - ((i-1)*(h/2.0)-X_0)*0.0174550649282176);
 }
}
```

Here X_0 is the point where the sea bed changes (with respect to shore). Your tasks are as follows:

- Develop an algorithm for solving the new wave equation and write a program which implements it. Pay in particular attention to the implementation of the boundary conditions and the initial conditions. Figure out how to deal with the fictitious values in time and space for the discretized functions. You need also to find the functional form of $H(x,y) = H(x)$.
  Be careful to scale the equations properly. With the depth of 5000 m, extensions $\sigma_x = 80$ km and $\sigma_y = 200$ km you need to figure out the proper dimensions of the grid $L \times L$. Scale the equations so that you can use dimensionless quantities.
  With the above parameters, initial values and boundary conditions, study the temporal evolution of the wave towards the coastline. Comment your results. It can be useful to make animations of the results (a simple recipe with gnuplot and python for this is available under the project link for project 4 at the webpage).
  It also important that you keep in mind the stability condition

$$\Delta t \leq \frac{1}{\sqrt{\max \lambda(x,y)}} \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} \right)^{-1/2}$$

- We keep now the same shape of the sea bottom and the same parameters as in d), but we shift the center of the earthquake to the right with 40 km. Which one of the two earthquakes will produce the largest impact (wave elevation) at the coastline? Comment your results.

10.3. Consider a condensator with parallel plates separated at a distance L resulting in the voltage differences $u(x,0) = 100 sin(2\pi x/L)$ and $u(x,1) = -100 sin(2\pi x/L)$. These are our boundary conditions. Write a program which obtains the voltage u between the plates using both the Jacobi method and the Gauss-Seidel method. Parallelize your program as detailed in chapter 6 and study the stability of your solutions as functions of the number of mesh points. How does your parallel code scale?

10.4. The dominant way of transporting signals between neurons (nerve cells) in the brain is by means of diffusion of particular signal molecules called *neurotransmitters* across the synaptic cleft separating the cell membranes of the two cells. A drawing of a synapse is given in Fig. 10.6.

Following the arrival of an action potential in the axon terminal a process is initiated in which (i) vesicles inside the axon terminal (filled with neurotransmitter molecules) merge with the presynaptic (axon) membrane and (ii) release neurotransmitters into the synaptic cleft. These neurotransmitters diffuse across the synaptic cleft to receptors on the postsynaptic side which “receives” the signal. A schematic illustration of this process is shown in Fig. 10.7(left). Since the transport process in the synaptic cleft is governed by diffusion, we can describe it mathematically by

$$\frac{\partial u}{\partial t} = D\nabla^2 u, \tag{10.22}$$

where uăis the concentration of the particular neurotransmitter, and D is the diffusion coefficient of the neurotransmitter in this particular environment (solvent in synaptic cleft).

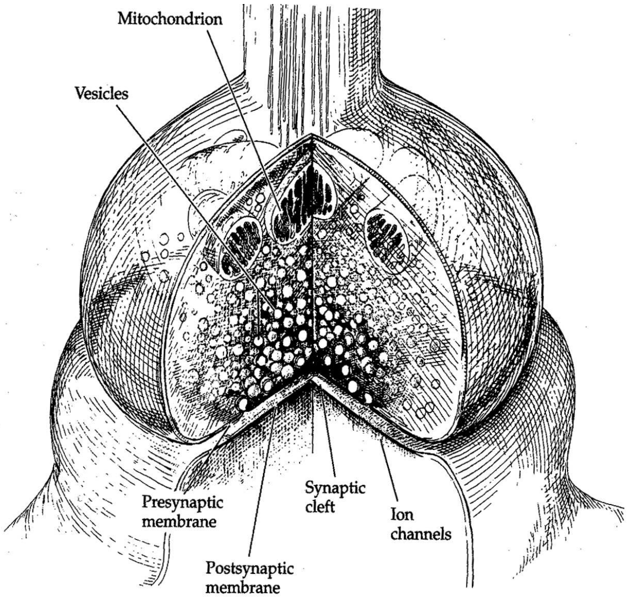


Fig. 10.6 Drawing of a synapse. The axon terminal is the knoblike structure and the spine of the receiving neuron is the bottom one. The synaptic cleft is the small space between the presynaptic (axon) and postsynaptic (dendritic spine) membrane. (From Thompson: "The Brain", Worth Publ., 2000)

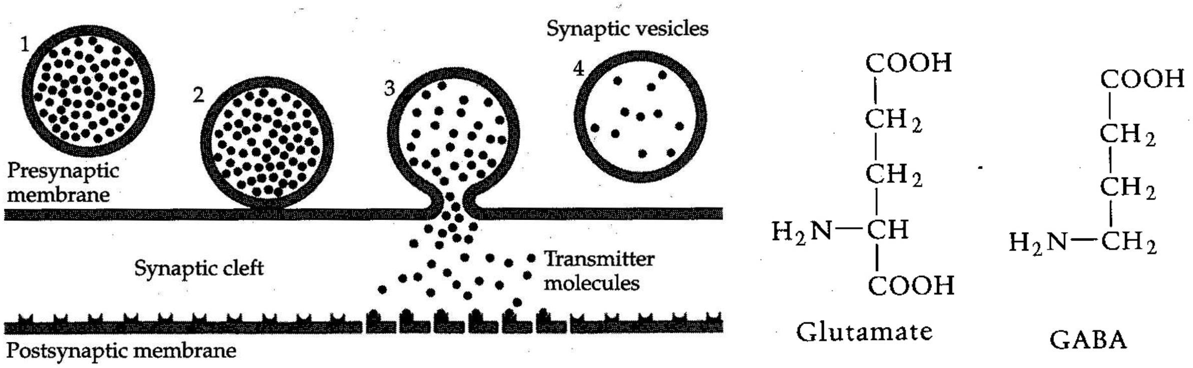


Fig. 10.7 Left: Schematic drawing of the process of vesicle release from the axon terminal and release of transmitter molecules into the synaptic cleft. (From Thompson: "The Brain", Worth Publ., 2000). Right: Molecular structure of the two important neurotransmitters *glutamate* and *GABA*.

If we assume (i) that the neurotransmitter is released roughly equally on the "presynaptic" side of the synaptic cleft, and (ii) that the synaptic cleft is roughly equally wide across the whole synaptic terminal, we can, given the large area of the synaptic cleft compared to its width, assume that the neurotransmitter concentration only varies in the direction across the synaptic cleft (from presynaptic to postsynaptic side). We choose this direction to be the x-direction (see Fig. 10.8). In this case $u(\mathbf{r}) = u(x)$, the diffusion equation reduces to

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}. \tag{10.23}$$

Immediately after the release of a neurotransmitter into the synaptic cleft ($t = 0$) the concentration profile in the x-direction is given by

$$u(x,t=0) = N\,\delta(x), \tag{10.24}$$

where N is the number of particle released into the synaptic cleft per area of membrane.

To get an idea over the time-dependence of the neurotransmitter concentration at the postsynaptic side ($x = d$), we can look at the solution of a “free” random walk (i.e., no obstacles or particle absorbers in either direction). The solution of Eq. (10.23) with the initial condition in Eq. (10.24) is given by (see Nelson: *Biological Physics*, p. 143 or Lectures notes chapter 12.3)

$$u(x,t) = \frac{N}{\sqrt{4\pi Dt}} e^{-x^2/4Dt} \quad . \tag{10.25}$$

The concentration at the postsynaptic side $u(d,t)$ approaches 0 in the limit $t \to 0$ ăand $t \to \infty$.

The above assumption regarding the neurotransmitter molecules undergoing a “free” random walk, is obviously a simplification. In the true diffusion process in the synaptic cleft the neurotransmitter molecules will, for example, occasionally bump into the presynaptic membrane they came from. Also at the postsynaptic side the neurotransmitters are absorbed by receptors located on the postsynaptic cell membrane and are thus (temporally) removed from the solution.

To approach this situation in our mathematical model we can impose the following boundary and initial conditions with $x \in [0,d]$

$$u(x=0,t>0) = u_0, \;\; u(x=d, \text{all } t) = 0, \;\; u(0<x<d, t<0) = 0 \;\; . \tag{10.26}$$

Hereafter we set $d = 1$. This corresponds to that (i) for $t < 0$ there are no neurotransmitters in the synaptic cleft, (ii) for $t > 0$ the concentration of neurotransmitters at the presynaptic boundary of the synaptic cleft ($x = 0$) is kept *fixed*ă at $u = u_0 = 1$ in our case, and (iii) that the postsynaptic receptors immediately absorb nearby neurotransmitters so that $u = 0$ on the postsynaptic side of the cleft ($x = d = 1$).

The full solution of the diffusion equation with boundary/initial conditions in Eq. (10.26) can be found in a closed form. We will use this solution to test our numerical calculations.

We are thus looking at a one-dimensional problem

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, t > 0, x \in [0,d]$$

or

$$u_{xx} = u_t,$$

with initial conditions, i.e., the conditions at $t = 0$,

$$u(x,0) = 0 \qquad 0 < x < d$$

with $d = 1$ the length of the x-region of interest. The boundary conditions are

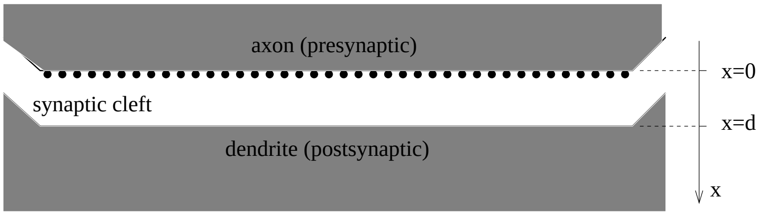


Fig. 10.8 Schematic drawing of the synaptic cleft in our model. The black dots represent neurotransmitter molecules, and the situation shown corresponds to the situation immediately after neurotransmitter release into the synaptic cleft.

$$u(0,t) = 1 \qquad t > 0,$$

and

$$u(d,t) = 0 \qquad t > 0.$$

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

   $$u_t \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t) - u(x_i,t_j)}{\Delta t}$$

   and

   $$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2},$$

   or

   $$u_{xx} \approx \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2}.$$

2. The implicit Backward Euler with

   $$u_t \approx \frac{u(x,t) - u(x,t-\Delta t)}{\Delta t} = \frac{u(x_i,t_j) - u(x_i,t_j-\Delta t)}{\Delta t}$$

   and

   $$u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2},$$

   or

   $$u_{xx} \approx \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2},$$

3. Finally we use the implicit Crank-Nicolson scheme with a time-centered scheme at $(x,t+\Delta t/2)$

   $$u_t \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t) - u(x_i,t_j)}{\Delta t}.$$

   The corresponding spatial second-order derivative reads

   $$u_{xx} \approx \frac{1}{2}\left(\frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2} + \right.$$

   $$\left. \frac{u(x_i+\Delta x,t_j+\Delta t) - 2u(x_i,t_j+\Delta t) + u(x_i-\Delta x,t_j+\Delta t)}{\Delta x^2}\right).$$

   Note well that we are using a time-centered scheme wih $t+\Delta t/2$ as center.

a) Find the closed form solution to this problem. You will need this in order to study the numerical accuracy of your results. To find the closed-form solution, we will need the stationary solution (steady-state solution). The solution to the steady-state problem is on the form $u(x) = Ax + b$. The solution for the steady-state case u_s that obeys the above boundary conditions is

   $$u_s(x) = 1 - x.$$

   You can use this solution to define a new function $v(x) = u(x) - u_s(x)$ with boundary conditions $v(0) = v(d) = 0$. The latter is easier to solve both numerically and on a closed form.

b) 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.

c) Find the truncation errors of these three schemes and investigate their stability properties.
d) Implement the three algorithms in the same code and perform tests of the solution for these three approaches for $\Delta x = 1/10$, $h = 1/100$ 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. Remember that for solving the tridiagonal equations you can use your code from project 1.
e) Compare the solutions at t_1 and t_2 with the closed form result for the continuous problem. Which of the schemes would you classify as the best?
f) The above problem can be solved using Monte Carlo methods and random walks. We follow here Farnell and Gibson in Journal of Computational Physics, volume **208**, pages 253-265 (2005). Choose a constant step length $l_0 = \sqrt{2D\Delta t}$ and equal probability for jumping left and right. Set up the algorithm for solving the above diffusion problem and write a code to do it. Compare your results with those from the partial differential equation solution and comment the results.
g) Change the above stepsize by using a Gaussian distribution with mean value 1 and standard deviation 0. The step length of the random walker is now $l_0 = \sqrt{2D\Delta t}\xi$, where ξ is random number chosen from the above Gaussian distribution. Implement this stepsize to the program from f) and compare the results and comment.

10.5. In this exercise the aim is to derive both an explicit and an implicit scheme for the two-dimensional diffusion equation.

Our differential equation is

$$\frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2} = \frac{\partial u(x,y,t)}{\partial t}, t > 0, x,y \in [0,1],$$

where we now have made a model with a square lattice for x and y. Here we will assume that the initial conditions are

$$u(x,y,0) = (1-y)\exp(x) \qquad 0 \le x,y \le 1.$$

The boundary conditions (so-called Dirichlet conditions) are

$$u(0,y,t) = (1-y)\exp(t) \qquad t \ge 0 \qquad 0 \le y \le 1,$$

$$u(1,y,t) = (1-y)\exp(1+t) \qquad t \ge 0 \qquad 0 \le y \le 1,$$

$$u(x,0,t) = \exp(x+t) \qquad t \ge 0 \qquad 0 \le x \le 1,$$

and

$$u(x,1,t) = 0 \qquad t \ge 0 \qquad 0 \le x \le 1,$$

This equation has a closed form solution of the type $u(x,y,t) = (1-y)\exp(x+t)$. It is easy to check this by insertion. In this exercise you are asked to set up an explicit and an implicit scheme for solving the above equation. You should also discuss convergence criteria and the numerical stability of the scheme(s) you have chosen. Outline the algorithms for solving the two-dimensional diffusion equation and implement these schemes as functions of Δx (assuming $\Delta x = \Delta y$). Solve the equations numerically and give a critical discussion of your results. Compare your results with the closed form answer.

Part IV
Monte Carlo Methods

We start with an introduction to the basic Monte Carlo philosophy in the first chapter. Thereafter we discuss Markov chains and Brownian motion, we present the famous Metropolis algorithm and discuss several applications of this algorithm to many problems in Science, from the simulation of phase transitions in materials, via studies quantum mechanical systems to the simulation of stock markets and genomic data.

Chapter 11
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*

Abstract We present here the basic philosophy behind stochastic Monte Carlo methods, with an emphasis on numerical integration. Random number generators and properties of probability density functions are also discussed.

11.1 Introduction

Monte Carlo methods are widely used in Science, 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. [58, 59]. Emerging fields like econophysics [60–62] are new examples of 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 are typically 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 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^2.

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 demystify 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

$$\{2,3,4,5,6,7,8,9,10,11,12\}.$$

These values are called the *domain*. To this domain we have the corresponding *probabilities*

$$\{1/36,2/36/3/36,4/36,5/36,6/36,5/36,4/36,3/36,2/36,1/36\}.$$

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

$$\{2,3,4,5,6,7,8,9,10,11,12\},$$

appear in a random order. After 11 throws the results may look like

$$\{10,8,6,3,6,9,11,8,12,4,5\}.$$

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 truly 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 truly 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

$$\lambda \exp -(\lambda x),$$

with λ being proportional to the half-life of the given nucleus which decays.

If you wish to read more about the more formal aspects of Monte Carlo methods, see for example Refs. [63–65].

11.1.1 Definitions

Random numbers as we use them here are numerical approximations to the statistical concept of stochastic variables, sometimes just called random variables. To understand the behavior of pseudo random numbers we must first establish the theoretical framework of stochastic variables. Although this is typical textbook material, the nomenclature may differ from one textbook to another depending on the level of difficulty of the book. We would therefore like to establish a nomenclature suitable for our purpose, one that we are going to use consequently throughout this text.

A stochastic variable can be either continuous or discrete. In any case, we will denote stochastic variables by capital letters $X, Y, \dots$

There are two main concepts associated with a stochastic variable. The *domain* is the set $\mathbb{D} = \{x\}$ of all accessible values the variable can assume, so that $X \in \mathbb{D}$. An example of a discrete domain is the set of six different numbers that we may get by throwing of a dice, $x \in \{1, 2, 3, 4, 5, 6\}$.

The *probability distribution function (PDF)* is a function $p(x)$ on the domain which, in the discrete case, gives us the probability or relative frequency with which these values of X occur

$$p(x) = \mathrm{Prob}(X = x)$$

In the continuous case, the PDF does not directly depict the actual probability. Instead we define the probability for the stochastic variable to assume any value on an infinitesimal interval around x to be $p(x)dx$. The continuous function $p(x)$ then gives us the *density* of the probability rather than the probability itself. The probability for a stochastic variable to assume any value on a non-infinitesimal interval $[a, b]$ is then just the integral

$$\mathrm{Prob}(a \leq X \leq b) = \int_a^b p(x)dx$$

Qualitatively speaking, a stochastic variable represents the values of numbers chosen as if by chance from some specified PDF so that the selection of a large set of these numbers reproduces this PDF.

Also of interest to us is the *cumulative probability distribution function (CDF)*, $P(x)$, which is just the probability for a stochastic variable X to assume any value less than x

$$P(x) = \mathrm{Prob}(X \leq x) = \int_{-\infty}^{x} p(x')dx'$$

The relation between a CDF and its corresponding PDF is then

$$p(x) = \frac{d}{dx}P(x)$$

There are two properties that all PDFs must satisfy. The first one is positivity (assuming that the PDF is normalized)

$$0 \leq p(x) \leq 1$$

Naturally, it would be nonsensical for any of the values of the domain to occur with a probability greater than 1 or less than 0. Also, the PDF must be normalized. That is, all the probabilities must add up to unity. The probability of "anything" to happen is always unity. For both discrete and continuous PDFs, this condition is

$$\sum_{x_i \in \mathbb{D}} p(x_i) = 1$$

$$\int_{x \in \mathbb{D}} p(x)\,dx = 1$$

In addition to the exponential distribution discussed above, there are two other continuous PDFs that are especially important. The first one is the most basic PDF; namely the uniform distribution

$$p(x) = \frac{1}{b-a}\theta(x-a)\theta(b-x) \tag{11.1}$$

with:

$$\theta(x) = 0 \; x < 0$$
$$\theta(x) = 1 \; x \geq 0$$

The second one is the Gaussian Distribution, often called the normal distribution

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}}\exp{(-\frac{(x-\mu)^2}{2\sigma^2})}$$

Let $h(x)$ be an arbitrary function on the domain of the stochastic variable X whose PDF is $p(x)$. We define the *expectation value* of h with respect to p as follows

$$\langle h \rangle_X \equiv \int h(x)p(x)\,dx \tag{11.2}$$

Whenever the PDF is known implicitly, like in this case, we will drop the index X for clarity.

A particularly useful class of special expectation values are the *moments*. The n-th moment of the PDF p is defined as follows

$$\langle x^n \rangle \equiv \int x^n p(x)\,dx$$

The zero-th moment $\langle 1 \rangle$ is just the normalization condition of p. The first moment, $\langle x \rangle$, is called the *mean* of p and often denoted by the letter μ

$$\langle x \rangle = \mu \equiv \int xp(x)\,dx$$

Qualitatively it represents the centroid or the average value of the PDF and is therefore often simply called the expectation value of p.[1] A PDF can in principle be expanded in the set of its moments [66]. For two PDFs to be equal, each of their moments must be equal.

A special version of the moments is the set of *central moments*, the n-th central moment defined as

$$\langle (x-\langle x\rangle)^n\rangle \equiv \int (x-\langle x\rangle)^n p(x)\,dx$$

The zero-th and first central moments are both trivial, equal 1 and 0, respectively. But the second central moment, known as the *variance* of p, is of particular interest. For the stochastic variable X, the variance is denoted as σ_X^2 or $\mathrm{Var}(X)$

$$\begin{aligned}\sigma_X^2 \;=\; \mathrm{Var}(X) &= \langle (x-\langle x\rangle)^2\rangle = \int (x-\langle x\rangle)^2 p(x)\,dx\\ &= \int \left(x^2-2x\langle x\rangle \;+\langle x\rangle^2\right) p(x)\,dx\\ &= \langle x^2\rangle - 2\langle x\rangle\langle x\rangle + \langle x\rangle^2\\ &= \langle x^2\rangle - \langle x\rangle^2\end{aligned}$$

The square root of the variance, $\sigma = \sqrt{\langle (x-\langle x\rangle)^2\rangle}$ is called the *standard deviation* of p. It is clearly just the RMS (root-mean-square) value of the deviation of the PDF from its mean value, interpreted qualitatively as the "spread" of p around its mean.

We will also be interested in finding the PDF of a *function* of a stochastic variable. Let the stochastic variable X have the PDF $p_X(x)$, and let $Y=h(X)$ be a function of X. What we want to find is the PDF of Y, $p_Y(y)$. We will have to restrict ourselves to the case where $h(X)$ is invertible, so that it has to be strictly monotonous. First we construct the cumulative distribution of Y, considering only the case where h increases

$$P_Y(y) = \mathrm{Prob}(Y\le y) = \mathrm{Prob}(h(X)\le y) = \mathrm{Prob}(X\le h^{-1}(y)) = P_X(h^{-1}(y))$$

where h^{-1} is the inverse function of h, meaning that if $y=h(x)$ then $x=h^{-1}(y)$. This gives the PDF of Y

$$p_Y(y) = \frac{d}{dy}P_Y(y) = \frac{d}{dy}P_X(h^{-1}(y))$$

Considering in a similar manner the other case of a decreasing h we arrive at

$$p_Y(y) = p_X(h^{-1}(y))\left|\frac{d}{dy}h^{-1}(y)\right| \qquad (11.3)$$

This formula will become useful when transforming simple pseudo random number generators to more general ones.

All the PDFs above have been written as functions of only one stochastic variable. Such PDFs are called *univariate*. A PDF may well consist of any number of variables, in which case we call it *multivariate*. A general multivariate expectation value is defined similarly as for the univariate case, but all stochastic variables are taken into account at once. Let $P(x_1,\ldots,x_n)$ be the multivariate PDF for the set $\{X_i\}$ of n stochastic variables and let $H(x_1,\ldots,x_n)$ be an arbitrary function over the joint domain of all X_i. The expectation value of H with respect to P is defined as follows

[1] We should now formulate 11.2 in a more rigorous manner. It is mathematically more correct to speak of h as a function transforming the stochastic variable X to the stochastic variable Y, $Y=h(X)$. Let $p_X(x)$ be the known PDF of X, and $p_Y(y)$ be the unknown PDF of Y. It can then be shown [66] that the expectation value of Y, namely $\langle y\rangle_Y = \int y p_Y(y)\,dy$, must equal what we have defined as the expectation value of $h(x)$ with respect to p_X, namely $\langle h\rangle_X = \int h(x)p_X(x)\,dx$.

$$\langle H\rangle_{X_1\dots X_n} = \int\cdots\int H(x_1,\dots,x_n)P(x_1,\dots,x_n)\,dx_1\dots dx_n$$

If we want to find the expectation value of an arbitrary function $h(x_i)$ on the domain of just one stochastic variable X_i, we must still use the joint PDF P and remember to integrate over the total domain of all X_i

$$\langle h\rangle_{X_1\dots X_n} = \int\cdots\int h(x_i)P(x_1,\dots,x_n)\,dx_1\dots dx_n \tag{11.4}$$

We will now define the property of correlation, of great importance for our study of random numbers. Let us continue with the same set of n stochastic variables $\{X_i\}$ as above. The variables are *uncorrelated* (or independent) if P may be factorized in the following form

$$P(x_1,x_2,\dots,x_n) = \prod_{i=1}^{n} p_i(x_i)$$

where $p_i(x_i)$ is the univariate PDF of X_i. Notice, that if all X_i are uncorrelated, then the above equation for the expectation value of the univariate function h, eq. (11.4) reduces, nicely to the familiar simple univariate form of eq. (11.2).

To understand the definition of independence qualitatively, consider a process of n sequential events determined by the stochastic variables $X_i\ \forall\ i\in\{1,2,\dots,n\}$. The PDF $p_i(x_i)$ determines the probability density that the i-th event (governed by X_i) will have the outcome x_i. If the individual events are to be independent, then the joint probability density should intuitively be just the product of the individual densities. The events receive no information about each other. The probability to get some particular outcome of an event is independent of whether other events are happening at all or not.

11.1.2 First Illustration of the Use of Monte-Carlo Methods

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 5 we discussed standard methods for evaluating an integral like

$$I = \int_0^1 f(x)dx \approx \sum_{i=1}^{N}\omega_i f(x_i),$$

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. (5.5), 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

$$I = \int_0^1 f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_{i-1/2}),$$

where $x_{i-1/2}$ are the midpoint values of x. Introducing the concept of the average of the function f for a given PDF $p(x)$ as

$$\langle f \rangle = \frac{1}{N}\sum_{i=1}^{N} f(x_i)p(x_i),$$

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 $x \in [0,1]$

$$I = \int_0^1 f(x)dx \approx \langle f \rangle.$$

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

$$\sigma_f^2 = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - \langle f \rangle)^2 p(x_i),$$

and inserting the uniform distribution this yields

$$\sigma_f^2 = \frac{1}{N}\sum_{i=1}^{N} f(x_i)^2 - \left(\frac{1}{N}\sum_{i=1}^{N} f(x_i)\right)^2,$$

or

$$\sigma_f^2 = \left(\langle f^2 \rangle - \langle f \rangle^2\right).$$

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 recalculate the above average and variance for a series of different measurements. If each such measurement 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

$$\langle I \rangle_M = \frac{1}{M}\sum_{l=1}^{M}\langle f \rangle_l.$$

We show in section 11.3 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 $M = N$)

$$\sigma_N^2 \approx \frac{1}{N}\left(\langle f^2 \rangle - \langle f \rangle^2\right) = \frac{\sigma_f^2}{N}. \qquad (11.5)$$

We note that the standard deviation is proportional to the inverse square root of the number of measurements

$$\sigma_N \sim \frac{1}{\sqrt{N}}.$$

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 chapter 5 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 algorithm with error in powers of k 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 one Petaflops, that is 10^{15} 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

$$\int_0^1 dx \frac{4}{1+x^2} = \pi,$$

where the input is the desired number of Monte Carlo samples. The program is listed below.

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

$$p(x) = \frac{1}{b-a} \Theta(x-a) \Theta(b-x),$$

with $a = 0$ og $b = 1$ and where Θ is the standard Heaviside function or simply the step function. If we have a general interval $[a,b]$, we can still use these random number generators through a change of variables

$$z = a + (b-a)x,$$

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 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 $[0,1]$ through 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[2] implements the above algorithm using the library function *ran*0 to compute π. Note again the inclusion of the *lib.h* file which has the random number generator function **ran0**.

http://folk.uio.no/mhjensen/compphys/programs/chapter11/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;
  value = 4/(1.+x*x);
  return value;
} // end of function to evaluate
```

[2] 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.

Note that we transfer the variable *idum* in order to initialize the random number generator from the function *ran*0. The variable *idum* gets changed for every sampling. This variable is called the *seed*. The results of our computations are listed in Table 11.1. 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 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;
```

Table 11.1 Results for $I = \pi = 4\int_0^1 dx/(1+x^2)$ 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

11.1.3 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 this

- Choose the number of particles N.
- Make a loop over time, where the maximum time (or maximum number of steps) 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://folk.uio.no/mhjensen/compphys/programs/chapter11/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
```

Figure 11.1 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

$$\sigma = \sqrt{\langle n_l^2 \rangle - \langle n_l \rangle^2}. \tag{11.6}$$

This problem has also an analytic solution to which we can compare our numerical simulation. If $n_l(t)$ is the number of particles in the left half after t moves, the change in $n_l(t)$ in the time interval Δt is

$$\Delta n = \left(\frac{N - n_l(t)}{N} - \frac{n_l(t)}{N} \right) \Delta t,$$

and assuming that n_l and t are continuous variables we arrive at

$$\frac{dn_l(t)}{dt} = 1 - \frac{2n_l(t)}{N},$$

whose solution is

$$n_l(t) = \frac{N}{2}\left(1 + e^{-2t/N}\right),$$

with the initial condition $n_l(t = 0) = N$. Note that we have assumed n to be a continuous variable. Obviously, particles are discrete objects.

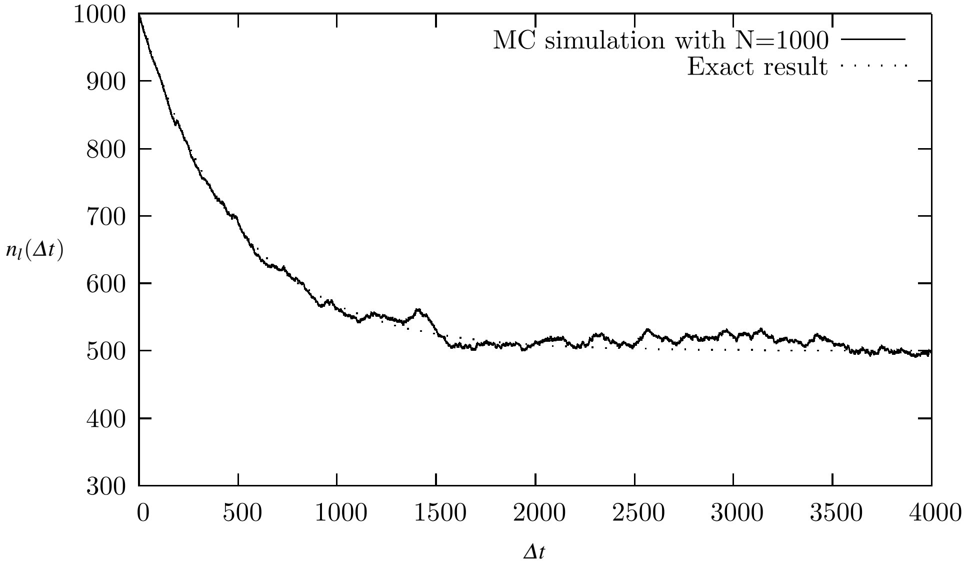


Fig. 11.1 Number of particles in the left half of the container as function of the number of time steps. The solution is compared with the analytic expression. $N = 1000$.

11.1.4 Radioactive Decay

Radioactive decay is among one of the classical examples of Monte-Carlo simulations. Assume that at 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

$$dN(t) = -\omega N(t)dt,$$

whose solution is

$$N(t) = N(0)e^{-\omega t},$$

where we have defined the mean lifetime τ of X as

$$\tau = \frac{1}{\omega}.$$

If a nucleus X decays to a daugther nucleus Y which also can decay, we get the following coupled equations

$$\frac{dN_X(t)}{dt} = -\omega_X N_X(t),$$

and

$$\frac{dN_Y(t)}{dt} = -\omega_Y N_Y(t) + \omega_X N_X(t).$$

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.

11.1.5 Program Example for Radioactive Decay

The program is split in four tasks, a main program with various declarations,

http://folk.uio.no/mhjensen/compphys/programs/chapter11/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 but contained in the full listing available at the webpage. 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.

11.1.6 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 an 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 11.4 discusses Monte Carlo integration in general, how to choose the correct weighting function and how to evaluate integrals with dimensions $d > 1$.

11.2 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 11.2 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	$0 \le p(x_i) \le 1$	$p(x) \ge 0$
Positivity	$0 \le P_i \le 1$	$0 \le P(x) \le 1$
Monotonic	$P_i \ge P_j$ if $x_i \ge x_j$	$P(x_i) \ge P(x_j)$ if $x_i \ge x_j$
Normalization	$P_N = 1$	$P(b) = 1$

With a PDF we can compute expectation values of selected quantities such as

$$\langle x^k \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i^k p(x_i),$$

if we have a discrete PDF or

$$\langle x^k \rangle = \int_a^b x^k p(x) dx,$$

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

$$\langle f\rangle = \int_a^b f(x)p(x)dx.$$

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
$$p(x) = \frac{1}{b-a}\Theta(x-a)\Theta(b-x),$$
yielding probabilities different from zero in the interval $[a,b]$. The mean value and the variance for this distribution are discussed in section 11.3.
2. The exponential distribution
$$p(x) = \alpha\exp(-\alpha x),$$
yielding probabilities different from zero in the interval $[0,\infty)$ and with mean value
$$\mu = \int_0^\infty xp(x)dx = \int_0^\infty x\alpha\exp(-\alpha x)dx = \frac{1}{\alpha}$$
and variance
$$\sigma^2 = \int_0^\infty x^2p(x)dx - \mu^2 = \frac{1}{\alpha^2}.$$
3. Finally, we have the so-called univariate normal distribution, or just the normal distribution
$$p(x) = \frac{1}{b\sqrt{2\pi}}\exp\left(-\frac{(x-a)^2}{2b^2}\right)$$
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
$$\mu = \int_0^\infty xp(x)dx = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty} x\exp\left(-\frac{(x-a)^2}{2b^2}\right)dx,$$
which becomes with a suitable change of variables
$$\mu = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty} b\sqrt{2}(a+b\sqrt{2}y)\exp{-y^2}dy = a.$$
Similarly, the variance becomes
$$\sigma^2 = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty}(x-\mu)^2\exp\left(-\frac{(x-a)^2}{2b^2}\right)dx,$$
and inserting the mean value and performing a variable change we obtain
$$\sigma^2 = \frac{1}{b\sqrt{2\pi}}\int_{-\infty}^{\infty} b\sqrt{2}(b\sqrt{2}y)^2\exp\left(-y^2\right)dy = \frac{2b^2}{\sqrt{\pi}}\int_{-\infty}^{\infty} y^2\exp\left(-y^2\right)dy,$$
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
$$p(x) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{x^2}{2}\right). \tag{11.7}$$

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

$$P(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} \exp\left(-\frac{t^2}{2}\right)dt,$$

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

$$p(x) = \binom{n}{x} y^x(1-y)^{n-x} \qquad x = 0,1,\dots,n,$$

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 (either heads or tails) 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

$$(a+b)^m = \sum_{n=0}^{m}\binom{m}{n} a^n b^{m-n},$$

which can be used to show that

$$\sum_{x=0}^{n}\binom{n}{x} y^x(1-y)^{n-x} = (y+1-y)^n = 1,$$

the PDF is normalized to one. The mean value is

$$\mu = \sum_{x=0}^{n} x\binom{n}{x} y^x(1-y)^{n-x} = \sum_{x=0}^{n} x\frac{n!}{x!(n-x)!}y^x(1-y)^{n-x},$$

resulting in

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

which we rewrite as

$$\mu = ny\sum_{\nu=0}^{n}\binom{n-1}{\nu} y^\nu(1-y)^{n-1-\nu} = ny(y+1-y)^{n-1} = ny.$$

The variance is slightly trickier to get. 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

$$p(x) = \frac{\lambda^x}{x!}e^{-\lambda} \qquad x = 0,1,\dots,;\lambda > 0.$$

In this case both the mean value and the variance are easier to calculate,

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

and the variance is $\sigma^2 = \lambda$. An example of applications of the Poisson distribution could be 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

$$\lim_{n\to\infty}\binom{n}{x}y^x(1-y)^{n-x}e^{-\lambda} = \sum_{x=1}^{\infty}\frac{\lambda^x}{x!}e^{-\lambda},$$

see for example Refs. [63, 64] for a proof.

11.2.1 Multivariable Expectation Values

An important quantity is the so called covariance, a variant of the variance. Consider the set $\{X_i\}$ of n stochastic variables (not necessarily uncorrelated) with the multivariate PDF $P(x_1,\dots,x_n)$. The *covariance* of two of the stochastic variables, X_i and X_j, is defined as follows

$$\begin{aligned}\mathrm{Cov}(X_i,X_j) &\equiv \big\langle (x_i - \langle x_i\rangle)(x_j - \langle x_j\rangle)\big\rangle \\ &= \int\cdots\int (x_i - \langle x_i\rangle)(x_j - \langle x_j\rangle)\,P(x_1,\dots,x_n)\,dx_1\dots dx_n \end{aligned} \tag{11.8}$$

with

$$\langle x_i\rangle = \int\cdots\int x_i\,P(x_1,\dots,x_n)\,dx_1\dots dx_n$$

If we consider the above covariance as a matrix $C_{ij} = \mathrm{Cov}(X_i,X_j)$, then the diagonal elements are just the familiar variances, $C_{ii} = \mathrm{Cov}(X_i,X_i) = \mathrm{Var}(X_i)$. It turns out that all the off-diagonal elements are zero if the stochastic variables are uncorrelated. This is easy to show, keeping in mind the linearity of the expectation value. Consider the stochastic variables X_i and X_j, $(i \neq j)$

$$\begin{aligned}\mathrm{Cov}(X_i,X_j) &= \big\langle (x_i - \langle x_i\rangle)(x_j - \langle x_j\rangle)\big\rangle \\ &= \langle x_ix_j - x_i\langle x_j\rangle - \langle x_i\rangle x_j + \langle x_i\rangle\langle x_j\rangle\rangle \\ &= \langle x_ix_j\rangle - \langle x_i\langle x_j\rangle\rangle - \langle\langle x_i\rangle x_j\rangle + \langle\langle x_i\rangle\langle x_j\rangle\rangle \\ &= \langle x_ix_j\rangle - \langle x_i\rangle\langle x_j\rangle - \langle x_i\rangle\langle x_j\rangle + \langle x_i\rangle\langle x_j\rangle \\ &= \langle x_ix_j\rangle - \langle x_i\rangle\langle x_j\rangle \end{aligned}$$

If X_i and X_j are independent, we get $\langle x_ix_j\rangle = \langle x_i\rangle\langle x_j\rangle$, resulting in $\mathrm{Cov}(X_i,X_j) = 0 \ \ (i \neq j)$.

Also useful for us is the covariance of linear combinations of stochastic variables. Let $\{X_i\}$ and $\{Y_i\}$ be two sets of stochastic variables. Let also $\{a_i\}$ and $\{b_i\}$ be two sets of scalars. Consider the linear combination

$$U = \sum_i a_iX_i \qquad V = \sum_j b_jY_j$$

By the linearity of the expectation value, it can be shown [66] that

$$\mathrm{Cov}(U,V) = \sum_{i,j} a_i b_j \mathrm{Cov}(X_i,Y_j)$$

Now, since the variance is just $\mathrm{Var}(X_i) = \mathrm{Cov}(X_i,X_i)$, we get the variance of the linear combination $U = \sum_i a_i X_i$

$$\mathrm{Var}(U) = \sum_{i,j} a_i a_j \mathrm{Cov}(X_i,X_j) \tag{11.9}$$

And in the special case when the stochastic variables are uncorrelated, the off-diagonal elements of the covariance are as we know zero, resulting in

$$\mathrm{Var}(U) = \sum_i a_i^2 \mathrm{Cov}(X_i,X_i) = \sum_i a_i^2 \mathrm{Var}(X_i)$$

$$\mathrm{Var}(\sum_i a_i X_i) = \sum_i a_i^2 \mathrm{Var}(X_i)$$

which will become very useful in our study of the error in the mean value of a set of measurements.

Now that we have constructed an idealized mathematical framework, let us try to apply it to empirical observations. Examples of relevant physical phenomena may be spontaneous decays of nuclei, or a purely mathematical set of numbers produced by some deterministic mechanism. It is the latter we will deal with, using so-called pseudo-random number generators. In general our observations will contain only a limited set of observables. We remind the reader that a *stochastic process* is a process that produces sequentially a chain of values

$$\{x_1,x_2,\dots x_k,\dots\}.$$

We will call these values our *measurements* and the entire set as our measured *sample*. The action of measuring all the elements of a sample we will call a stochastic *experiment* (since, operationally, they are often associated with results of empirical observation of some physical or mathematical phenomena; precisely an experiment). We assume that these values are distributed according to some PDF $p_X(x)$, where X is just the formal symbol for the stochastic variable whose PDF is $p_X(x)$. Instead of trying to determine the full distribution p we are often only interested in finding the few lowest moments, like the mean μ_X and the variance σ_X.

In practical situations however, a sample is always of finite size. Let that size be n. The expectation value of a sample α, the *sample mean*, is then defined as follows

$$\langle x_\alpha \rangle \equiv \frac{1}{n}\sum_{k=1}^{n} x_{\alpha,k}.$$

The *sample variance* is:

$$\mathrm{Var}(x) \equiv \frac{1}{n}\sum_{k=1}^{n} (x_{\alpha,k} - \langle x_\alpha \rangle)^2,$$

with its square root being the *standard deviation of the sample*.

You can think of the above observables as a set of quantities which define a given experiment. This experiment is then repeated several times, say m times. The total average is then

$$\langle X_m \rangle = \frac{1}{m}\sum_{\alpha=1}^{m} x_\alpha = \frac{1}{mn}\sum_{\alpha,k} x_{\alpha,k}, \tag{11.10}$$

where the last sums end at m and n. The total variance is

$$\sigma_m^2 = \frac{1}{mn^2}\sum_{\alpha=1}^{m}\left(\langle x_\alpha\rangle - \langle X_m\rangle\right)^2,$$

which we rewrite as

$$\sigma_m^2 = \frac{1}{m}\sum_{\alpha=1}^{m}\sum_{kl=1}^{n}(x_{\alpha,k} - \langle X_m\rangle)(x_{\alpha,l} - \langle X_m\rangle). \tag{11.11}$$

We define also the sample variance σ^2 of all mn individual experiments as

$$\sigma^2 = \frac{1}{mn}\sum_{\alpha=1}^{m}\sum_{k=1}^{n}(x_{\alpha,k} - \langle X_m\rangle)^2. \tag{11.12}$$

These quantities, being known experimental values or the results from our calculations, may differ, in some cases significantly, from the similarly named exact values for the mean value μ_X, the variance $\mathrm{Var}(X)$ and the covariance $\mathrm{Cov}(X,Y)$.

The law of large numbers (see for example [66] and the next subsection) states that as the size of our sample grows to infinity, the sample mean approaches the true mean μ_X of the chosen PDF:

$$\lim_{n\to\infty}\langle x_\alpha\rangle = \mu_X$$

The sample mean $\bar{x}_n$ works therefore as an estimate of the true mean μ_X.

What we need to find out is how good an approximation $\bar{x}_n$ is to μ_X. In any stochastic measurement, an estimated mean is of no use to us without a measure of its error. A quantity that tells us how well we can reproduce it in another experiment. We are therefore interested in the PDF of the sample mean itself. Its standard deviation will be a measure of the spread of sample means, and we will simply call it the *error* of the sample mean, or just sample error, and denote it by err_X. In practice, we will only be able to produce an *estimate* of the sample error since the exact value would require the knowledge of the true PDFs behind, which we usually do not have.

The straight forward brute force way of estimating the sample error is simply by producing a number of samples, and treating the mean of each as a measurement. The standard deviation of these means will then be an estimate of the original sample error. If we are unable to produce more than one sample, we can split it up sequentially into smaller ones, treating each in the same way as above. This procedure is known as *blocking* and will be given more attention in later chapters. At this point it is worth while exploring more indirect methods of estimation that will help us understand some important underlying principles of correlation effects.

Let us first take a look at what happens to the sample error as the size of the sample grows. We derive here the central limit theorem first.

11.2.2 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 m such mean values x_i

$$z = \frac{x_1 + x_2 + \cdots + x_m}{m},$$

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

$$\tilde{p}(z) = \int dx_1 p(x_1) \int dx_2 p(x_2) \dots \int dx_m p(x_m) \delta(z - \frac{x_1 + x_2 + \cdots + x_m}{m}),$$

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 $p(x_i)$. The independence assumption is important in the derivation of the central limit theorem.

If we use the integral expression for the δ-function

$$\delta(z - \frac{x_1 + x_2 + \cdots + x_m}{m}) = \frac{1}{2\pi} \int_{-\infty}^{\infty} dq \exp\left(iq(z - \frac{x_1 + x_2 + \cdots + x_m}{m})\right),$$

and inserting $e^{i\mu q - i\mu q}$ where μ is the mean value we arrive at

$$\tilde{p}(z) = \frac{1}{2\pi} \int_{-\infty}^{\infty} dq \exp\left(iq(z - \mu)\right) \left[\int_{-\infty}^{\infty} dx p(x) \exp\left(iq(\mu - x)/m\right) \right]^m,$$

with the integral over x resulting in

$$\int_{-\infty}^{\infty} dx p(x) \exp\left(iq(\mu - x)/m\right) = \int_{-\infty}^{\infty} dx p(x) \left[1 + \frac{iq(\mu - x)}{m} - \frac{q^2(\mu - x)^2}{2m^2} + \dots \right].$$

The second term on the rhs disappears since this is just the mean and employing the definition of σ^2 we have

$$\int_{-\infty}^{\infty} dx p(x) e^{(iq(\mu - x)/m)} = 1 - \frac{q^2\sigma^2}{2m^2} + \dots,$$

resulting in

$$\left[\int_{-\infty}^{\infty} dx p(x) \exp\left(iq(\mu - x)/m\right) \right]^m \approx \left[1 - \frac{q^2\sigma^2}{2m^2} + \dots \right]^m,$$

and in the limit $m \to \infty$ we obtain

$$\tilde{p}(z) = \frac{1}{\sqrt{2\pi}(\sigma/\sqrt{m})} \exp\left(-\frac{(z - \mu)^2}{2(\sigma/\sqrt{m})^2} \right),$$

which is the normal distribution with variance $\sigma_m^2 = \sigma^2/m$, 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 m 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 m, the number of values used to compute z.

The theorem is satisfied by a large class of PDFs. Note however that for a finite m, 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

$$\sigma_m = \frac{\sigma}{\sqrt{m}}.$$

The latter is true only if the average value is known exactly. This is obtained in the limit $m \to \infty$ only. Because the mean and the variance are measured quantities we obtain the familiar expression in statistics

$$\sigma_m \approx \frac{\sigma}{\sqrt{m-1}},$$

see for example Ref. [66] for further discussions.

In many cases however the above estimate for the standard deviation, in particular if correlations are strong, may be too simplistic. We need therefore a more precise defintion of the error and the variance in our results.

11.2.3 Definition of Correlation Functions and Standard Deviation

Let us now return to the definition of the variance and standard deviation of our measurements. Our estimate of the true average μ_X is then the sample mean $\langle X_m \rangle$

$$\mu_X \approx X_m = \frac{1}{mn}\sum_{\alpha=1}^{m}\sum_{k=1}^{n} x_{\alpha,k}.$$

We can then use Eq. (11.11)

$$\sigma_m^2 = \frac{1}{mn^2}\sum_{\alpha=1}^{m}\sum_{kl=1}^{n}(x_{\alpha,k}-\langle X_m\rangle)(x_{\alpha,l}-\langle X_m\rangle),$$

and rewrite it as

$$\sigma_m^2 = \frac{\sigma^2}{n} + \frac{2}{mn^2}\sum_{\alpha=1}^{m}\sum_{k<l}^{n}(x_{\alpha,k}-\langle X_m\rangle)(x_{\alpha,l}-\langle X_m\rangle),$$

where the first term is the sample variance of all mn experiments divided by n and the last term is nothing but the covariance which arises when $k \neq l$. If the observables are uncorrelated, then the covariance is zero and we obtain a total variance which agrees with the central limit theorem. Correlations may often be present in our data set, resulting in a non-zero covariance. The first term is normally called the uncorrelated contribution. Computationally the uncorrelated first term is much easier to treat efficiently than the second. We just accumulate separately the values x^2 and x for every measurement x we receive. The correlation term, though, has to be calculated at the end of the experiment since we need all the measurements to calculate the cross terms. Therefore, all measurements have to be stored throughout the experiment.

Let us analyze the problem by splitting up the correlation term into partial sums of the form

$$f_d = \frac{1}{nm}\sum_{\alpha=1}^{m}\sum_{k=1}^{n-d}(x_{\alpha,k}-\langle X_m\rangle)(x_{\alpha,k+d}-\langle X_m\rangle),$$

The correlation term of the total variance can now be rewritten in terms of f_d

$$\frac{2}{mn^2}\sum_{\alpha=1}^{m}\sum_{k<l}^{n}(x_{\alpha,k}-\langle X_m\rangle)(x_{\alpha,l}-\langle X_m\rangle) = \frac{2}{n}\sum_{d=1}^{n-1} f_d$$

The value of f_d reflects the correlation between measurements separated by the distance d in the samples. Notice that for $d=0$, f is just the sample variance, σ^2. If we divide f_d by σ^2, we arrive at the so called *autocorrelation function*

$$\kappa_d = \frac{f_d}{\sigma^2} \tag{11.13}$$

which gives us a useful measure of the correlation pair correlation starting always at 1 for $d=0$.

The sample variance of the mn experiments can now be written in terms of the autocorrelation function

$$\sigma_m^2 = \frac{\sigma^2}{n} + \frac{2}{n}\cdot\sigma^2\sum_{d=1}^{n-1}\frac{f_d}{\sigma^2} = \left(1+2\sum_{d=1}^{n-1}\kappa_d\right)\frac{1}{n}\sigma^2 = \frac{\tau}{n}\cdot\sigma^2 \tag{11.14}$$

and we see that σ_m can be expressed in terms of the uncorrelated sample variance times a correction factor τ which accounts for the correlation between measurements. We call this correction factor the *autocorrelation time*

$$\tau = 1+2\sum_{d=1}^{n-1}\kappa_d \tag{11.15}$$

For a correlation free experiment, τ equals 1. From the point of view of Eq. (11.14) we can interpret a sequential correlation as an effective reduction of the number of measurements by a factor τ. The effective number of measurements becomes

$$n_{\mathrm{eff}} = \frac{n}{\tau}$$

To neglect the autocorrelation time τ will always cause our simple uncorrelated estimate of $\sigma_m^2 \approx \sigma^2/n$ to be less than the true sample error. The estimate of the error will be too “good”. On the other hand, the calculation of the full autocorrelation time poses an efficiency problem if the set of measurements is very large. The solution to this problem is given by more practically oriented methods like the blocking technique, see for example Ref. [67] for more details. This method is discussed in more detail in chapter 15.

11.3 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 algorithms are in reality pseudo random numbers, hopefully abiding to the following criteria:

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

$$x_{i+1} = cx_i(1-x_i),$$

which is often used as an example of a chaotic system. The variable c is a constant and for certain values of c and x_0 the system can settle down quickly into a regular periodic sequence of values $x_1, x_2, x_3, \ldots$. For $x_0 = 0.1$ and $c = 3.2$ we obtain a periodic pattern as shown in Fig. 11.2. 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.

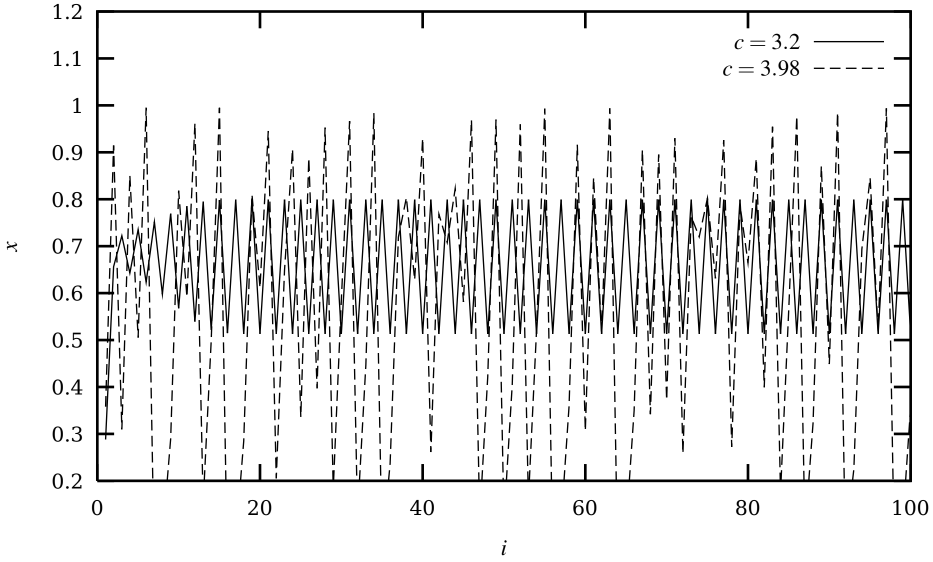


Fig. 11.2 Plot of the logistic mapping $x_{i+1} = cx_i(1-x_i)$ for $x_0 = 0.1$ and $c = 3.2$ and $c = 3.98$.

The most common random number generators are based on so-called Linear congruential relations of the type

$$N_i = (aN_{i-1} + c)\mathrm{MOD}(M),$$

which yield a number in the interval [0,1] through

$$x_i = N_i/M$$

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

$$N_i = (6N_{i-1} + 7)\mathrm{MOD}(5),$$

with a seed $N_0 = 2$. This generator produces the sequence $4, 1, 3, 0, 2, 4, 1, 3, 0, 2, \ldots\ldots$, i.e., a sequence with period 5. However, increasing M may not guarantee a larger period as the following example shows

$$N_i = (27N_{i-1} + 11)\mathrm{MOD}(54),$$

which still, with $N_0 = 2$, results in $11, 38, 11, 38, 11, 38, \ldots$, 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,

$$N_l = (aN_{l-i} + cN_{l-j})\mathrm{MOD}(M).$$

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 [68] which consists of two congruential relations

$$N_l = (N_{l-3} - N_{l-1})\mathrm{MOD}(2^{31} - 69), \tag{11.16}$$

followed by

$$N_l = (69069N_{l-1} + 1013904243)\mathrm{MOD}(2^{32}), \tag{11.17}$$

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

$$N_l = (N_{l-i}) \oplus (N_{l-j})$$

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 how the function $ran0$ implements

$$N_i = (aN_{i-1})\mathrm{MOD}(M).$$

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

$$(aN_{i-1})\mathrm{MOD}(M) = (aN_{i-1} - [N_{i-1}/q]M)\mathrm{MOD}(M), \tag{11.18}$$

since we can add or subtract any integer multiple of M from aN_{i-1}. The last term $[N_{i-1}/q]M\mathrm{MOD}(M)$ is zero since the integer division $[N_{i-1}/q]$ just yields a constant which is multiplied with M. We can now rewrite Eq. (11.18) as

$$(aN_{i-1})\mathrm{MOD}(M) = (aN_{i-1} - [N_{i-1}/q](aq+r))\mathrm{MOD}(M), \tag{11.19}$$

which results in

$$(aN_{i-1})\mathrm{MOD}(M) = (a(N_{i-1} - [N_{i-1}/q]q) - [N_{i-1}/q]r))\,\mathrm{MOD}(M), \tag{11.20}$$

yielding

$$(aN_{i-1})\mathrm{MOD}(M) = (a(N_{i-1}\mathrm{MOD}(q)) - [N_{i-1}/q]r))\,\mathrm{MOD}(M). \tag{11.21}$$

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 $N_{i-1}\mathrm{MOD}(q)$ is between zero and $q-1$ then $a(N_{i-1}\mathrm{MOD}(q)) < aq$. 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
   ** 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 Ref. [36]. Here we limit ourselves to study selected properties of these generators.

11.3.1 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 11.3 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, the mean value μ is then

$$\mu = \langle x \rangle = \frac{1}{2}$$

while the standard deviation is

$$\sigma = \sqrt{\langle x^2 \rangle - \mu^2} = \frac{1}{\sqrt{12}} = 0.2886.$$

The various random number generators produce results which agree rather well with these limiting values.

Table 11.3 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

There are many other tests which can be performed. Often a picture of the numbers generated may reveal possible patterns.

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 defined previously, see again Eq. (11.13). We rewrite it here as

$$C_k = \frac{f_d}{\sigma^2},$$

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. (11.5) is no longer valid.

Figure 11.3 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.

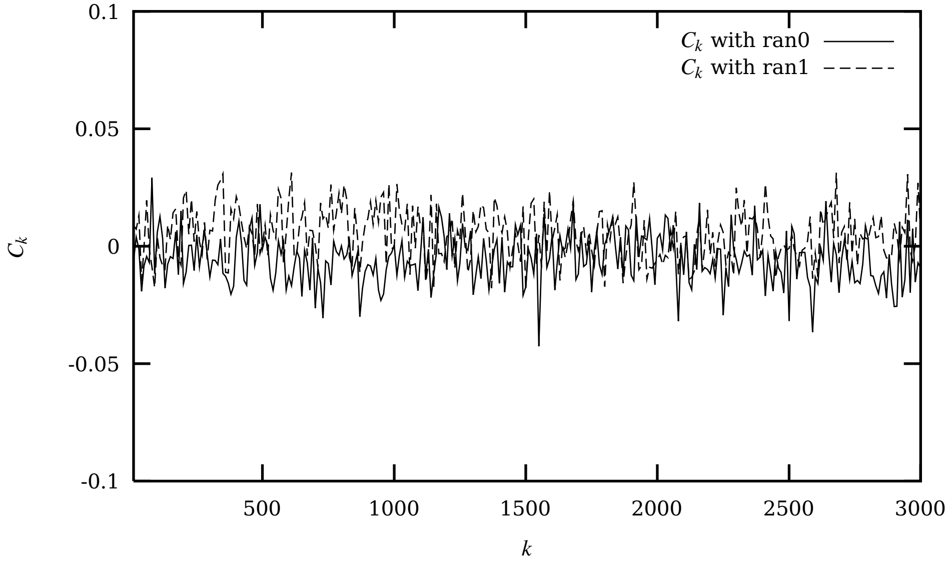


Fig. 11.3 Plot of the auto-correlation function C_k for various k-values for $N = 10000$ using the random number generators *ran*0 and *ran*1.

11.4 Improved Monte Carlo Integration

In section 11.1 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 $[0,1]$

$$I = \int_0^1 f(x)dx = \langle f \rangle.$$

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 or zero. 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

$$H(x)\Psi(x) = E\Psi(x),$$

where $H(x)$ is the hamiltonian under consideration. The expectation value of H, assuming that the quantum mechanical PDF is normalized, is given by

$$\langle H\rangle = \int dx\Psi(x)^* H(x)\Psi(x).$$

We could insert $\Psi(x)/\Psi(x)$ right to the left of H and rewrite the last equation as

$$\langle H\rangle = \int dx\Psi(x)^*\Psi(x)\frac{H(x)}{\Psi(x)}\Psi(x), \tag{11.22}$$

or

$$\langle H\rangle = \int dxp(x)\tilde{H}(x),$$

which is on the form of an expectation value with

$$\tilde{H}(x) = \frac{H(x)}{\Psi(x)}\Psi(x).$$

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

$$\langle H\rangle = \int dxp(x)E = E,$$

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.

11.4.1 Change of Variables

The starting point is always the uniform distribution

$$p(x)dx = \begin{cases} dx & 0 \le x \le 1 \\ 0 & else \end{cases}$$

with $p(x) = 1$ and satisfying

$$\int_{-\infty}^{\infty} p(x)dx = 1.$$

All random number generators provided in the program library generate numbers in this domain.

When we attempt a transformation to a new variable $x \rightarrow y$ we have to conserve the probability

$$p(y)dy = p(x)dx,$$

which for the uniform distribution implies

$$p(y)dy = dx.$$

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

$$x(y) = \int_0^y p(y')dy',$$

which is nothing but the cumulative distribution of $p(y)$, i.e.,

$$x(y) = P(y) = \int_0^y p(y')dy'.$$

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.

11.4.1.1 Transformed Uniform Distribution

Suppose we have the general uniform distribution

$$p(y)dy = \begin{cases} \frac{dy}{b-a} & a \le y \le b \\ 0 & else \end{cases}$$

If we wish to relate this distribution to the one in the interval $x \in [0,1]$ we have

$$p(y)dy = \frac{dy}{b-a} = dx,$$

and integrating we obtain the cumulative function

$$x(y) = \int_a^y \frac{dy'}{b-a},$$

yielding

$$y = a + (b-a)x,$$

a well-known result!

11.4.1.2 Exponential Distribution

Assume that

$$p(y) = \exp(-y),$$

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

$$p(y)dy = \exp(-y)dy = dx,$$

which yields after integration

$$x(y) = P(y) = \int_0^y \exp(-y')dy' = 1 - \exp(-y),$$

or

$$y(x) = -\ln(1-x).$$

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 *ran*0.

This means that if we can factor out $\exp(-y)$ from an integrand we may have

$$I = \int_0^{\infty} F(y)dy = \int_0^{\infty} \exp(-y)G(y)dy$$

which we rewrite as

$$\int_0^{\infty} \exp(-y)G(y)dy = \int_0^1 G(y(x))dx \approx \frac{1}{N}\sum_{i=1}^{N} G(y(x_i)),$$

where x_i is a random number in the interval [0,1]. We have changed the integration limits in the second integral, since we have performed a change of variables. Since we have used the uniform distribution defined for $x \in [0,1]$, the integration limits change to 0 and 1. The variable y is now a function of x. Note also that in practical implementations, our random number generators for the uniform distribution never return exactly 0 or 1, but we may come very close.

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 *ran*0, *ran*1, *ran*2 or *ran*3 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);
.....
```

11.4.1.3 Another Example

Another function which provides an example for a PDF is

$$p(y)dy = \frac{dy}{(a+by)^n},$$

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

$$\int_0^{\infty} \frac{dy}{(a+by)^n} = \frac{1}{(n-1)ba^{n-1}},$$

gives

$$p(y)dy = \frac{(n-1)ba^{n-1}}{(a+by)^n}dy,$$

which in turn gives the cumulative function

$$x(y) = P(y) = \int_0^y \frac{(n-1)ba^{n-1}}{(a+bx)^n} dy',$$

resulting in

$$x(y) = 1 - \frac{1}{(1+b/ay)^{n-1}},$$

or

$$y = \frac{a}{b}\left((1-x)^{-1/(n-1)} - 1\right).$$

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.

11.4.1.4 Normal Distribution

For the normal distribution, expressed here as

$$g(x,y) = \exp\left(-(x^2+y^2)/2\right)dxdy.$$

it is rather difficult to find an inverse since the cumulative distribution is given by the error function $erf(x)$

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

We obviously would like to avoid computing an integral everytime we need a random variable. If we however switch to polar coordinates, we have for x and y

$$r = \left(x^2+y^2\right)^{1/2} \qquad \theta = tan^{-1}\frac{x}{y},$$

resulting in

$$g(r,\theta) = r\exp\left(-r^2/2\right)drd\theta,$$

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

$$u = \frac{1}{2}r^2,$$

and define a PDF

$$\exp\left(-u\right)du,$$

with $u \in [0,\infty)$. Using the results from example 2 for the exponential distribution, we have

$$u = -\ln\left(1-x'\right),$$

where x' is a random number generated for $x' \in [0,1]$. With

$$x = r\cos\left(\theta\right) = \sqrt{2u}\cos\left(\theta\right),$$

and

$$y = r\sin\left(\theta\right) = \sqrt{2u}\sin\left(\theta\right),$$

we can obtain new random numbers x, y through

$$x = \sqrt{-2\ln\left(1-x'\right)}\cos\left(\theta\right),$$

and

$$y = \sqrt{-2\ln(1-x')}\sin(\theta),$$

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);
.....
```

11.4.2 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

$$\int_a^b p(y)dy = 1.$$

We can rewrite our integral as

$$I = \int_a^b F(y)dy = \int_a^b p(y)\frac{F(y)}{p(y)}dy.$$

This integral resembles our discussion on the evaluation of the energy for a quantum mechanical system in Eq. (11.22).

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

$$x(y) = \int_a^y p(y')dy',$$

where we used

$$p(x)dx = dx = p(y)dy.$$

If we can invert $x(y)$, we find $y(x)$ as well.

With this change of variables we can express the integral of Eq. (11.4.2) as

$$I = \int_a^b p(y)\frac{F(y)}{p(y)}dy = \int_{\tilde{a}}^{\tilde{b}} \frac{F(y(x))}{p(y(x))}dx,$$

meaning that a Monte Carlo evaluation of the above integral gives

$$\int_{\tilde{a}}^{\tilde{b}} \frac{F(y(x))}{p(y(x))}dx = \frac{1}{N}\sum_{i=1}^{N}\frac{F(y(x_i))}{p(y(x_i))}.$$

Note the well the change in integration limits from a and b to $\tilde{a}$ and $\tilde{b}$. 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

$$\tilde{F} = \frac{F(y(x))}{p(y(x))},$$

given by

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}\left(\tilde{F}\right)^2 - \left(\frac{1}{N}\sum_{i=1}^{N}\tilde{F}\right)^2.$$

The algorithm for this procedure is

- Use the uniform distribution to find the random variable y in the interval [0,1]. The function $p(x)$ is a user provided PDF.
- Evaluate thereafter
$$I = \int_a^b F(x)dx = \int_a^b p(x)\frac{F(x)}{p(x)}dx,$$
by rewriting
$$\int_a^b p(x)\frac{F(x)}{p(x)}dx = \int_{\tilde{a}}^{\tilde{b}} \frac{F(x(y))}{p(x(y))}dy,$$
since
$$\frac{dy}{dx} = p(x).$$
- Perform then a Monte Carlo sampling for
$$\int_{\tilde{a}}^{\tilde{b}} \frac{F(x(y))}{p(x(y))}dy, \approx \frac{1}{N}\sum_{i=1}^{N}\frac{F(x(y_i))}{p(x(y_i))},$$
with $y_i \in [0,1]$,
- and evaluate the variance as well according to Eq. (11.4.2).

11.4.3 Acceptance-Rejection Method

This is a 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

$$p(x) \leq M \qquad x \in [a,b].$$

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

$$p(x) \geq s,$$

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 a closed-form expression 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 obtained 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
```

11.5 Monte Carlo Integration of Multidimensional Integrals

When we deal with multidimensional integrals of the form

$$I = \int_{a_1}^{b_1} dx_1 \int_{a_2}^{b_2} dx_2 \dots \int_{a_d}^{b_d} dx_d g(x_1, \dots, x_d),$$

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 six-dimensional integral

$$\int_{-\infty}^{\infty} \mathbf{dxdy} g(\mathbf{x}, \mathbf{y}),$$

where

$$g(\mathbf{x},\mathbf{y}) = \exp{(-\mathbf{x}^2-\mathbf{y}^2)}(\mathbf{x}-\mathbf{y})^2$$

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

$$\frac{1}{\sqrt{\pi}}\exp{(-x^2)},$$

and using this normal distribution we rewrite our integral as

$$\pi^3\int\prod_{i=1}^{6}\left(\frac{1}{\sqrt{\pi}}\exp{(-x_i^2)}\right)(\mathbf{x}-\mathbf{y})^2 dx_1 \ldots dx_6,$$

which is rewritten in a more compact form as

$$\int f(x_1,\ldots,x_d)F(x_1,\ldots,x_d)\prod_{i=1}^{6}dx_i,$$

where f is the above normal distribution and

$$F(x_1,\ldots,x_6) = F(\mathbf{x},\mathbf{y}) = (\mathbf{x}-\mathbf{y})^2,$$

Below we list two codes, one for the brute force integration and the other employing importance sampling with a gaussian distribution.

11.5.1 Brute Force Integration

http://folk.uio.no/mhjensen/compphys/programs/chapter11/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 jacobidet = 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) << jacobidet*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)
{
// 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(-xx-yy)*xy;
} // end function for the integrand
```

11.5.2 Importance Sampling

This code includes a call to the function *normal_random,* which produces random numbers from a gaussian distribution.

http://folk.uio.no/mhjensen/compphys/programs/chapter11/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 jacobidet = 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) << jacobidet*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)
{
// 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 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 11.4 Results as function of number of Monte Carlo samples N. The exact answer is $I \approx 93.020$ for the integral. The suffix *cr* stands for the brute force approach while *is* stands for the importance sampling results. All calculations use ran0 as function to generate the uniform distribution.

N	I_{cr}	I_{gd}
10000	9.92072E+01	9.33225E+01
100000	8.75039E+01	9.30042E+01
1000000	9.56759E+01	9.29988E+01
10000000	9.15446E+01	9.30203E+01

11.6 Classes for Random Number Generators

We end this chapter with presenting a possible class for using random number genrerators. The class consists of five files, one which defines the random number generators, random.h and four separate files Ran0.h, Ran1.h, Ran2.h and Ran3.h discussed in the text. We list here only the definitions contained in random.h. The file is well commented and all information is contained within the file itself.

The file random.h

```
/**
* @file Random.h
* @class Random
*
* Interface for random number generators (RNG). The particular RNG are
* implemented in the various subclasses.
*
**/

#ifndef RANDOM_H
#define RANDOM_H

class Random{
 protected:
   long seed;

 public:
   /**
   * @brief Constructor.
   *
   * @param seed_ A negative long integer. If none is given, seed takes the default value
       -1.
   **/
   Random(long seed_=-1): seed(seed_){}

   //! Destructor
   virtual ~Random();

   /**
   * This function is useful in cases where it is necessary to take care of the seed in
       order
   * to reproduce experiments with the same sequences.
   *
   * @return The seed used during the initialization of the Random Number Generator.
   **/
   long getSeed()const{return seed;}

   //! Modify the seed.
   virtual reseed(long seed_){seed = seed_;}
   /**
   * @return A random number from a particular Random Number Generator
   * implemented in the subclasses.
   **/
   virtual double sample()=0;
};
#inline Random::~Random(){}
#endif
```

11.7 Exercises

11.1. Calculate the cumulative functions $P(x)$ for the binomial and the Poisson distributions and their variances.

11.2. Make a program which computes random numbers according to the algorithm of Marsaglia and Zaman, Eqs. (11.16) and (11.17). Compute the correlation function C_k and compare with the auto-correlation function from the function $ran0$.

11.3. Make a function *normal_random* which computes random numbers for the normal distribution based on random numbers generated from the function $ran0$.

11.4. 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 $ran0$.

11.5. 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.

2. 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.

11.6. In this exercise 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.

1. Find closed form solutions for the above equations assuming continuous variables and setting the number of ^{210}Po nuclei equal zero at $t = 0$.
2. 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.
3. 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 closed form solutions for $N_X(0) = 10$, 100 and 1000.

11.7. The task here 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. 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 a closed form 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

$$\langle \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \rangle = \int_{-\infty}^{\infty} d\mathbf{r}_1 d\mathbf{r}_2 e^{-2\alpha(r_1+r_2)} \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|}. \tag{11.23}$$

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.

1. Set up a program which performs a Monte Carlo integration of the above integral, but without using importance sampling. That is, use only the uniform distribution. An example of a program which implements this can be written as

```
    double int_mc = 0.; double variance = 0.;
    double sum_sigma= 0. ; long idum=-1 ;
    double length=1.5; // we fix the max size of the box to L=3
    double jacobidet=pow((2*length),6.);

//  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++) {
        // Maps U[0,1] to U[-L,L]
        x[j]=-length+2*length*ran0(&idum);
     }
     fx=brute_force_MC(x);
     int_mc += fx;
     sum_sigma += fx*fx;
    }
    int_mc = jacobidet*int_mc/((double) n );
    sum_sigma = jacobidet*sum_sigma/((double) n );
    variance=sum_sigma-int_mc*int_mc;
    ....
```

We include also an example of a function which sets up the function to integrate

```
double brute_force_MC(double *x)
{
  double alpha = 2.;
// evaluate the different terms of the exponential
  double exp1=-2*alpha*sqrt(x[0]*x[0]+x[1]*x[1]+x[2]*x[2]);
  double exp2=-2*alpha*sqrt(x[3]*x[3]+x[4]*x[4]+x[5]*x[5]);
  double deno=sqrt(pow((x[0]-x[3]),2)
       +pow((x[1]-x[4]),2)+pow((x[2]-x[5]),2));
  double value=exp(exp1+exp2)/deno;
 return value;
} // end function for the integrand
```

2. 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. An extract from a code which performs the importance sampling is included here.

```
  double int_mc = 0.; double variance = 0.;
  double sum_sigma= 0. ; long idum=-1 ;
// The 'volume' contains 4 jacobideterminants(pi,pi,2pi,2pi)
// and a scaling factor 1/16
  double jacobidet=4*pow(acos(-1.),4.)*1./16;
//  evaluate the integral with importance sampling
  for ( int i = 1; i <= n; i++){
    for (int j = 0; j < 2; j++) {
   y=ran0(&idum);
   x[j]=-0.25*log(1.-y);
    }
    for (int j = 2; j < 4; j++) {
   x[j] = 2*acos(-1.)*ran0(&idum);
    }
    for (int j = 4; j < 6; j++) {
     x[j] = acos(-1.)*ran0(&idum);
     }
  fx=integrand_MC(x);
      ....
```

The importance sampling improves considerably the results, as we noted in the example with the normal distribution. Typical results are

Table 11.5 Results obtained with the uniform distribution only and importance sampling. The suffix *ud* stands for the approach with the uniform distribution while *is* stands for the use of importance sampling.

N	I_{ud}	σ_{ud}	time(s)	I_{is}	σ_{is}	time(s)
1E6	0.19238	3.85124E-4	0.6	0.19176	1.01515E-4	1.4
10E6	0.18607	1.18053E-4	6	0.192254	1.22430E-4	14
100E6	0.18846	4.37163E-4	57	0.192720	1.03346E-4	138
1000E6	0.18843	1.35879E-4	581	0.192789	3.28795E-5	1372

3. Parallelize your code from the previous point and compare the CPU time needed with that from the first point above. Do you achieve a good speedup?
4. The integral of Eq. (11.23) has a closed form solution. Can you find it?

Chapter 12
Random walks and the Metropolis algorithm

The way that can be spoken of is not the constant way. (Tao Te Ching, Book I, I.1) *Lao Tzu*

Abstract We present the theory of random walks, Markov chains and present the Metropolis algorithm.

12.1 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 from this main area of applications. The focus is on diffusion and random walks. Furthermore, we will use these topics to derive the famous Metropolis algorithm.

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.

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 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 12.5. 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 12.3 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.

12.2 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. [63, 69].

12.2.1 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

$$j(x,t) = -D\frac{\partial w(x,t)}{\partial x},$$

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

$$\frac{\partial j(x,t)}{\partial x} = -\frac{\partial w(x,t)}{\partial t},$$

which leads to

$$\frac{\partial w(x,t)}{\partial t} = D\frac{\partial^2 w(x,t)}{\partial x^2}, \tag{12.1}$$

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

$$\langle x(t)\rangle = \int_{-\infty}^{\infty} xw(x,t)dx,$$

or

$$\langle x^2(t)\rangle = \int_{-\infty}^{\infty} x^2 w(x,t)dx,$$

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

$$\langle f(x,t)\rangle = \int_{-\infty}^{\infty} f(x,t)w(x,t)dx.$$

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

$$\int_{-\infty}^{\infty} w(x,t)dx = 1$$

imposes significant constraints on $w(x,t)$. These are

$$w(x=\pm\infty,t) = 0 \qquad \frac{\partial^n w(x,t)}{\partial x^n}|_{x=\pm\infty} = 0,$$

implying that when we study the time-derivative $\partial\langle x(t)\rangle/\partial t$, we obtain after integration by parts and using Eq. (12.1)

$$\frac{\partial\langle x\rangle}{\partial t} = \int_{-\infty}^{\infty} x\frac{\partial w(x,t)}{\partial t}dx = D\int_{-\infty}^{\infty} x\frac{\partial^2 w(x,t)}{\partial x^2}dx,$$

leading to

$$\frac{\partial\langle x\rangle}{\partial t} = Dx\frac{\partial w(x,t)}{\partial x}|_{x=\pm\infty} - D\int_{-\infty}^{\infty}\frac{\partial w(x,t)}{\partial x}dx,$$

implying that

$$\frac{\partial\langle x\rangle}{\partial t} = 0.$$

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

$$\frac{\partial\langle x^2\rangle}{\partial t} = Dx^2\frac{\partial w(x,t)}{\partial x}|_{x=\pm\infty} - 2D\int_{-\infty}^{\infty} x\frac{\partial w(x,t)}{\partial x}dx,$$

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

$$\frac{\partial \langle x^2\rangle}{\partial t} = -Dxw(x,t)|_{x=\pm\infty} + 2D\int_{-\infty}^{\infty} w(x,t)dx = 2D,$$

leading to

$$\langle x^2\rangle = 2Dt,$$

and the variance as

$$\langle x^2\rangle - \langle x\rangle^2 = 2Dt. \tag{12.2}$$

The root mean square displacement after a time t is then

$$\sqrt{\langle x^2\rangle - \langle x\rangle^2} = \sqrt{2Dt}.$$

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. 12.1 we have assumed that our distribution is given by a normal distribution with variance $\sigma^2 = 2Dt$, centered at $x = 0$. The distribution reads

$$w(x,t)dx = \frac{1}{\sqrt{4\pi Dt}}\exp{(-\frac{x^2}{4Dt})}dx.$$

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. 12.1 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.

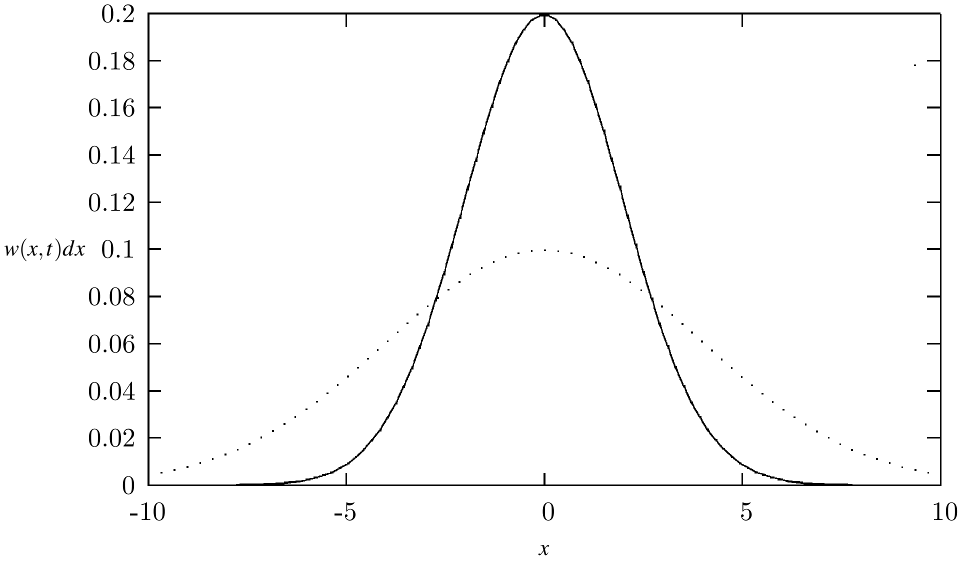


Fig. 12.1 Time development of a normal distribution with variance $\sigma^2 = 2Dt$ and with $D = 1\mathrm{m}^2/\mathrm{s}$. The solid line represents the distribution at $t = 2$s while the dotted line stands for $t = 8$s.

12.2.2 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. 12.2. 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

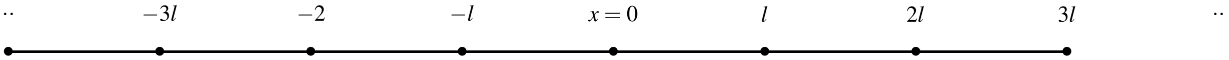


Fig. 12.2 One-dimensional walker which can jump either to the left or to the right. Every step has length $\Delta x = l$.

$$\langle x(n)\rangle = \sum_i^n \Delta x_i = 0 \qquad \Delta x_i = \pm l,$$

since we have an equal probability of jumping either to the left or to right. The value of $\langle x(n)^2\rangle$ is

$$\langle x(n)^2\rangle = \left(\sum_i^n \Delta x_i\right)\left(\sum_j^n \Delta x_j\right) = \sum_i^n \Delta x_i^2 + \sum_{i\neq j}^n \Delta x_i \Delta x_j = l^2 n.$$

For many enough steps the non-diagonal contribution is

$$\sum_{i\neq j}^N \Delta x_i \Delta x_j = 0,$$

since $\Delta x_{i,j} = \pm l$. The variance is then

$$\langle x(n)^2\rangle - \langle x(n)\rangle^2 = l^2 n. \qquad (12.3)$$

It is also rather straightforward to compute the variance for $L \neq R$. The result is

$$\langle x(n)^2\rangle - \langle x(n)\rangle^2 = 4LRl^2 n.$$

In Eq. (12.3) 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. (12.2) by defining the diffusion constant as

$$D = \frac{l^2}{\Delta t}.$$

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/chapter12/program2.cpp

`http://folk.uio.no/mhjensen/compphys/programs/chapter12/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. 12.3 shows that the variance increases linearly as function of the number of time steps, as expected from the closed-form results. Similarly, the mean displacement in Fig. 12.4 oscillates around zero.

12.3 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

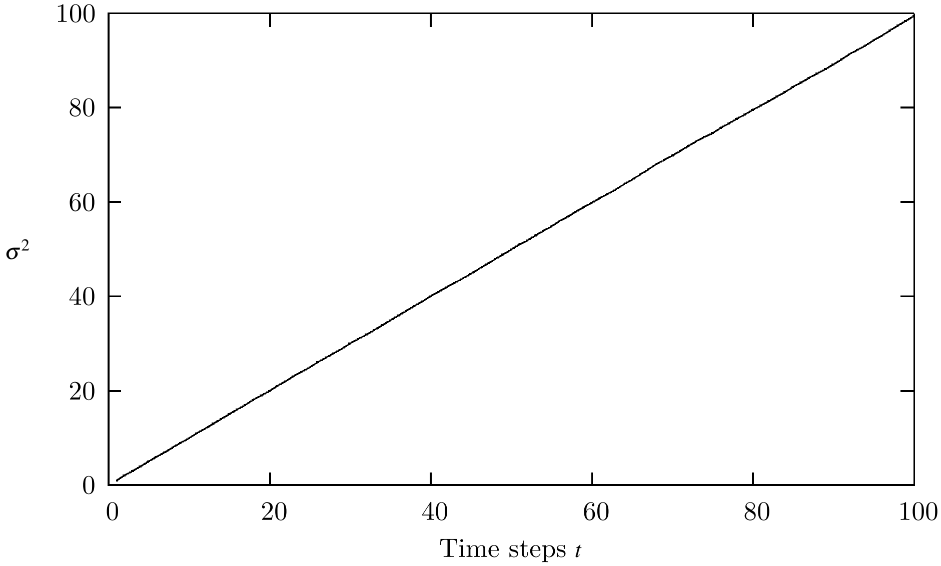


Fig. 12.3 Time development of σ^2 for a random walker. 100000 Monte Carlo samples were used with the function ran1 and a seed set to -1.

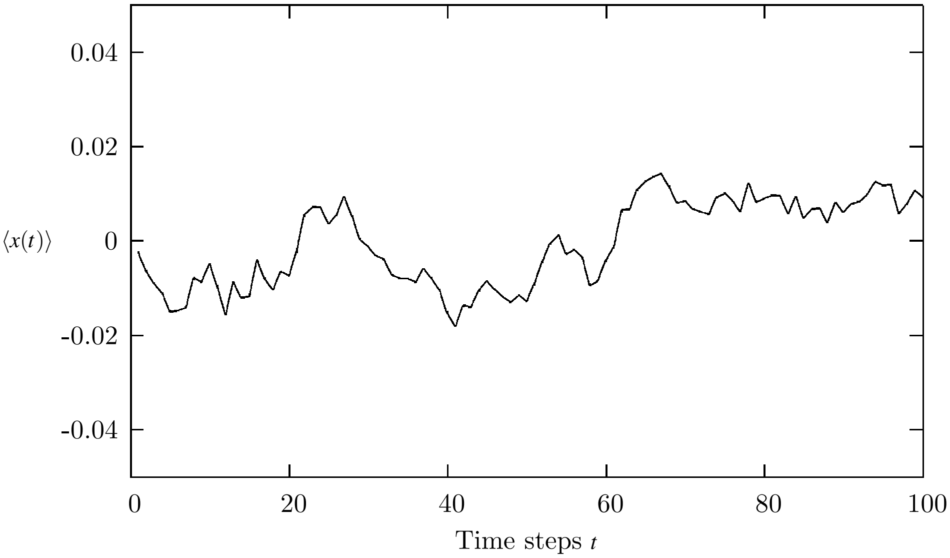


Fig. 12.4 Time development of $\langle x(t)\rangle$ for a random walker. 100000 Monte Carlo samples were used with the function ran1 and a seed set to -1.

$$\frac{\partial w(x,t)}{\partial t} \approx \frac{w(i,n+1)-w(i,n)}{\Delta t},$$

whereas the gradient is approximated as

$$D\frac{\partial^2 w(x,t)}{\partial x^2} \approx D\frac{w(i+1,n)+w(i-1,n)-2w(i,n)}{(\Delta x)^2},$$

resulting in the discretized diffusion equation

$$\frac{w(i,n+1)-w(i,n)}{\Delta t}=D\frac{w(i+1,n)+w(i-1,n)-2w(i,n)}{(\Delta x)^2},$$

where n represents a given time step and i a step in the x-direction. The solution of such equations is discussed in our chapter on partial differential equations, see Chapter 10. The aim here is to show 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.

12.3.1 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 independent, 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 $\Delta t = \varepsilon$, 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. 12.2, 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

$$W_{ij}(\varepsilon)=W(il-jl,\varepsilon)=\begin{cases}\frac{1}{2} & |i-j|=1\\ 0 & \text{else}\end{cases},$$

where W_{ij} is normally called the transition probability and we can represent it, see below, as a matrix. Note that this matrix is not a stochastic matrix as long as it is a finite matrix. Our new PDF $w_i(t=\varepsilon)$ is now related to the PDF at $t = 0$ through the relation

$$w_i(t=\varepsilon)=\sum_j W(j\rightarrow i)w_j(t=0).$$

This equation represents the discretized time-development of an original PDF. It is a microscopic way of representing the process shown in Fig. 12.1. Since both W and w represent probabilities, they have to be normalized, i.e., we require that at each time step we have

$$\sum_i w_i(t)=1,$$

and

$$\sum_j W(j\rightarrow i)=1,$$

which applies for all j-values. 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 $t_n = n\varepsilon$ our initial distribution has developed into

$$w_i(t_n) = \sum_j W_{ij}(t_n) w_j(0),$$

and defining

$$W(il - jl, n\varepsilon) = (W^n(\varepsilon))_{ij}$$

we obtain

$$w_i(n\varepsilon) = \sum_j (W^n(\varepsilon))_{ij} w_j(0),$$

or in matrix form

$$\hat{w}(n\varepsilon) = \hat{W}^n(\varepsilon)\hat{w}(0). \tag{12.4}$$

The matrix $\hat{W}$ can be written in terms of two matrices

$$\hat{W} = \frac{1}{2}\left(\hat{L} + \hat{R}\right),$$

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

$$\hat{R} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

and

$$\hat{L} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

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

$$\hat{L}\hat{R} = \hat{R}\hat{L} = I, \tag{12.5}$$

which applies in the case of infinite matrices and

$$\hat{L} = \hat{R}^{-1} \tag{12.6}$$

To see that $\hat{L}\hat{R} = \hat{R}\hat{L} = 1$, perform e.g., the matrix multiplication

$$\hat{L}\hat{R} = \sum_k \hat{L}_{ik}\hat{R}_{kj} = \sum_k \delta_{(i+1),k}\delta_{k,(j+1)} = \delta_{i+1,j+1} = \delta_{i,j},$$

and only the diagonal matrix elements are different from zero.

For the first time step we have thus

$$\hat{W} = \frac{1}{2}\left(\hat{L} + \hat{R}\right),$$

and using the properties in Eqs. (12.5) and (12.6) we have after two time steps

$$\hat{W}^2(2\varepsilon) = \frac{1}{4}\left(\hat{L}^2 + \hat{R}^2 + 2\hat{R}\hat{L}\right),$$

and similarly after three time steps

$$\hat{W}^3(3\varepsilon) = \frac{1}{8}\left(\hat{L}^3 + \hat{R}^3 + 3\hat{R}\hat{L}^2 + 3\hat{R}^2\hat{L}\right).$$

Using the binomial formula

$$\sum_{k=0}^{n}\binom{n}{k}\hat{a}^k\hat{b}^{n-k}=(a+b)^n,$$

ee we have that the transition matrix after n time steps can be written as

$$\hat{W}^n(n\varepsilon))=\frac{1}{2^n}\sum_{k=0}^{n}\binom{n}{k}\hat{R}^k\hat{L}^{n-k},$$

or

$$\hat{W}^n(n\varepsilon))=\frac{1}{2^n}\sum_{k=0}^{n}\binom{n}{k}\hat{L}^{n-2k}=\frac{1}{2^n}\sum_{k=0}^{n}\binom{n}{k}\hat{R}^{2k-n},$$

and using $R_{ij}^m=\delta_{i,(j+m)}$ and $L_{ij}^m=\delta_{(i+m),j}$ we arrive at

$$W(il-jl,n\varepsilon)=\begin{cases}\frac{1}{2^n}\binom{n}{\frac{1}{2}(n+i-j)} & |i-j|\leq n\\ 0 & \text{else}\end{cases}, \tag{12.7}$$

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. (12.4) we have

$$w_i(n\varepsilon)=\sum_j(W^n(\varepsilon))_{ij}w_j(0)=\sum_j\frac{1}{2^n}\binom{n}{\frac{1}{2}(n+i-j)}\delta_{j,0},$$

resulting in

$$w_i(n\varepsilon)=\frac{1}{2^n}\binom{n}{\frac{1}{2}(n+i)}\qquad |i|\leq n.$$

We can then use the recursion relation for the binomials

$$\binom{n+1}{\frac{1}{2}(n+1+i)}=\binom{n}{\frac{1}{2}(n+i+1)}+\binom{n}{\frac{1}{2}(n+i-1)} \tag{12.8}$$

to obtain the discretized diffusion equation. In order to achieve this, we define $x=il$, where l and i are integers, and $t=n\varepsilon$. We can then rewrite the probability distribution as

$$w(x,t)=w(il,n\varepsilon)=w_i(n\varepsilon)=\frac{1}{2^n}\binom{n}{\frac{1}{2}(n+i)}\qquad |i|\leq n,$$

and rewrite Eq. (12.8) as

$$w(x,t+\varepsilon)=\frac{1}{2}w(x+l,t)+\frac{1}{2}w(x-l,t).$$

Adding and subtracting $w(x,t)$ and multiplying both sides with l^2/ε we have

$$\frac{w(x,t+\varepsilon)-w(x,t)}{\varepsilon}=\frac{l^2}{2\varepsilon}\frac{w(x+l,t)-2w(x,t)+w(x-l,t)}{l^2}.$$

If we identify $D = l^2/2\varepsilon$ and $l = \Delta x$ and $\varepsilon = \Delta t$ 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.

12.3.1.1 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 4×4 matrix $\hat{W}$

$$\hat{W}=\begin{pmatrix} 1/4 & 1/9 & 3/8 & 1/3 \\ 2/4 & 2/9 & 0 & 1/3 \\ 0 & 1/9 & 3/8 & 0 \\ 1/4 & 5/9 & 2/8 & 1/3 \end{pmatrix},$$

and we choose our initial state as

$$\hat{w}(t=0)=\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

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. Furthermore, the largest eigenvalue is one. We act then on $\hat{w}$ with $\hat{W}$. The first iteration is

$$\hat{w}(t=\varepsilon)=\hat{W}\hat{w}(t=0),$$

resulting in

$$\hat{w}(t=\varepsilon)=\begin{pmatrix} 1/4 \\ 1/2 \\ 0.0 \\ 1/4 \end{pmatrix}.$$

The next iteration results in

$$\hat{w}(t=2\varepsilon)=\hat{W}\hat{w}(t=\varepsilon),$$

resulting in

$$\hat{w}(t=2\varepsilon)=\begin{pmatrix} 0.201389 \\ 0.319444 \\ 0.055556 \\ 0.423611 \end{pmatrix}.$$

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

$$\begin{aligned}
W_{11}w_1(t=\infty)+W_{12}w_2(t=\infty)+W_{13}w_3(t=\infty)+W_{14}w_4(t=\infty)&=w_1(t=\infty)\\
W_{21}w_1(t=\infty)+W_{22}w_2(t=\infty)+W_{23}w_3(t=\infty)+W_{24}w_4(t=\infty)&=w_2(t=\infty)\\
W_{31}w_1(t=\infty)+W_{32}w_2(t=\infty)+W_{33}w_3(t=\infty)+W_{34}w_4(t=\infty)&=w_3(t=\infty)\\
W_{41}w_1(t=\infty)+W_{42}w_2(t=\infty)+W_{43}w_3(t=\infty)+W_{44}w_4(t=\infty)&=w_4(t=\infty)
\end{aligned} \tag{12.9}$$

with the constraint that

$$\sum_i w_i(t=\infty)=1,$$

yielding as solution

$$\hat{w}(t=\infty)=\begin{pmatrix}0.244318\\0.319602\\0.056818\\0.379261\end{pmatrix}.$$

Table 12.1 demonstrates the convergence as a function of the number of iterations or time steps. After twelve iterations we have reached the exact value with six leading digits.

Table 12.1 Convergence to the steady state as function of number of iterations.

Iteration	w_1	w_2	w_3	w_4
0	1.000000	0.000000	0.000000	0.000000
1	0.250000	0.500000	0.000000	0.250000
2	0.201389	0.319444	0.055556	0.423611
3	0.247878	0.312886	0.056327	0.382909
4	0.245494	0.321106	0.055888	0.377513
5	0.243847	0.319941	0.056636	0.379575
6	0.244274	0.319547	0.056788	0.379391
7	0.244333	0.319611	0.056801	0.379255
8	0.244314	0.319610	0.056813	0.379264
9	0.244317	0.319603	0.056817	0.379264
10	0.244318	0.319602	0.056818	0.379262
11	0.244318	0.319602	0.056818	0.379261
12	0.244318	0.319602	0.056818	0.379261
$\hat{w}(t=\infty)$	0.244318	0.319602	0.056818	0.379261

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.

12.3.2 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 $\Delta t = \varepsilon$ is

$$w_i(t) = \sum_j W(j \to i) w_j(t=0).$$

The continuous analog to $w_i(0)$ is

$$w(\mathbf{x}) \to \delta(\mathbf{x}), \tag{12.10}$$

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

$$w(\mathbf{y}, t+\Delta t) = \int W(\mathbf{y}, \mathbf{x}, \Delta t) w(\mathbf{x}, t) d\mathbf{x},$$

and after m time steps we have

$$w(\mathbf{y}, t+m\Delta t) = \int W(\mathbf{y}, \mathbf{x}, m\Delta t) w(\mathbf{x}, t) d\mathbf{x}.$$

When equilibrium is reached we have

$$w(\mathbf{y}) = \int W(\mathbf{y}, \mathbf{x}, t) w(\mathbf{x}) d\mathbf{x}.$$

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

$$w(\mathbf{x}, t) = \int_{-\infty}^{\infty} d\mathbf{k} \exp(i\mathbf{k}\mathbf{x}) \tilde{w}(\mathbf{k}, t), \tag{12.11}$$

and using the definition of the δ-function

$$\delta(\mathbf{x}) = \frac{1}{2\pi} \int_{-\infty}^{\infty} d\mathbf{k} \exp(i\mathbf{k}\mathbf{x}),$$

we see that

$$\tilde{w}(\mathbf{k}, 0) = 1/2\pi.$$

We can then use the Fourier-transformed diffusion equation

$$\frac{\partial \tilde{w}(\mathbf{k}, t)}{\partial t} = -D\mathbf{k}^2 \tilde{w}(\mathbf{k}, t), \tag{12.12}$$

with the obvious solution

$$\tilde{w}(\mathbf{k}, t) = \tilde{w}(\mathbf{k}, 0) \exp\left[-(D\mathbf{k}^2 t)\right) = \frac{1}{2\pi} \exp\left[-(D\mathbf{k}^2 t)\right].$$

Using Eq. (12.11) we obtain

$$w(\mathbf{x},t) = \int_{-\infty}^{\infty} d\mathbf{k} \exp\left[i\mathbf{k}\mathbf{x}\right] \frac{1}{2\pi} \exp\left[-(D\mathbf{k}^2 t)\right] = \frac{1}{\sqrt{4\pi D t}} \exp\left[-(\mathbf{x}^2/4Dt)\right], \tag{12.13}$$

with the normalization condition

$$\int_{-\infty}^{\infty} w(\mathbf{x},t) d\mathbf{x} = 1.$$

It is rather easy to verify by insertion that Eq. (12.13) 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. (12.7). The results from the central limit theorem, see Sect. 11.2.2, state that transition probability in the limit $n \to \infty$ converges to the normal distribution. It is then possible to show that

$$W(il - jl, n\varepsilon) \to W(\mathbf{y}, \mathbf{x}, \Delta t) = \frac{1}{\sqrt{4\pi D \Delta t}} \exp\left[-((\mathbf{y}-\mathbf{x})^2/4D\Delta t)\right],$$

and that it satisfies the normalization condition and is itself a solution to the diffusion equation.

12.3.3 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://folk.uio.no/mhjensen/compphys/programs/chapter12/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. 12.5 shows the resulting probability distribution after n steps In Fig. 12.5 we have plotted the probability distribution function after a given number of time steps. Do you recognize the shape of the probabiliy distributions?

12.4 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 purpose 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/chapter12/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

$$S = -\sum_i w_i ln(w_i),$$

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 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. 12.6. The code used to compute these results is in programs/chapter12/program4.cpp. Here we have used 100 walkers on a lattice of length from $L = -50$ to $L = 50$ employing periodic

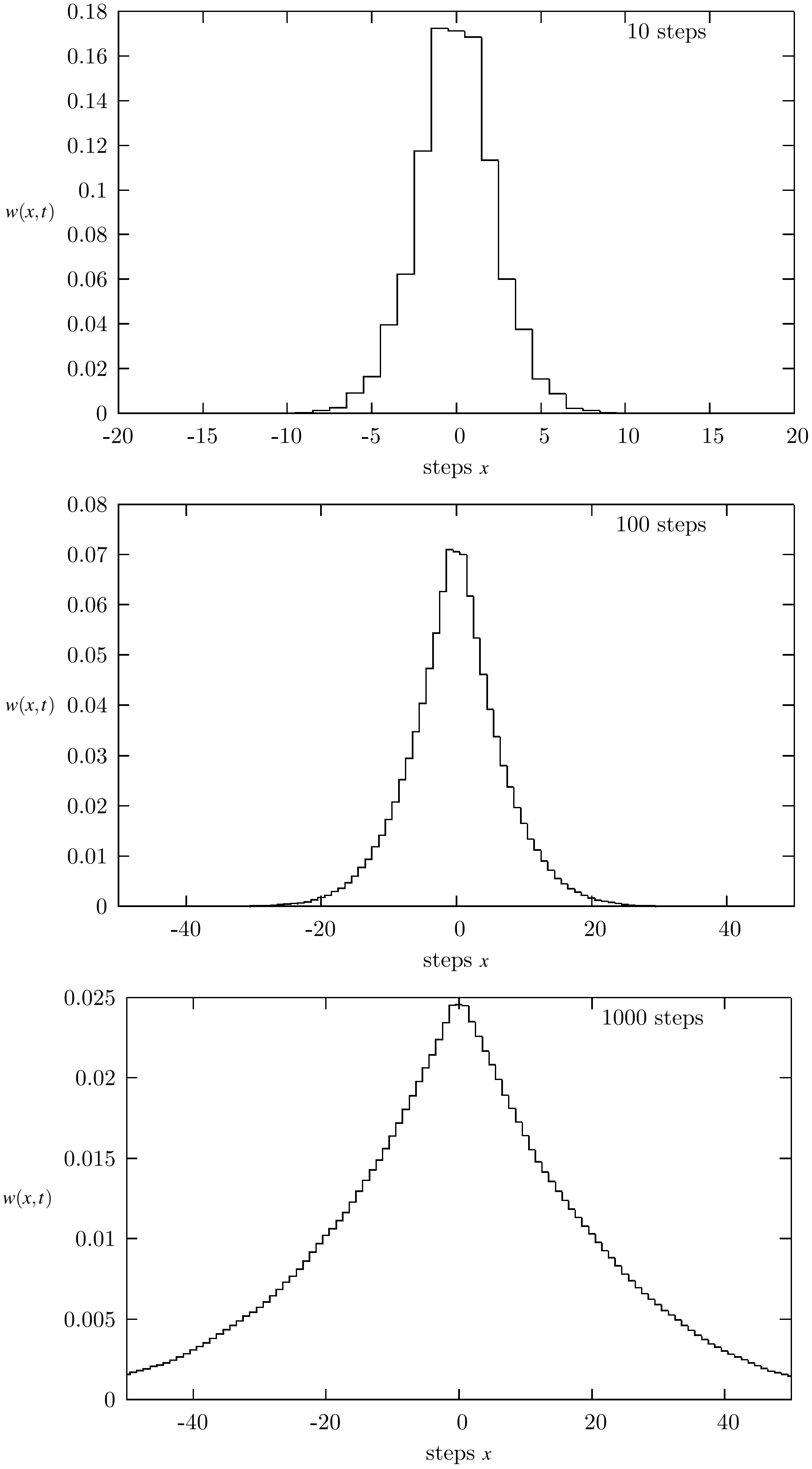


Fig. 12.5 Probability distribution for one walker after 10, 100 and 1000 steps.

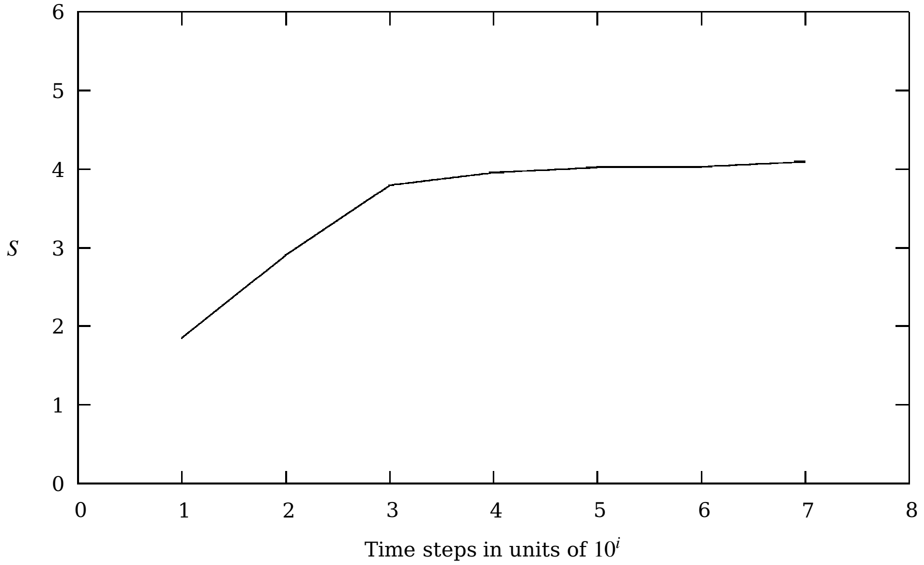


Fig. 12.6 Entropy S_j as function of number of time steps j for a random walk in one dimension. Here we have used 100 walkers on a lattice of length from $L = -50$ to $L = 50$ employing periodic boundary conditions meaning that if a walker reaches the point $x = L+1$ it is shifted to $x = -L$ and if $x = -L$ it is shifted to $x = L$.

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. 12.6 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. For a discussion of the ergodicity hypothesis and the Metropoli algorithm, see for example Ref. [63]. 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.

12.5 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
$$w_i(t=\varepsilon)=W(j\rightarrow i)w_j(t=0).$$
This equation represents the discretized time-development of an original PDF. We can rewrite this as a
$$w_i(t=\varepsilon)=W_{ij}w_j(t=0).$$
with the transition matrix W for a random walk given by
$$W_{ij}(\varepsilon)=W(il-jl,\varepsilon)=\begin{cases}\frac{1}{2} & |i-j|=1\\ 0 & \text{else}\end{cases}$$
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 at each time step we have
$$\sum_i w_i(t)=1,$$
and
$$\sum_j W(j\rightarrow i)=1.$$
Here we have written the previous matrix $W_{ij}=W(j\rightarrow i)$. 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
$$w_i(t+1)=\sum_j W_{ij}w_j(t),$$
or as vector-matrix relation
$$\hat{\mathbf{w}}(t+1)=\hat{\mathbf{W}}\hat{\mathbf{w}}(t),$$
and if we have that $||\hat{\mathbf{w}}(t+1)-\hat{\mathbf{w}}(t)||\rightarrow 0$, 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
$$\mathbf{w}(t=\infty)=\mathbf{W}\mathbf{w}(t=\infty). \qquad (12.14)$$

In most situations, the transition probability $W_{ij}=W(j\rightarrow i)$ is not known[1]. It can represent a complicated set of chemical reactions which we are not capable of modeling or, we are able to write down and account for all the boundary and the initial conditions needed to describe $W(j\rightarrow i)$. A Markov chain is a process where this probability is in general unknown. The question then is how can we model anything under such a severe lack of knowledge? The Metropolis algorithm comes to our rescue here. Since $W(j\rightarrow i)$ is unknown, we model it as the product of two probabilities, a probability for accepting the proposed move from the state j to the state j, and a probability for making the transition to the state i being in the state j. We label these probabilities $A(j\rightarrow i)$ and $T(j\rightarrow i)$, respectively. Our total transition probability is then
$$W(j\rightarrow i)=T(j\rightarrow i)A(j\rightarrow i).$$

[1] Note that the discrete equations here can easily be replaced by continuous ones.

The algorithm can then be expressed as

- We make a suggested move to the new state i with some transition or moving probability $T_{j\rightarrow i}$.
- We accept this move to the new state with an acceptance probability $A_{j\rightarrow i}$. The new state i is in turn used as our new starting point for the next move. We reject this proposed moved with a $1-A_{j\rightarrow i}$ and the original state j is used again as a sample.

We wish to derive the required properties of the probabilities T and A such that $w_i^{(t\rightarrow\infty)} \rightarrow w_i$, starting from any distribution, will lead us to the correct distribution.

We can now derive the dynamical process towards equilibrium. To obtain this equation we note that after t time steps the probability for being in a state i is related to the probability of being in a state j and performing a transition to the new state together with the probability of actually being in the state i and making a move to any of the possible states j from the previous time step. We can express this as, assuming that T and A are time-independent,

$$w_i(t+1) = \sum_j \left[w_j(t)T_{j\rightarrow i}A_{j\rightarrow i} + w_i(t)T_{i\rightarrow j}\left(1-A_{i\rightarrow j}\right)\right].$$

All probabilities are normalized, meaning that $\sum_j T_{i\rightarrow j} = 1$. Using the latter, we can rewrite the previous equation as

$$w_i(t+1) = w_i(t) + \sum_j \left[w_j(t)T_{j\rightarrow i}A_{j\rightarrow i} - w_i(t)T_{i\rightarrow j}A_{i\rightarrow j}\right],$$

which can be rewritten as

$$w_i(t+1) - w_i(t) = \sum_j \left[w_j(t)T_{j\rightarrow i}A_{j\rightarrow i} - w_i(t)T_{i\rightarrow j}A_{i\rightarrow j}\right].$$

This equation is very similar to the so-called Master equation, which relates the temporal dependence of a PDF $w_i(t)$ to various transition rates. The equation can be derived from the so-called Chapman-Einstein-Enskog-Kolmogorov equation, see for example Ref. [70]. The equation is given as

$$\frac{dw_i(t)}{dt} = \sum_j \left[W(j\rightarrow i)w_j - W(i\rightarrow j)w_i\right], \tag{12.15}$$

which simply states that the rate at which the systems moves from a state j to a final state i (the first term on the right-hand side of the last equation) is balanced by the rate at which the system undergoes transitions from the state i to a state j (the second term). If we have reached the so-called steady state, then the temporal development is zero since we are now satisfying Eq. (12.5). This means that in equilibrium we have

$$\frac{dw_i(t)}{dt} = 0.$$

In the limit $t\rightarrow\infty$ we require that the two distributions $w_i(t+1) = w_i$ and $w_i(t) = w_i$ and we have

$$\sum_j w_j T_{j\rightarrow i}A_{j\rightarrow i} = \sum_j w_i T_{i\rightarrow j}A_{i\rightarrow j},$$

which is the condition for balance when the most likely state (or steady state) has been reached. We see also that the right-hand side can be rewritten as

$$\sum_j w_i T_{i\rightarrow j}A_{i\rightarrow j} = \sum_j w_i W_{i\rightarrow j},$$

and using the property that $\sum_j W_{i\rightarrow j} = 1$, we can rewrite our equation as

$$w_i = \sum_j w_j T_{j\to i} A_{j\to i} = \sum_j w_j W_{j\to i},$$

which is nothing but the standard equation for a Markov chain when the steady state has been reached.

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 may risk to end up with so-called cyclic solutions. To avoid this we therefore introduce an additional condition, namely that of detailed balance

$$W(j \to i) w_j = W(i \to j) w_i.$$

These equations were derived by Lars Onsager when studying irreversible processes, see Ref. [71]. At equilibrium detailed balance gives thus

$$\frac{W(j \to i)}{W(i \to j)} = \frac{w_i}{w_j}.$$

Rewriting the last equation in terms of our transition probabilities T and acceptance probobalities A we obtain

$$w_j(t) T_{j\to i} A_{j\to i} = w_i(t) T_{i\to j} A_{i\to j}.$$

Since we normally have an expression for the probability distribution functions w_i, we can rewrite the last equation as

$$\frac{T_{j\to i} A_{j\to i}}{T_{i\to j} A_{i\to j}} = \frac{w_i}{w_j}.$$

In statistical physics this condition ensures that it is e.g., the Boltzmann distribution which is generated when equilibrium is reached.

We introduce now the Boltzmann distribution

$$w_i = \frac{\exp(-\beta(E_i))}{Z},$$

which states that the probability of finding the system in a state i with energy E_i at an inverse temperature $\beta = 1/k_B T$ is $w_i \propto \exp(-\beta(E_i))$. 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

$$Z = \sum_j \exp(-\beta(E_i)).$$

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 a position where we can generate expectation values for a given variable A through the definition

$$\langle A \rangle = \sum_j A_j w_j = \frac{\sum_j A_j \exp(-\beta(E_j)}{Z}.$$

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

$$\frac{w_i}{w_j} = \exp\left(-\beta(E_i - E_j)\right).$$

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

$$\begin{array}{cccccccccc} \uparrow & \uparrow & \uparrow & \dots & \uparrow & \downarrow & \uparrow & \dots & \uparrow & \downarrow \\ 1 & 2 & 3 & \dots & k-1 & k & k+1 & \dots & N-1 & N \end{array}$$

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, when equilibrium is established, in the Boltzmann distribution. Else, our algorithm for selecting microstates is wrong.

As stated above, we do in general not know the closed-form expression of the transition rate and we are free to model it as $W(i \to j) = T(i \to j)A(i \to j)$. Our ratio between probabilities gives us

$$\frac{A_{j\to i}}{A_{i\to j}} = \frac{w_i T_{i\to j}}{w_j T_{j\to i}}.$$

The simplest form of the Metropolis algorithm (sometimes called for brute force Metropolis) assumes that the transition probability $T(i \to j)$ is symmetric, implying that $T(i \to j) = T(j \to i)$. We obtain then (using the Boltzmann distribution)

$$\frac{A(j \to i)}{A(i \to j)} = \exp\left(-\beta(E_i - E_j)\right).$$

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. The problem with our ratio

$$\frac{A(j\rightarrow i)}{A(i\rightarrow j)} = \exp\left(-\beta(E_i - E_j)\right),$$

is that we do not know the acceptance probability. This equation only specifies the ratio of pairs of probabilities. Normally we want an algorithm which is as efficient as possible and maximizes the number of accepted moves. Moreover, we know that the acceptance probability has 0 as its smallest value and 1 as its largest. If we assume that the largest possible acceptance probability is 1, we adjust thereafter the other acceptance probability to this constraint.

To understand this better, assume that we have two energies, E_i and E_j, with $E_i < E_j$. This means that the largest acceptance value must be $A(j\rightarrow i)$ since we move to a state with lower energy. It follows from also from the fact that the probability w_i is larger than w_j. The trick then is to fix this value to $A(j\rightarrow i) = 1$. It means that the other acceptance probability has to be

$$A(i\rightarrow j) = \exp\left(-\beta(E_j - E_i)\right).$$

One possible way to encode this equation reads

$$A(j\rightarrow i) = \left\{ \begin{array}{cc} \exp\left(-\beta(E_i - E_j)\right) & E_i - E_j > 0 \\ 1 & else \end{array} \right.,$$

implying that if we move to a state with a lower energy, we always accept this move with acceptance probability $A(j\rightarrow i) = 1$. If the energy is higher, we need to check this acceptance probability with the ratio between the probabilities from our PDF. From a practical point of view, the above ratio is compared with a random number. If the ratio is smaller than a given random number we accept the move to a higher energy, else we stay in the same state.

Nothing hinders us obviously in choosing another acceptance ratio, like a weighting of the two energies via

$$A(j\rightarrow i) = \exp\left(-\frac{1}{2}\beta(E_i - E_j)\right).$$

However, it is easy to see that such an acceptance ratio woud result in fewer accepted moves.

12.5.1 Brief Summary

The Monte Carlo approach, combined with the theory for Markov chains can be summarized as follows: A Markov chain Monte Carlo method for the simulation of a distribution w is any method producing an ergodic Markov chain of events x whose stationary distribution is w. The Metropolis algorithm can be phrased as

- Generate an initial value $x^{(i)}$.
- Generate a trial value y_t with probability $T(y_t|x^{(i)})$. The latter quantity represents the probability of generating y_t given $x^{(i)}$.
- Take a new value

$$x^{(i+1)} = \left\{ \begin{array}{cc} y_t & \text{with probability} = A(x^{(i)}\rightarrow y_t) \\ x^{(i)} & \text{with probability} = 1 - A(x^{(i)}\rightarrow y_t) \end{array} \right.$$

- We have defined the transition (acceptance) probability as

$$A(x\rightarrow y) = \min\left\{ \frac{w(y)T(x|y)}{w(x)T(y|x)}, 1 \right\}.$$

The distribution f is often called the instrumental (we will relate it to the jumping of a walker) or proposal distribution while A is the Metropolis-Hastings acceptance probability. When $T(y|x)$ is symmetric it is just called the Metropolis algorithm.

Using the Metropolis algorithm we can in turn set up the general calculational scheme as shown in Fig. 12.7.

The dynamical equation can be written as

$$w_i(t+1) = \sum_j M_{ij} w_j(t) \tag{12.16}$$

with the matrix M given by

$$M_{ij} = \delta_{ij} \left[1 - \sum_k T_{i\to k} A_{i\to k} \right] + T_{j\to i} A_{j\to i}\,. \tag{12.17}$$

Summing over i shows that $\sum_i M_{ij} = 1$, and since $\sum_k T_{i\to k} = 1$, and $A_{i\to k} \leq 1$, the elements of the matrix satisfy $M_{ij} \geq 0$. The matrix M is therefore a stochastic matrix.

The Metropolis method is simply the power method for computing the right eigenvector of M with the largest magnitude eigenvalue. By construction, the correct probability distribution is a right eigenvector with eigenvalue 1. Therefore, for the Metropolis method to converge to this result, one has to show that M has only one eigenvalue with this magnitude, and all other eigenvalues are smaller.

12.6 Langevin and Fokker-Planck Equations

We end this chapter with a discussion and derivation of the Fokker-Planck and Langevin equations. These equations will in turn be used in our discussion on advanced Monte Carlo methods for quantum mechanical systems, see chapter for example chapter 16.

12.6.1 Fokker-Planck Equation

For many physical systems initial distributions of a stochastic variable y tend to an equilibrium distribution $w_{\mathrm{equilibrium}}(y)$, that is $w(y,t) \to w_{\mathrm{equilibrium}}(y)$ as $t \to \infty$. In equilibrium, detailed balance constrains the transition rates

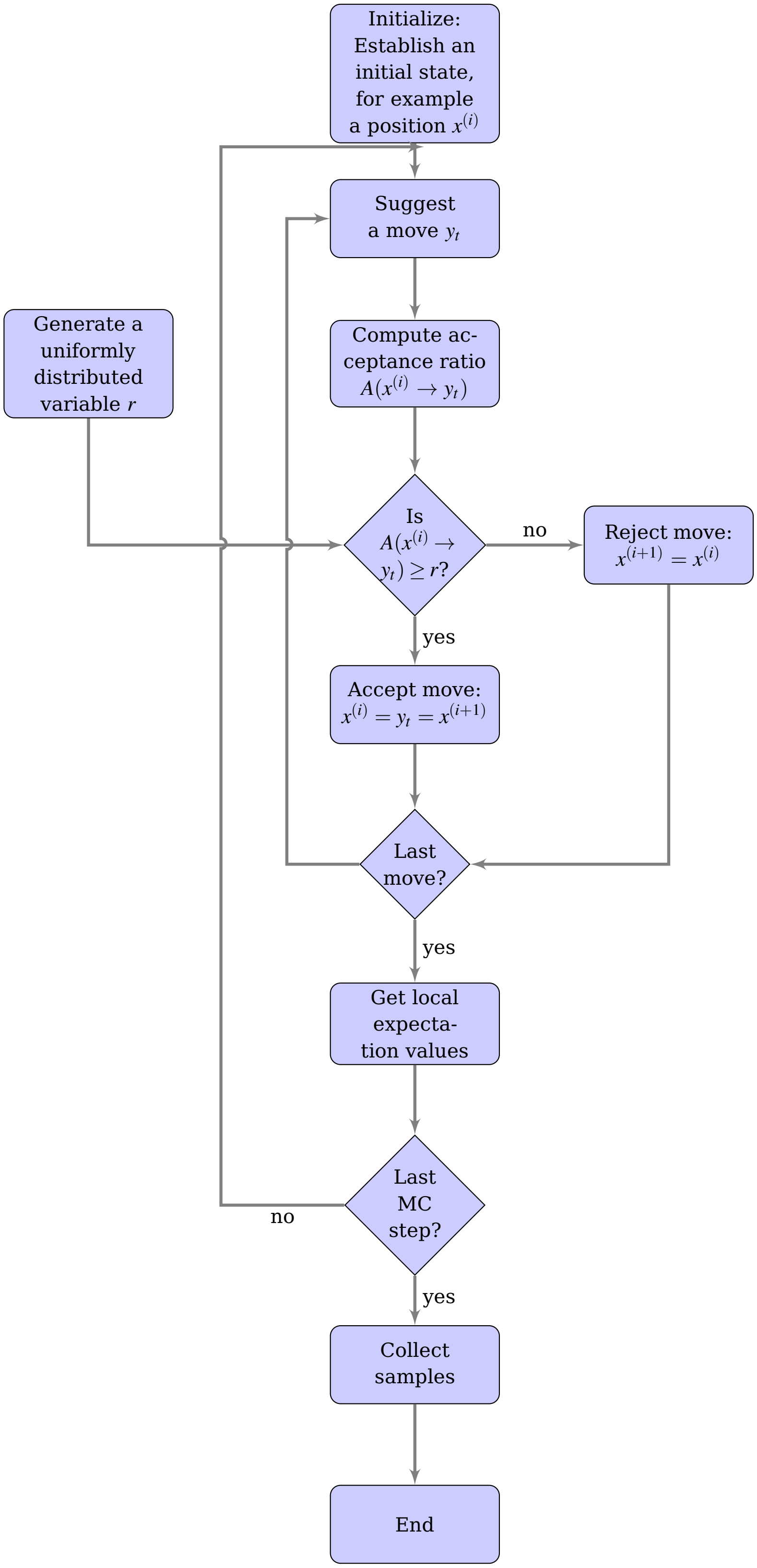
$$W(y \to y')w(y) = W(y' \to y)w_{\mathrm{equilibrium}}(y),$$

where $W(y' \to y)$ is the probability per unit time that the system changes from a state $|y\rangle$, characterized by the value y for the stochastic variable Y , to a state $|y'\rangle$.

Note that for a system in equilibrium the transition rate $W(y' \to y)$ and the reverse $W(y \to y')$ may be very different.

Let us now assume that we have three probability distribution functions for times $t_0 < t' < t$, that is $w(\mathbf{x}_0,t_0)$, $w(\mathbf{x}',t')$ and $w(\mathbf{x},t)$. We have then

$$w(\mathbf{x},t) = \int_{-\infty}^{\infty} W(\mathbf{x}.t|\mathbf{x}'.t')w(\mathbf{x}',t')d\mathbf{x}',$$



Initialize: Establish an initial state, for example a position $x^{(i)}$
Suggest a move y_t
Generate a uniformly distributed variable r
Compute acceptance ratio $A(x^{(i)} \to y_t)$
Is $A(x^{(i)} \to y_t) \geq r$?
no
Reject move: $x^{(i+1)} = x^{(i)}$
yes
Accept move: $x^{(i)} = y_t = x^{(i+1)}$
Last move?
yes
Get local expectation values
Last MC step?
no
yes
Collect samples
End

Fig. 12.7 Chart flow for the Metropolis algorithm.

and

$$w(\mathbf{x},t) = \int_{-\infty}^{\infty} W(\mathbf{x}.t|\mathbf{x}_0.t_0)w(\mathbf{x}_0,t_0)d\mathbf{x}_0,$$

and

$$w(\mathbf{x}',t') = \int_{-\infty}^{\infty} W(\mathbf{x}'.t'|\mathbf{x}_0,t_0)w(\mathbf{x}_0,t_0)d\mathbf{x}_0.$$

We can combine these equations and arrive at the famous Einstein-Smoluchenski-Kolmogorov-Chapman (ESKC) relation

$$W(\mathbf{x}t|\mathbf{x}_0t_0) = \int_{-\infty}^{\infty} W(\mathbf{x},t|\mathbf{x}',t')W(\mathbf{x}',t'|\mathbf{x}_0,t_0)d\mathbf{x}'.$$

We can replace the spatial dependence with a dependence upon say the velocity (or momentum), that is we have

$$W(\mathbf{v},t|\mathbf{v}_0,t_0) = \int_{-\infty}^{\infty} W(\mathbf{v},t|\mathbf{v}',t')W(\mathbf{v}',t'|\mathbf{v}_0,t_0)d\mathbf{x}'.$$

We will now derive the Fokker-Planck equation. We start from the ESKC equation

$$W(\mathbf{x},t|\mathbf{x}_0,t_0) = \int_{-\infty}^{\infty} W(\mathbf{x},t|\mathbf{x}',t')W(\mathbf{x}',t'|\mathbf{x}_0,t_0)d\mathbf{x}'.$$

We define $s = t' - t_0$, $\tau = t - t'$ and $t - t_0 = s + \tau$. We have then

$$W(\mathbf{x},s+\tau|\mathbf{x}_0) = \int_{-\infty}^{\infty} W(\mathbf{x},\tau|\mathbf{x}')W(\mathbf{x}',s|\mathbf{x}_0)d\mathbf{x}'.$$

Assume now that τ is very small so that we can make an expansion in terms of a small step xi, with $\mathbf{x}' = \mathbf{x} - \xi$, that is

$$W(\mathbf{x},s|\mathbf{x}_0) + \frac{\partial W}{\partial s}\tau + O(\tau^2) = \int_{-\infty}^{\infty} W(\mathbf{x},\tau|\mathbf{x}-\xi)W(\mathbf{x}-\xi,s|\mathbf{x}_0)d\mathbf{x}'.$$

We assume that $W(\mathbf{x},\tau|\mathbf{x}-\xi)$ takes non-negligible values only when ξ is small. This is just another way of stating the Master equation!

We say thus that $\mathbf{x}$ changes only by a small amount in the time interval τ. This means that we can make a Taylor expansion in terms of ξ, that is we expand

$$W(\mathbf{x},\tau|\mathbf{x}-\xi)W(\mathbf{x}-\xi,s|\mathbf{x}_0) = \sum_{n=0}^{\infty} \frac{(-\xi)^n}{n!} \frac{\partial^n}{\partial x^n}\left[W(\mathbf{x}+\xi,\tau|\mathbf{x})W(\mathbf{x},s|\mathbf{x}_0)\right].$$

We can then rewrite the ESKC equation as

$$\frac{\partial W}{\partial s}\tau = -W(\mathbf{x},s|\mathbf{x}_0) + \sum_{n=0}^{\infty} \frac{(-\xi)^n}{n!} \frac{\partial^n}{\partial x^n}\left[W(\mathbf{x},s|\mathbf{x}_0)\int_{-\infty}^{\infty} \xi^n W(\mathbf{x}+\xi,\tau|\mathbf{x})d\xi\right].$$

We have neglected higher powers of τ and have used that for $n = 0$ we get simply $W(\mathbf{x},s|\mathbf{x}_0)$ due to normalization.

We say thus that $\mathbf{x}$ changes only by a small amount in the time interval τ. This means that we can make a Taylor expansion in terms of ξ, that is we expand

$$W(\mathbf{x},\tau|\mathbf{x}-\xi)W(\mathbf{x}-\xi,s|\mathbf{x}_0) = \sum_{n=0}^{\infty} \frac{(-\xi)^n}{n!} \frac{\partial^n}{\partial x^n}\left[W(\mathbf{x}+\xi,\tau|\mathbf{x})W(\mathbf{x},s|\mathbf{x}_0)\right].$$

We simplify the above by introducing the moments

$$M_n = \frac{1}{\tau}\int_{-\infty}^{\infty}\xi^n W(\mathbf{x}+\xi,\tau|\mathbf{x})d\xi = \frac{\langle[\Delta x(\tau)]^n\rangle}{\tau},$$

resulting in

$$\frac{\partial W(\mathbf{x},s|\mathbf{x}_0)}{\partial s} = \sum_{n=1}^{\infty}\frac{(-\xi)^n}{n!}\frac{\partial^n}{\partial x^n}\left[W(\mathbf{x},s|\mathbf{x}_0)M_n\right].$$

When $\tau \to 0$ we assume that $\langle[\Delta x(\tau)]^n\rangle \to 0$ more rapidly than τ itself if $n > 2$. When τ is much larger than the standard correlation time of system then M_n for $n > 2$ can normally be neglected. This means that fluctuations become negligible at large time scales.

If we neglect such terms we can rewrite the ESKC equation as

$$\frac{\partial W(\mathbf{x},s|\mathbf{x}_0)}{\partial s} = -\frac{\partial M_1 W(\mathbf{x},s|\mathbf{x}_0)}{\partial x} + \frac{1}{2}\frac{\partial^2 M_2 W(\mathbf{x},s|\mathbf{x}_0)}{\partial x^2}.$$

In a more compact form we have

$$\frac{\partial W}{\partial s} = -\frac{\partial M_1 W}{\partial x} + \frac{1}{2}\frac{\partial^2 M_2 W}{\partial x^2},$$

which is the Fokker-Planck equation. It is trivial to replace position with velocity (momentum).

The solution to this equation is a Gaussian distribution and can be used to constrain proposed transitions moves, that one can model the transition probabilities T from our discussion of the Metropolis algorithm.

12.6.2 Langevin Equation

Consider a particle suspended in a liquid. On its path through the liquid it will continuously collide with the liquid molecules. Because on average the particle will collide more often on the front side than on the back side, it will experience a systematic force proportional with its velocity, and directed opposite to its velocity. Besides this systematic force the particle will experience a stochastic force $F(t)$. The equations of motion then read

$$\frac{dr}{dt} = v,$$

$$\frac{dv}{dt} = -\xi v + F,$$

The last equation is the Langevin equation. The original Langevin equation was meant to describe Brownian motion. It is a stochastic differential equation used to describe the time evolution of collective (normally macroscopic) variables that change only slowly with respect to the microscopic ones. The latter are responsible for the stochastic nature of the Langevin equation. We can say that we model our ignorance about the microscopic physics in a stochastic term. From the Langevin equation we can in turn derive for example the fluctuation dissipation theorem discussed below. To see, we need some information about the friction constant from hydrodynamics. From hydrodynamics we know that the friction constant ξ is given by

$$\xi = 6\pi\eta a/m$$

where η is the viscosity of the solvent and a is the radius of the particle.

Solving the Langevin equation we get

$$v(t) = v_0 e^{-\xi t} + \int_0^t d\tau e^{-\xi(t-\tau)}F(\tau).$$

If we want to get some useful information out of this, we have to average over all possible realizations of $F(t)$, with the initial velocity as a condition. A useful quantity is then

$$\langle v(t)\cdot v(t)\rangle_{v_0} = v_0^{-\xi 2t} + 2\int_0^t d\tau e^{-\xi(2t-\tau)} v_0\cdot\langle F(\tau)\rangle_{v_0}$$

$$+\int_0^t d\tau' \int_0^t d\tau e^{-\xi(2t-\tau-\tau')}\langle F(\tau)\cdot F(\tau')\rangle_{v_0}.$$

In order to continue we have to make some assumptions about the conditional averages of the stochastic forces. In view of the chaotic character of the stochastic forces the following assumptions seem to be appropriate. We assume that

$$\langle F(t)\rangle = 0,$$

and

$$\langle F(t)\cdot F(t')\rangle_{v_0} = C_{v_0}\delta(t-t').$$

We omit the subscript v_0 when the quantity of interest turns out to be independent of v_0. Using the last three equations we get

$$\langle v(t)\cdot v(t)\rangle_{v_0} = v_0^2 e^{-2\xi t} + \frac{C_{v_0}}{2\xi}(1-e^{-2\xi t}).$$

For large t this should be equal to the well-known result $3kT/m$, from which it follows that

$$\langle F(t)\cdot F(t')\rangle = 6\frac{kT}{m}\xi\delta(t-t').$$

This result is called the fluctuation-dissipation theorem.

Integrating

$$v(t) = v_0 e^{-\xi t} + \int_0^t d\tau e^{-\xi(t-\tau)}F(\tau),$$

we get

$$r(t) = r_0 + v_0\frac{1}{\xi}(1-e^{-\xi t}) + \int_0^t d\tau \int_0^\tau \tau' e^{-\xi(\tau-\tau')}F(\tau'),$$

from which we calculate the mean square displacement

$$\langle (r(t)-r_0)^2\rangle_{v_0} = \frac{v_0^2}{\xi}(1-e^{-\xi t})^2 + \frac{3kT}{m\xi^2}(2\xi t - 3 + 4e^{-\xi t} - e^{-2\xi t}).$$

For very large t this becomes

$$\langle (r(t)-r_0)^2\rangle = \frac{6kT}{m\xi}t$$

from which we get the Einstein relation

$$D = \frac{kT}{m\xi}$$

where we have used $\langle (r(t)-r_0)^2\rangle = 6Dt$.

The standard approach in for example quantum mechanical diffusion Monte Carlo calculations, is to use the Langevin equation to propose new moves (for examples new velocities or positions) since they will depend on the given probability distributions. These new proposed states or values are then used to compute the transition probability T, where the latter is the solution of for example the Fokker-Planck equation.

12.7 Exercises

12.1. 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.

12.2. Use the second program to fit the computed probability distribution with a normal distribution using your calculated values of σ^2 and $\langle x\rangle$.

12.3. In this exercise the aim is to show that the Metropolis algorithm generates the Boltzmann distribution

$$P(\beta) = \frac{e^{-\beta E}}{Z},$$

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 modelled 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

$$E = \frac{1}{2}kT = \frac{1}{2}v^2,$$

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 for example $v_{max} \sim 10\sqrt{T}$. This should include all relevant velocities which give a non-zero probability. But you need to check whether this is true or not.
  Then you construct a velocity interval defined by v_{max} and divide 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
      ...
    }
```

1. Make your own algorithm which sets up the histogram $P(v)dv$, find mean velocity, energy $\langle E\rangle$, energy variance $\mathrm{Var}(E)$ 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 closed form result $\langle E\rangle = kT/2$ for one dimension. Find also the closed-form expressions for the energy variance and the mean velocity and compare your calculations with these results. Use $T = 4$ and set the intial velocity to zero, i.e., $v_0 = 0$. Try different values of δv. Check the final result for the energy as a function of the number of Monte Carlo cycles.
2. Repeat the calculation in the previous exercise but using now a normal distribution. Does that improve your results compared with the exact expressions?
3. Make thereafter a plot of $\log(P(v))$ as function of E and see if you get a straight line. Comment the result.
4. In our analysis under [1) we have not discussed how the system reaches the most likely state, that is whether equilibrium has been reached or not. Make a plot of the mean velocity, energy, energy variance and the number of accepted steps for a given temperature as function of the number of Monte Carlo samples. Perform these calculations for several temperatures, namely $T = 0.5$, $T = 1$, $T = 2$ and $T = 10$ and comment your results. Can you find a rough measure for when the most likely state has been reached?
5. The analysis in point [4) is rather rough and obviously user dependent, in the sense that it is very much up to the user to define when an equilibrium situation has been reached or not. To improve upon this, compute the so-called time autocorrelation function defined here as

$$\phi(t) = \frac{1}{t_{\mathrm{max}} - t}\sum_{t'=0}^{t_{\mathrm{max}}-t} \bar{E}(t')\bar{E}(t'+t) - \frac{1}{t_{\mathrm{max}} - t}\sum_{t'=0}^{t_{\mathrm{max}}-t} \bar{E}(t') \times \frac{1}{t_{\mathrm{max}} - t}\sum_{t'=0}^{t_{\mathrm{max}}-t} \bar{E}(t'+t)$$

   for the mean energy $E\bar{(}t)$ and plot it as function of the number of Monte Carlo steps for the temperatures in [c). The time t corresponds to a given number of Monte Carlo cycles. Can you extract an equilibration measure? How does the correlation time behave as function of temperature? Comment your results. Be careful in choosing values of t, they should not be too close to t_{max}. Compute the autocorrelation function for all temperatures listed in [d) and compare your results with those in [d). Comment your results.
6. In the previous analysis we computed the time autocorrelation function. This quantity can be related to the covariance of our measurements. To achieve this you need to store the results of all contributions to the measurements of the mean energy and its variance σ_E^2 given by

$$\sigma_E^2 = \frac{1}{n^2}\sum_{k=1}^{n}(E_k - \bar{E})^2 + \frac{2}{n^2}\sum_{k<l}(E_k - \bar{E})(E_l - \bar{E})$$

   Here we assume that n corresponds to the number of Monte Carlo samples in one experiment and that we repeat these experiments a given time. We can assume here that we repeat these experiments $m = n$ times. The value $\bar{E}$ is the mean energy while $E_{k,l}$ represent individual measurements. The first term is the same as the error in the uncorrelated case. This means that the second term accounts for the error correction due to correlation between the measurements. For uncorrelated measurements this second term is zero.

Computationally the uncorrelated first term is much easier to treat efficiently than the second.

$$\mathrm{Var}(E) = \frac{1}{n}\sum_{k=1}^{n}(E_k - \langle E\rangle)^2 = \left(\frac{1}{n}\sum_{k=1}^{n} E_k^2\right) - \langle E\rangle^2$$

We just accumulate separately the values E_k^2 and E_k for every measurement E_k we receive. The correlation term, though, has to be calculated at the end of the experiment since we need all the measurements to calculate the cross terms. Therefore, all measurements have to be stored throughout the experiment.
Let us analyze the problem by splitting up the correlation term into partial sums of the form:

$$f_d = \frac{1}{n}\sum_{k=1}^{n-d}(E_k - \langle E\rangle)(E_{k+d} - \langle E\rangle)$$

The correlation term of the error can now be rewritten in terms of f_d:

$$\frac{2}{n}\sum_{k<l}(E_k - \langle E\rangle)(E_l - \langle E\rangle) = 2\sum_{d=1}^{n-1} f_d$$

The value of f_d reflects the correlation between measurements separated by the distance d in the samples. Notice that for $d = 0$, f is just the sample variance, $\mathrm{Var}(E)$. If we divide f_d by $\mathrm{Var}(E)$, we arrive at the so called *autocorrelation function*:

$$\kappa_d = \frac{f_d}{\mathrm{Var}(E)}$$

which gives us a useful measure of the correlation pair correlation starting always at 1 for $d = 0$.
The sample variance can now be written in terms of the autocorrelation function:

$$\begin{aligned}
\sigma_E^2 &= \frac{1}{n}\mathrm{Var}(E) + \frac{2}{n}\cdot\mathrm{Var}(E)\sum_{d=1}^{n-1}\frac{f_d}{\mathrm{Var}(E)} \\
&= \left(1 + 2\sum_{d=1}^{n-1}\kappa_d\right)\frac{1}{n}\mathrm{Var}(E) \\
&= \frac{\tau}{n}\cdot\mathrm{Var}(E) \qquad (12.18)
\end{aligned}$$

and we see that σ_E^2 can be expressed in terms the uncorrelated sample variance times a correction factor τ which accounts for the correlation between measurements. We call this correction factor the *autocorrelation time*:

$$\tau = 1 + 2\sum_{d=1}^{n-1}\kappa_d$$

For a correlation free experiment, τ equals 1. From the point of view of Eq. (12.18) we can interpret a sequential correlation as an effective reduction of the number of measurements by a factor τ. The effective number of measurements becomes

$$n_{\mathrm{eff}} = \frac{n}{\tau}$$

From the previous exercise you needed to store all experiments E_k in order to compute the time autocorrelation function. You can reuse these data in this exercise and compute the full variance σ_E^2, the covariance, the autocorrelation time τ and the effective number of measurements n_{eff}. It is sufficient to choose only one of the temperatures. Comment your results. Can you relate the correlation time τ to what you found [5)? What about the covariance and the time autocorrelation function?

12.4. The aim of this exercise is to simulate financial transactions among financial agents using Monte Carlo methods. The final goal is to extract a distribution of income as function of the income m. From Pareto's work (V. Pareto, 1897) it is known from empirical studies that the higher end of the distribution of money follows a distribution

$$w_m \propto m^{-1-\alpha},$$

with $\alpha \in [1,2]$. We will here follow the analysis made by Patriarca *et al* [72].

Here we will study numerically the relation between the microdynamical relations among financial agents and the resulting macroscopic money distribution.

We assume we have N agents that exchange money in pairs (i,j). We assume also that all agents start with the same amount of money $m_0 > 0$. At a given 'time step', we choose randomly a pair of agents (i,j) and let a transaction take place. This means that agent i's money m_i changes to m_i' and similarly we have $m_j \rightarrow m_j'$. Money is conserved during a transaction, meaning that

$$m_i + m_j = m_i' + m_j'. \tag{12.19}$$

The change is done via a random reassignement (a random number) ε, meaning that

$$m_i' = \varepsilon(m_i + m_j),$$

leading to

$$m_j' = (1-\varepsilon)(m_i + m_j).$$

The number ε is extracted from a uniform distribution. In this simple model, no agents are left with a debt, that is $m \geq 0$. Due to the conservation law above, one can show that the system relaxes toward an equilibrium state given by a Gibbs distribution

$$w_m = \beta \exp(-\beta m),$$

with

$$\beta = \frac{1}{\langle m \rangle},$$

and $\langle m \rangle = \sum_i m_i/N = m_0$, the average money. It means that after equilibrium has been reached that the majority of agents is left with a small number of money, while the number of richest agents, those with m larger than a specific value m', exponentially decreases with m'.

We assume that we have $N = 500$ agents. In each simulation, we need a sufficiently large number of transactions, say 10^7. Our aim is find the final equilibrium distribution w_m. In order to do that we would need several runs of the above simulations, at least $10^3 - 10^4$ runs (experiments).

a) Your task is to first set up an algorithm which simulates the above transactions with an initial amount m_0. The challenge here is to figure out a Monte Carlo simulation based on the above equations. You will in particular need to make an algorithm which sets up a histogram as function of m. This histogram contains the number of times a value m is registered and represents $w_m \Delta m$. You will need to set up a value for the interval Δm (typically $0.01 - 0.05$). That means you need to account for the number of times you register an income in the interval $m, m + \Delta m$. The number of times you register this income, represents the value that enters the histogram. You will also need to find a criterion for when the equilibrium situation has been reached.

b) Make thereafter a plot of $\log(w_m)$ as function of m and see if you get a straight line. Comment the result.

c) We can then change our model to allow for a saving criterion, meaning that the agents save a fraction λ of the money they have before the transaction is made. The final distribution will then no longer be given by Gibbs distribution. It could also include a taxation on financial transactions.

The conservation law of Eq. (12.19) holds, but the money to be shared in a transaction between agent i and agent j is now $(1-\lambda)(m_i+m_j)$. This means that we have

$$m_i' = \lambda m_i + \varepsilon(1-\lambda)(m_i+m_j),$$

and

$$m_j' = \lambda m_j + (1-\varepsilon)(1-\lambda)(m_i+m_j),$$

which can be written as

$$m_i' = m_i + \delta m$$

and

$$m_j' = m_j - \delta m,$$

with

$$\delta m = (1-\lambda)(\varepsilon m_j - (1-\varepsilon)m_i),$$

showing how money is conserved during a transaction. Select values of $\lambda = 0.25, 0.5$ and $\lambda = 0.9$ and try to extract the corresponding equilibrium distributions and compare these with the Gibbs distribution. Comment your results. If you have time, see if you can extract a parametrization of the above curves (see Patriarca *et al* [72])

Chapter 13
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*

Abstract 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 two of the most studied models, the Ising model and the Potts model for the interaction among classical spins. These models have been widely used for studies of phase transitions.

13.1 Introduction and Motivation

Fluctuations play a central role in our understanding of phase transitions. Their behavior near critical points convey important information about the underlying many-particle interactions. In this chapter we will focus on two widely studied models in statistical physics, the Ising model and the Potts model for interacting spins. The main focus is on the Ising model. Both models can exhibit first and second order phase transitions and are perhaps among the most studied systems in statistical physics with respect to simulations of phase transitions. The Norwegian-born chemist Lars Onsager developed in 1944 an ingenious mathematical description of the Ising model [73] 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 do-

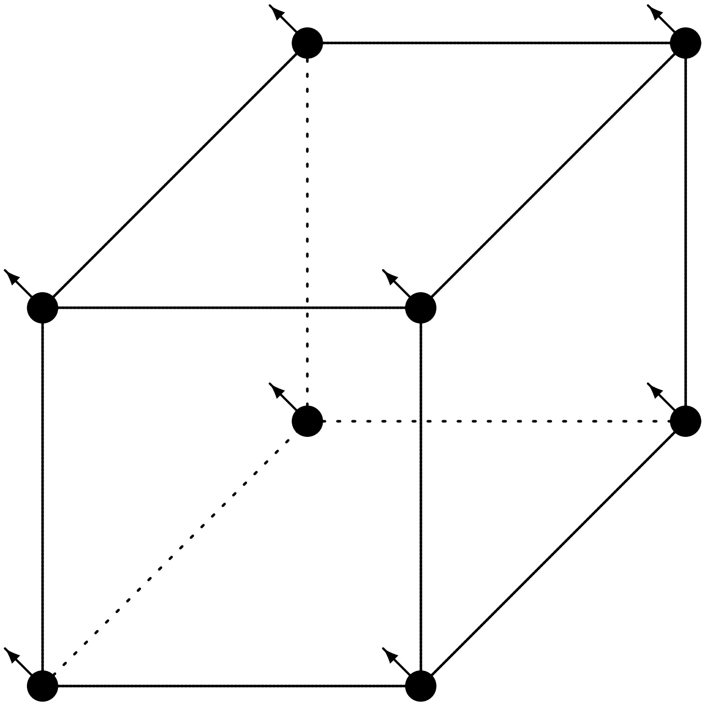


Fig. 13.1 Example of a cubic lattice with atoms at each corner. Each atom has a finite magnetic moment which points in a particular direction.
.

mains 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 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, respectively. 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. 13.1. 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 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 [74].

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. 13.2.

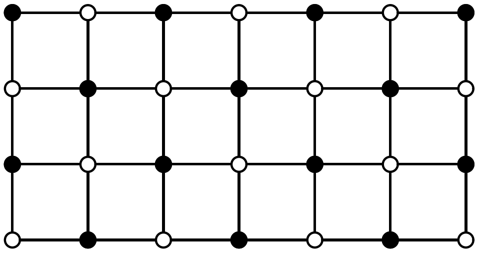


Fig. 13.2 The open (white) circles at each lattice point can represent a vacant site, while the black circles can represent the absorption of an atom on a metal surface.
.

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.

13.2 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 13.1 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 13.1 Overview of the most common ensembles and their variables. Here we have define $\mathscr{M}$ - to be the magnetization, $\mathscr{D}$ - the electric dipole moment, $\mathscr{H}$ - the magnetic field and $\mathscr{E}$ - 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	$V,\mathscr{M},\mathscr{D}$ E N	$V,\mathscr{M},\mathscr{D}$ T N	$V,\mathscr{M},\mathscr{D}$ T μ	$P,\mathscr{H},\mathscr{E}$ T N
Potential	Entropy	Helmholtz	PV	Gibbs
Energy	Internal	Internal	Internal	Enthalpy

13.2.1 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

$$S = k_B ln\Omega.$$

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

$$\frac{1}{k_B T} = \left(\frac{\partial \log \Omega}{\partial E}\right)_{N,V},$$

the pressure

$$\frac{p}{k_BT} = \left(\frac{\partial \log \Omega}{\partial V}\right)_{N,E},$$

or the chemical potential.

$$\frac{\mu}{k_BT} = -\left(\frac{\partial \log \Omega}{\partial N}\right)_{V,E}.$$

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.

13.2.2 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

$$P_i(\beta) = \frac{e^{-\beta E_i}}{Z}$$

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

$$Z = \sum_{i=1}^{M} e^{-\beta E_i},$$

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

$$F = -k_BTlnZ = \langle E\rangle - TS.$$

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. 13.1 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

$$S = k_BlnZ + k_BT\left(\frac{\partial lnZ}{\partial T}\right)_{N,V},$$

and the pressure by

$$p = k_BT\left(\frac{\partial lnZ}{\partial V}\right)_{N,T}.$$

Similarly we can compute the chemical potential as

$$\mu = -k_B T \left(\frac{\partial lnZ}{\partial N}\right)_{V,T}.$$

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

$$\langle E \rangle = k_B T^2 \left(\frac{\partial lnZ}{\partial T}\right)_{V,N}$$

or using the probability distribution P_i as

$$\langle E \rangle = \sum_{i=1}^{M} E_i P_i(\beta) = \frac{1}{Z}\sum_{i=1}^{M} E_i e^{-\beta E_i}.$$

The energy is proportional to the first derivative of the potential, Helmholtz' free energy. The corresponding variance is defined as

$$\sigma_E^2 = \langle E^2 \rangle - \langle E \rangle^2 = \frac{1}{Z}\sum_{i=1}^{M} E_i^2 e^{-\beta E_i} - \left(\frac{1}{Z}\sum_{i=1}^{M} E_i e^{-\beta E_i}\right)^2.$$

If we divide the latter quantity with kT^2 we obtain the specific heat at constant volume

$$C_V = \frac{1}{k_B T^2}\left(\langle E^2 \rangle - \langle E \rangle^2\right),$$

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

$$\langle \mathscr{M} \rangle = \sum_{i}^{M} \mathscr{M}_i P_i(\beta) = \frac{1}{Z}\sum_{i}^{M} \mathscr{M}_i e^{-\beta E_i},$$

and the corresponding variance

$$\sigma_{\mathscr{M}}^2 = \langle \mathscr{M}^2 \rangle - \langle \mathscr{M} \rangle^2 = \frac{1}{Z}\sum_{i=1}^{M} \mathscr{M}_i^2 e^{-\beta E_i} - \left(\frac{1}{Z}\sum_{i=1}^{M} \mathscr{M}_i e^{-\beta E_i}\right)^2.$$

This quantity defines also the susceptibility χ

$$\chi = \frac{1}{k_B T}\left(\langle \mathscr{M}^2 \rangle - \langle \mathscr{M} \rangle^2\right).$$

13.2.3 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,

$$pV = k_B T ln\Xi,$$

and the entropy is given by

$$S = k_B ln\Xi + k_B T \left(\frac{\partial ln\Xi}{\partial T}\right)_{V,\mu},$$

while the mean number of particles is

$$\langle N \rangle = k_B T \left(\frac{\partial ln\Xi}{\partial \mu}\right)_{V,T}.$$

The pressure is determined as

$$p = k_B T \left(\frac{\partial ln\Xi}{\partial V}\right)_{\mu,T}.$$

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

$$G = -k_B T ln\Delta,$$

and the entropy is given by

$$S = k_B ln\Delta + k_B T \left(\frac{\partial ln\Delta}{\partial T}\right)_{p,N}.$$

We can compute the volume as

$$V = -k_B T \left(\frac{\partial ln\Delta}{\partial p}\right)_{N,T},$$

and finally the chemical potential

$$\mu = -k_B T \left(\frac{\partial ln\Delta}{\partial N}\right)_{p,T}.$$

In this chapter we work with the canonical ensemble only.

13.3 Ising Model and Phase Transitions in Magnetic Systems

13.3.1 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

$$E = -J \sum_{<kl>}^{N} s_k s_l - \mathscr{B} \sum_{k}^{N} s_k,$$

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 $\mathscr{B}$ 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 $\mathscr{B}=0$ only.

In order to calculate expectation values such as the mean energy $\langle E\rangle$ or magnetization $\langle \mathscr{M}\rangle$ in statistical physics at a given temperature, we need a probability distribution

$$P_i(\beta)=\frac{e^{-\beta E_i}}{Z}$$

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

$$Z=\sum_{i=1}^{M}e^{-\beta E_i},$$

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

$$E_i=-J\sum_{<kl>}^{N}s_ks_l.$$

To better understand what is meant with a configuration, consider first the case of the one-dimensional Ising model with $\mathscr{B}=0$. In general, a given configuration of N spins in one dimension may look like

$$\begin{array}{cccccccccc}\uparrow\uparrow\uparrow & \dots & \uparrow & \downarrow & \uparrow & \dots & \uparrow & \downarrow\\ 1\;2\;3 & \dots & i-1 & i & i+1 & \dots & N-1 & N\end{array}$$

In order to illustrate these features let us further specialize to just two spins.

With two spins, since each spin takes two values only, 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. This means that there is no contribution from points to the right or left of the endpoints. For the one-dimensional case, the energy is then written as a sum over a single index

   $$E_i=-J\sum_{j=1}^{N-1}s_js_{j+1},$$

   If we label the first spin as s_1 and the second as s_2 we obtain the following expression for the energy

   $$E=-Js_1s_2.$$

   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 the neighbour to the right of s_N is assumed to take the value of s_1. Similarly, the neighbour to the left of s_1 takes the value s_N. In this case the energy for the one-dimensional lattice reads

$$E_i = -J \sum_{j=1}^{N} s_j s_{j+1},$$

and we obtain the following expression for the two-spin case

$$E = -J(s_1 s_2 + s_2 s_1).$$

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 ;
}
```

The magnetization is however the same, defined as

$$\mathscr{M}_i = \sum_{j=1}^{N} s_j,$$

where we sum over all spins for a given configuration i.

Table 13.2 lists the energy and magnetization for both free ends and periodic boundary conditions.

We can reorganize Table 13.2 according to the number of spins pointing up, as shown in Table 13.3. 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

Table 13.2 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 13.3 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

we have $N \to \infty$, and 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

$$E = -8J \qquad \begin{array}{c}\uparrow\uparrow\\ \uparrow\uparrow\end{array} \qquad E = 0 \qquad \begin{array}{c}\uparrow\uparrow\\ \uparrow\downarrow\end{array} \qquad E = 0 \qquad \begin{array}{c}\downarrow\downarrow\\ \uparrow\downarrow\end{array} \qquad E = -8J \qquad \begin{array}{c}\downarrow\downarrow\\ \downarrow\downarrow\end{array}$$

In the Table 13.4 we group these configurations according to their total energy and magnetization.

Table 13.4 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

$$Z_N = \sum_{s_1=\pm 1} \dots \sum_{s_N=\pm 1} \exp\left(\beta J \sum_{j=1}^{N-1} s_j s_{j+1}\right),$$

and since the last spin occurs only once in the last sum in the exponential, we can single out the last spin as follows

$$\sum_{s_N=\pm 1} \exp\left(\beta J s_{N-1} s_N\right) = 2cosh(\beta J).$$

The partition function consists then of a part from the last spin and one from the remaining spins resulting in

$$Z_N = Z_{N-1} 2cosh(\beta J).$$

We can repeat this process and obtain

$$Z_N = (2cosh(\beta J))^{N-2} Z_2,$$

with Z_2 given by

$$Z_2 = \sum_{s_1=\pm 1} \sum_{s_2=\pm 1} \exp\left(\beta J s_1 s_2\right) = 4cosh(\beta J),$$

resulting in

$$Z_N = 2(2cosh(\beta J))^{N-1}.$$

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

$$Z_N = \sum_{s_1=\pm 1} \dots \sum_{s_N=\pm 1} \exp\left(\beta J \sum_{j=1}^{N} s_j s_{j+1}\right),$$

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

$$Z_N = \sum_{\{s_i=\pm 1\}} \prod_{i=1}^{N} \exp\left(\beta J s_i s_{i+1}\right),$$

where the first sum is meant to represent all lattice sites. Introducing the matrix $\hat{\mathbf{T}}$ (the so-called transfer matrix)

$$\hat{\mathbf{T}} = \begin{pmatrix} e^{\beta J} & e^{-\beta J} \\ e^{-\beta J} & e^{\beta J} \end{pmatrix},$$

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

$$Z_N = \sum_{\{s_i=\pm 1\}} \hat{\mathbf{T}}_{s_1 s_2} \hat{\mathbf{T}}_{s_2 s_3} \dots \hat{\mathbf{T}}_{s_N s_1} = Tr \hat{\mathbf{T}}^N.$$

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

$$Z_N = \lambda_1^N + \lambda_2^N = 2^N \left([cosh(\beta J)]^N + [sinh(\beta J)]^N \right).$$

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 closed-form 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

$$\langle E \rangle = -\frac{\partial lnZ}{\partial \beta} = -(N-1)Jtanh(\beta J).$$

Helmholtz's free energy is given by

$$F = -k_BTlnZ_N = -Nk_BTln\left(2cosh(\beta J)\right).$$

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

$$Z_2 = \sum_{i=1}^{2} e^{-\beta E_i} = 2e^{-\beta J} + 2e^{\beta J} = 4cosh(\beta J)$$

If we take the limit $T \to 0$ ($\beta \to \infty$) and set $N = 2$, we obtain

$$\lim_{\beta \to \infty} \langle E \rangle = -J\frac{e^{J\beta} - e^{-J\beta}}{e^{J\beta} + e^{-J\beta}} = -J,$$

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

$$C_V = \frac{1}{kT^2}\frac{\partial^2}{\partial \beta^2} lnZ_N = (N-1)k\left(\frac{\beta J}{cosh(\beta J)}\right)^2.$$

Note well that this expression for the specific heat from the one-dimensional Ising model does not diverge or exhibit discontinuities, as can be seen from Fig. 13.3.

In one dimension we do not have a second order phase transition, although this is predicted by mean field models [57].

We can repeat this exercise for the case with periodic boundary conditions as well. Helmholtz's free energy is in this case

$$F = -k_BTln(\lambda_1^N + \lambda_2^N) = -k_BT\left\{Nln(\lambda_1) + ln\left(1 + (\frac{\lambda_2}{\lambda_1})^N\right)\right\},$$

which in the limit $N \to \infty$ results in $F = -k_BTNln(\lambda_1)$ 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.

Hitherto we have limited ourselves to studies of systems with zero external magnetic field, viz $\mathscr{B} = 0$. We will mostly study systems which exhibit a spontaneous magnitization. It is however instructive to extend the one-dimensional Ising model to $\mathscr{B} \neq 0$, yielding a partition

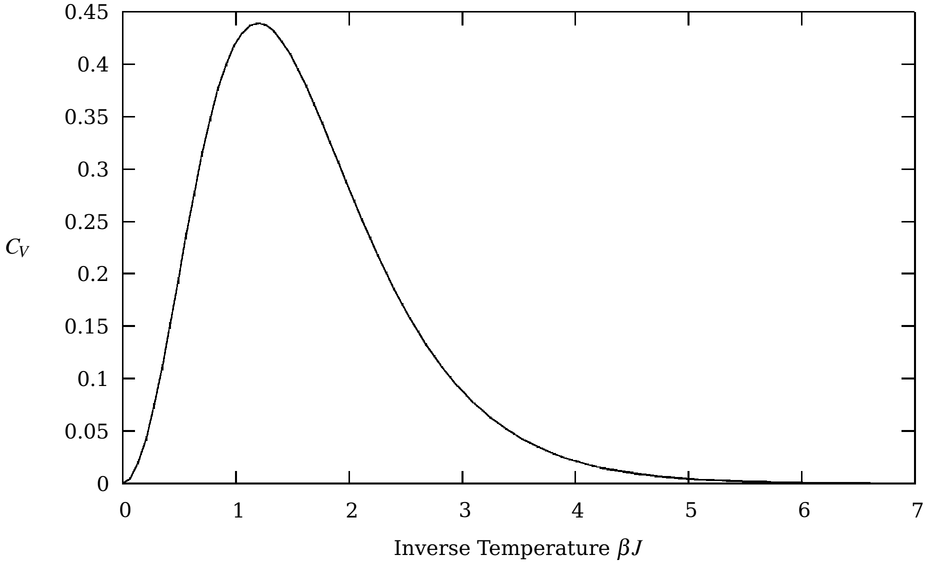


Fig. 13.3 Heat capacity per spin $(C_V/(N-1)k_B$ as function of inverse temperature β for the one-dimensional Ising model.

function (with periodic boundary conditions)

$$Z_N = \sum_{s_1=\pm 1} \cdots \sum_{s_N=\pm 1} \exp(\beta \sum_{j=1}^{N} (Js_js_{j+1} + \frac{\mathscr{B}}{2}(s_j + s_{j+1})),$$

which yields a new transfer matrix with matrix elements $t_{11} = e^{\beta(J+\mathscr{B})}$, $t_{1-1} = e^{-\beta J}$, $t_{-11} = e^{\beta J}$ and $t_{-1-1} = e^{\beta(J-\mathscr{B})}$ with eigenvalues

$$\lambda_1 = e^{\beta J}cosh(\beta J) + (e^{2\beta J}sinh^2(\beta\mathscr{B}) + e^{-2\beta J})^{1/2},$$

and

$$\lambda_1 = e^{\beta J}cosh(\beta J) - (e^{2\beta J}sinh^2(\beta\mathscr{B}) + e^{-2\beta J})^{1/2}.$$

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

$$F = -Nk_BTln\left(e^{\beta J}cosh(\beta J) + (e^{2\beta J}sinh^2(\beta\mathscr{B}) + e^{-2\beta J})^{1/2}\right).$$

It is now useful to compute the expectation value of the magnetisation per spin

$$\langle \mathscr{M}/N \rangle = \frac{1}{NZ}\sum_i^M \mathscr{M}_i e^{-\beta E_i} = -\frac{1}{N}\frac{\partial F}{\partial \mathscr{B}},$$

resulting in

$$\langle \mathscr{M}/N \rangle = \frac{sinh(\beta\mathscr{B})}{\left(sinh^2(\beta\mathscr{B}) + e^{-2\beta J})^{1/2}\right)}.$$

We see that for $\mathscr{B} = 0$ 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

$$Z = 2e^{-8J\beta} + 2e^{8J\beta} + 12,$$

and resulting mean energy

$$\langle E \rangle = -\frac{J}{Z}\left(16e^{8J\beta} - 16e^{-8J\beta}\right),$$

it is a highly non-trivial task to find the closed-form expression for Z_N in the thermodynamic limit. The closed-form expression for the Ising model in two dimensions was obtained via a mathematical tour de force in 1944 by the Norwegian chemist Lars Onsager [73]. The exact partition function for N spins in two dimensions and with zero magnetic field $\mathscr{B}$ is given by

$$Z_N = \left[2cosh(\beta J)e^{I}\right]^N,$$

with

$$I = \frac{1}{2\pi}\int_0^{\pi} d\phi ln\left[\frac{1}{2}\left(1 + (1 - \kappa^2 sin^2\phi)^{1/2}\right)\right],$$

and

$$\kappa = 2sinh(2\beta J)/cosh^2(2\beta J).$$

The resulting energy is given by

$$\langle E \rangle = -Jcoth(2\beta J)\left[1 + \frac{2}{\pi}(2tanh^2(2\beta J) - 1)K_1(q)\right],$$

with $q = 2sinh(2\beta J)/cosh^2(2\beta J)$ and the complete elliptic integral of the first kind

$$K_1(q) = \int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - q^2 sin^2\phi}}.$$

Differentiating once more with respect to temperature we obtain the specific heat given by

$$C_V = \frac{4k_B}{\pi}(\beta Jcoth(2\beta J))^2\left\{K_1(q) - K_2(q) - (1 - tanh^2(2\beta J))\left[\frac{\pi}{2} + (2tanh^2(2\beta J) - 1)K_1(q)\right]\right\}, \tag{13.1}$$

where

$$K_2(q) = \int_0^{\pi/2} d\phi\sqrt{1 - q^2 sin^2\phi}, \tag{13.2}$$

is the complete elliptic integral of the second kind. Near the critical temperature T_C the specific heat behaves as

$$C_V \approx -\frac{2}{\pi}\left(\frac{2J}{k_B T_C}\right)^2 ln\left|1 - \frac{T}{T_C}\right| + \text{const.} \tag{13.3}$$

In theories of critical phenomena one ca show that for temperatures T below a critical temperature T_C, the heat capacity scales as [75]

$$C_V \sim \left|1 - \frac{T}{T_C}\right|^{-\alpha},$$

and Onsager's result is a special case of this power law behavior. The limiting form of the function

$$lim_{\alpha\to 0}\frac{1}{\alpha}(Y^{-\alpha} - 1) = -lnY,$$

can be used to infer that the closed-form result is a special case of the power law singularity with $\alpha = 0$.

Similar relations applies to other expectation values. An example is the the spontaneous magnetisation per spin. This quantity is also highly non-trivial to compute. Here we simply limit ourselves to list Onsager's result

$$\langle \mathscr{M}(T)/N \rangle = \left[1 - \frac{(1-tanh^2(\beta J))^4}{16tanh^4(\beta J)}\right]^{1/8},$$

for $T < T_C$. For $T > T_C$ the magnetization is zero. From theories of critical phenomena one can show that the magnetization behaves as $T \to T_C$ from below

$$\langle \mathscr{M}(T)/N \rangle \sim (T_C - T)^{1/8}.$$

The susceptibility behaves as

$$\chi(T) \sim |T_C - T|^{-7/4}.$$

Before we proceed, we need to say some words about phase transitions and critical phenomena.

13.4 Phase Transitions and Critical Phenomena

A phase transition is marked by abrupt macroscopic changes as external parameters are changed, such as an increase of temperature. The point where a phase transition takes place is called a critical point.

We distinguish normally between two types of phase transitions; first-order transitions and second-order transitions. An important quantity in studies of phase transitions is the so-called correlation length ξ and various correlations functions like spin-spin correlations. For the Ising model we shall show below that the correlation length is related to the spin-correlation function, which again defines the magnetic susceptibility. The spin-correlation function is nothing but the covariance and expresses the degree of correlation between spins.

The correlation length defines the length scale at which the overall properties of a material start to differ from its bulk properties. It is the distance over which the fluctuations of the microscopic degrees of freedom (for example the position of atoms) are significantly correlated with each other. Usually it is of the order of few interatomic spacings for a solid. The correlation length ξ depends however on external conditions such as pressure and temperature.

First order/discontinuous phase transitions are characterized by two or more states on either side of the critical point that can coexist at the critical point. As we pass through the critical point we observe a discontinuous behavior of thermodynamical functions. The correlation length is normally finite at the critical point. Phenomena such as hysteris occur, viz. there is a continuation of state below the critical point into one above the critical point. This continuation is metastable so that the system may take a macroscopically long time to readjust. A classical example is the melting of ice. It takes a specific amount of time before all the ice has melted. The temperature remains constant and water and ice can coexist for a macroscopic time. The energy shows a discontinuity at the critical point, reflecting the fact that a certain amount of heat is needed in order to melt all the ice

Second order or continuous transitions are different and in general much difficult to understand and model. The correlation length diverges at the critical point, fluctuations are correlated over all distance scales, which forces the system to be in a unique critical phase. The two phases on either side of the critical point become identical. The disappearance of a spontaneous magnetization is a classical example of a second-order phase transitions. Structural transitions in solids are other types of second-order phase transitions. Strong correlations make a perturbative treatment impossible. From a theoretical point of view, the way out is renormalization group theory [75–81]. Table 13.5 lists some typical system with their pertinent order parameters.

Table 13.5 Examples of various phase transitions with respective order parameters.

System	Transition	Order Parameter
Liquid-gas	Condensation/evaporation	Density difference $\Delta\rho = \rho_{liquid} - \rho_{gas}$
Binary liquid	mixture/Unmixing	Composition difference
Quantum liquid	Normal fluid/superfluid	$<\phi>$, ψ = wavefunction
Liquid-solid	Melting/crystallisation	Reciprocal lattice vector
Magnetic solid	Ferromagnetic	Spontaneous magnetisation M
	Antiferromagnetic	Sublattice magnetisation M
Dielectric solid	Ferroelectric	Polarization P
	Antiferroelectric	Sublattice polarisation P

Using Ehrenfest's definition of the order of a phase transition we can relate the behavior around the critical point to various derivatives of the thermodynamical potential. In the canonical ensemble we are using, the thermodynamical potential is Helmholtz' free energy

$$F = \langle E \rangle - TS = -kT lnZ$$

meaning $lnZ = -F/kT = -F\beta$. The energy is given as the first derivative of F

$$\langle E \rangle = -\frac{\partial lnZ}{\partial \beta} = \frac{\partial(\beta F)}{\partial \beta}.$$

and the specific heat is defined via the second derivative of F

$$C_V = -\frac{1}{kT^2}\frac{\partial^2(\beta F)}{\partial \beta^2}.$$

We can relate observables to various derivatives of the partition function and the free energy. When a given derivative of the free energy or the partition function is discontinuous or diverges (logarithmic divergence for the heat capacity from the Ising model) we talk of a phase transition of order of the derivative. A first-order phase transition is recognized in a discontinuity of the energy, or the first derivative of F. The Ising model exhibits a second-order phase transition since the heat capacity diverges. The susceptibility is given by the second derivative of F with respect to external magnetic field. Both these quantities diverge.

13.4.1 The Ising Model and Phase Transitions

The Ising model in two dimensions with $\mathscr{B} = 0$ 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 $\langle \mathscr{M} \rangle \neq 0$. Above T_C the average magnetization is zero. The mean magnetization approaches zero at T_C with an infinite slope. Such a behavior is an example of what are called critical phenomena [78,80,82]. A critical phenomenon is normally marked by one or more thermodynamical variables which vanish above a critical point. In our case this is the mean magnetization $\langle \mathscr{M} \rangle \neq 0$. Such a parameter is normally called the order parameter.

Critical phenomena have been extensively studied in physics. One major reason is that we still do not have a satisfactory understanding of the properties of a system close to a critical point. Even for the simplest three-dimensional systems we cannot predict exactly the values of various thermodynamical variables. Simplified theoretical approaches like mean-field models discussed below, can even predict the wrong physics. Mean-field theory results in a second-order phase transition for the one-dimensional Ising model, whereas we saw in the previous section that 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

$$\begin{array}{ccccccccc} \uparrow\uparrow\uparrow\ldots & \uparrow & \uparrow & \uparrow & \ldots & \uparrow & \uparrow \\ 1\ 2\ 3\ \ldots & i-1 & i & i+1 & \ldots & N-1 & N \end{array}$$

which has a total energy $-NJ$ and magnetization N, where we used periodic boundary conditions. If we flip half of the spins we obtain a possible configuration where the first half of the spins point upwards and the last half points downwards we arrive at the configuration

$$\begin{array}{ccccccc} \uparrow\uparrow\uparrow\ldots & \uparrow & \uparrow & \downarrow & \ldots & \downarrow & \downarrow \\ 1\ 2\ 3\ \ldots & N/2-1 & N/2 & N/2+1 & \ldots & N-1 & N \end{array}$$

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 $\langle \mathscr{M} \rangle \neq 0$ to a paramagnetic phase with $\langle \mathscr{M} \rangle = 0$ 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 (below we will illustrate this in a qualitative way using mean-field theory).

We showed in the previous section that the mean magnetization is given by (for temperature below T_C)

$$\langle \mathscr{M}(T) \rangle \sim (T - T_C)^{\beta},$$

where $\beta = 1/8$ is a so-called critical exponent. A similar relation applies to the heat capacity

$$C_V(T) \sim |T_C - T|^{-\alpha},$$

and the susceptibility

$$\chi(T) \sim |T_C - T|^{-\gamma},$$

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 is close to 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 the correlation ξ near T_C is

$$\xi(T) \sim |T_C - T|^{-\nu}. \tag{13.4}$$

A second-order phase transition is characterized by a correlation length which spans the whole system. The correlation length is typically of the order of some few interatomic distances. The fact that a system like the Ising model, whose energy is described by the interaction between neighboring spins only, can yield correlation lengths of macroscopic size at a critical point is still a feature which is not properly understood. Stated differently, how can the spins propagate their correlations so extensively when we approach the critical point, in particular since the interaction acts only between nearest spins? Below we will compute the correlation length via the spin-sin correlation function for the one-dimensional Ising model.

In our actual calculations of the two-dimensional Ising model, we are however always limited to a finite lattice and ξ will be proportional with the size of the lattice at the critical point. Through finite size scaling relations [75, 78–80] it is possible to relate the behavior at finite lattices with the results for an infinitely large lattice. The critical temperature scales then as

$$T_C(L) - T_C(L = \infty) \propto aL^{-1/\nu}, \tag{13.5}$$

with a a constant and ν defined in Eq. (13.4). The correlation length for a finite lattice size can then be shown to be proportional to

$$\xi(T) \propto L \sim |T_C - T|^{-\nu}.$$

and if we set $T = T_C$ one can obtain the following relations for the magnetization, energy and susceptibility for $T \leq T_C$

$$\langle \mathscr{M}(T) \rangle \sim (T - T_C)^{\beta} \propto L^{-\beta/\nu},$$

$$C_V(T) \sim |T_C - T|^{-\gamma} \propto L^{\alpha/\nu},$$

and

$$\chi(T) \sim |T_C - T|^{-\alpha} \propto L^{\gamma/\nu}.$$

13.4.2 Critical Exponents and Phase Transitions from Mean-field Models

In order to understand the above critical exponents, we will derive some of the above relations using what is called mean-field theory.

In studies of phase transitions we are interested in minimizing the free energy by varying the average magnetisation, which is the order parameter for the Ising model. The magnetization disappears at T_C.

Here we use mean field theory to model the free energy F. In mean field theory the local magnetisation is a treated as a constant and all effects from fluctuations are neglected. Stated differently, we reduce a complicated system of many interacting spins to a set of equations for each spin. Each spin sees now a mean field which is set up by the surrounding spins. We neglect therefore the role of spin-spin correlations. A way to achieve this is to rewrite the interaction between two spins at sites i and j, respectively, by adding and subtracting the mean value of the spin $\langle S\rangle$, that is

$$S_iS_j = (S_i - \langle S\rangle + \langle S\rangle)(S_i - \langle S\rangle + \langle S\rangle) \approx \langle S\rangle^2 + \langle S\rangle(S_i - \langle S\rangle) + \langle S\rangle(S_j - \langle S\rangle),$$

where we have ignored terms of the order $(S_i - \langle S\rangle)(S_i - \langle S\rangle)$. These are the terms which lead to correlations between neighbouring spins. In mean field theory we ignore correlations.

This means that we can rewrite the Hamiltonian

$$E = -J\sum_{<ij>}^{N} S_kS_l - B\sum_{i}^{N} S_i,$$

as

$$E = -J\sum_{<ij>} \langle S\rangle^2 + \langle S\rangle(S_i - \langle S\rangle) + \langle S\rangle(S_j - \langle S\rangle) - B\sum_i S_i,$$

resulting in

$$E = -(B + zJ\langle S\rangle)\sum_i S_i + zJ\langle S\rangle^2,$$

with z the number of nearest neighbours for a given site i. We have included the external magnetic field B for completeness.

We can then define an effective field which all spins see, namely

$$B_{\mathrm{eff}} = (B + zJ\langle S\rangle).$$

To obtain the vaue of $\langle S\rangle)$ we employ again our results from the canonical ensemble. The partition function reads in this case

$$Z = e^{-NzJ\langle S\rangle^2/kT}\left(2cosh(B_{\mathrm{eff}}/kT)\right)^N,$$

with a free energy

$$F = -kTlnZ = -NkTln(2) + NzJ\langle S\rangle^2 - NkTln\left(cosh(B_{\mathrm{eff}}/kT)\right)$$

and minimizing F with respect to $\langle S\rangle$ we arrive at

$$\langle S\rangle = tanh(2cosh\left(B_{\mathrm{eff}}/kT\right)).$$

Close to the phase transition we expect $\langle S\rangle$ to become small and eventually vanish. We can then expand F in powers of $\langle S\rangle$ as

$$F = -NkTln(2) + NzJ\langle s\rangle^2 - NkT - BN\langle s\rangle + NkT\left(\frac{1}{2}\langle s\rangle^2 + \frac{1}{12}\langle s\rangle^4 + \dots\right),$$

and using $\langle M\rangle = N\langle S\rangle$ we can rewrite Helmholtz free energy as

$$F = F_0 - B\langle M\rangle + \frac{1}{2}a\langle M\rangle^2 + \frac{1}{4}b\langle M\rangle^4 + \dots$$

Let $\langle M\rangle = m$ and

$$F = F_0 + \frac{1}{2}am^2 + \frac{1}{4}bm^4 + \frac{1}{6}cm^6$$

F has a minimum at equilibrium $F'(m) = 0$ and $F''(m) > 0$

$$F'(m) = 0 = m(a + bm^2 + cm^4),$$

and if we assume that m is real we have two solutions

$$m = 0,$$

or

$$m^2 = \frac{b}{2c}\left(-1 \pm \sqrt{1 - 4ac/b^2}\right).$$

This relation can be used to describe both first and second-order phase transitions. Here we consider the second case. We assume that $b > 0$ and let $a \ll 1$ since we want to study a perturbation around $m = 0$. We reach the critical point when $a = 0$, that is

$$m^2 = \frac{b}{2c}\left(-1 \pm \sqrt{1 - 4ac/b^2}\right) \approx -a/b.$$

We define the temperature dependent function

$$a(T) = \alpha(T - T_C),$$

with $\alpha > 0$ and T_C being the critical temperature where the magnetization vanishes. If a is negative we have two solutions

$$m = \pm\sqrt{-a/b} = \pm\sqrt{\frac{\alpha(T_C - T)}{b}},$$

meaning that m evolves continuously to the critical temperature where $F = 0$ for $T \leq T_C$

We can now compute the entropy as follows

$$S = -\left(\frac{\partial F}{\partial T}\right).$$

For $T \geq T_C$ we have $m = 0$ and

$$S = -\left(\frac{\partial F_0}{\partial T}\right),$$

and for $T \leq T_C$

$$S = -\left(\frac{\partial F_0}{\partial T}\right) - \alpha^2(T_C - T)/2b,$$

and we see that there is a smooth crossover at T_C.

In theories of critical phenomena one has that

$$C_V \sim \left|1 - \frac{T}{T_C}\right|^{-\alpha},$$

and Onsager's result is a special case of this power law behavior. The limiting form of the function

$$lim_{\alpha\rightarrow 0}\frac{1}{\alpha}(Y^{-\alpha} - 1) = -\log(Y),$$

meaning that the closed-form result is a special case of the power law singularity with $\alpha = 0$.

13.5 The Metropolis Algorithm and the Two-dimensional Ising Model

We switch now back to the Ising model in two dimensions and explore how to write a program that will allow us to compute various thermodynamical quantities. The algorithm of choice for solving the Ising model is the approach proposed by Metropolis *et al.* [83] in 1953. As discussed in chapter 12, 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 configuration 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://folk.uio.no/mhjensen/compphys/programs/chapter13/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 external magnetic field is

$$\Delta E = E_2 - E_1 = J\sum_{<kl>}^{N} s_k^1 s_l^1 - J\sum_{<kl>}^{N} s_k^2 s_l^2,$$

which we can rewrite as

$$\Delta E = -J\sum_{<kl>}^{N} s_k^2(s_l^2 - s_l^1),$$

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

$$\Delta E = 2Js_l^1\sum_{<k>}^{N} s_k, \tag{13.6}$$

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

$$M_2 = M_1 + 2s_l^2, \tag{13.7}$$

where M_1 and M_2 are the magnetizations before and after the spin flip, respectively. Eqs. (13.6) and (13.7) 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. 13.4.

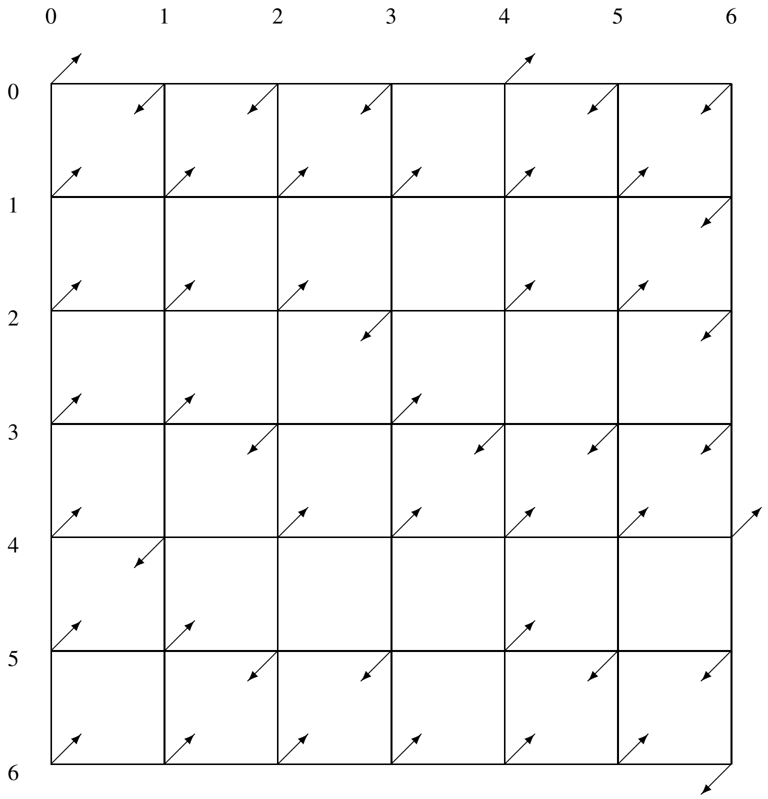


Fig. 13.4 Example of a two-dimensional 7×7 lattice with spins pointing either up or down. The variable spin_matrix[1][0] takes the value +1 while spin_matrix[0][6] is -1.

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 practically temperatures $T < 1.5$. Else, it keeps the value from the preceeding temperature. The latter is done in order to get a best possible estimate of the most likely state for the given temperature.

We have built up a code where we run over a larger temperature span, typically with values $T \in [1.0, 3.0]$.

```
// 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. This is

```
void output(int n_spins, int mcs, double temperature, double *average)
{
 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)/n_spins/n_spins;
 double Mvariance = (M2average - Maverage*Maverage)/n_spins/n_spins;
 double M2variance = (M2average - Mabsaverage*Mabsaverage)/n_spins/n_spins;
 double Mvariance = (M2average - Mabsaverage*Mabsaverage)/n_spins/n_spins;
 ofile << setiosflags(ios::showpoint | ios::uppercase);
 ofile << setw(15) << setprecision(8) << temperature;
 ofile << setw(15) << setprecision(8) << Eaverage/n_spins/n_spins;
 ofile << setw(15) << setprecision(8) << Evariance/temperature/temperature;
 // ofile << setw(15) << setprecision(8) << Maverage/n_spins/n_spins;
 ofile << setw(15) << setprecision(8) << M2variance/temperature;
 ofile << setw(15) << setprecision(8) << Mabsaverage/n_spins/n_spins << endl;
} // end output function
```

13.5.1 Parallelization of the Ising Model

To parallelize the Ising model, or many Monte Carlo procedures is in general rather simple. Here we show an example of a modified main program where we let different nodes perform a given set of Monte Carlo samples. We have fixed the size of the grid to a 40×40 lattice, but the reading of these variables can easily be done by the master node, either by reading the variables from the command line or via a user-defined file.

Note that every node has its own seed for the random number generators.

```
/*
  Program to solve the two-dimensional Ising model
  with zero external field using MPI
  The coupling constant J = 1
  Boltzmann's constant = 1, temperature has thus dimension energy
  Metropolis sampling is used. Periodic boundary conditions.
  The code needs an output file on the command line.
*/
#include "mpi.h"
```

```
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"

using namespace std;

// output file
ofstream ofile;

// inline function for periodic boundary conditions
inline int periodic(int i, int limit, int add) {
  return (i+limit+add) % (limit);
}
// Function to initialise energy and magnetization
void initialize(int, 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 begins here

int main(int argc, char* argv[])
{
  char *outfilename;
  long idum;
  int **spin_matrix, n_spins, mcs, my_rank, numprocs;
  double w[17], average[5], total_average[5],
       initial_temp, final_temp, E, M, temp_step;

  // MPI initializations
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  if (my_rank == 0 && argc <= 1) {
    cout << "Bad Usage: " << argv[0] <<
      " read output file" << endl;
    exit(1);
  }
  if (my_rank == 0 && argc > 1) {
    outfilename=argv[1];
    ofile.open(outfilename);
  }
  n_spins = 40; mcs = 1000000; initial_temp = 2.4; final_temp = 2.7; temp_step =0.1;
  /*
  Determine number of intervall which are used by all processes
  myloop_begin gives the starting point on process my_rank
  myloop_end gives the end point for summation on process my_rank
  */
  int no_intervalls = mcs/numprocs;
  int myloop_begin = my_rank*no_intervalls + 1;
  int myloop_end = (my_rank+1)*no_intervalls;
  if ( (my_rank == numprocs-1) &&( myloop_end < mcs) ) myloop_end = mcs;

  // broadcast to all nodes common variables
  MPI_Bcast (&n_spins, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast (&initial_temp, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast (&final_temp, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast (&temp_step, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
 // Allocate memory for spin matrix
 spin_matrix = (int**) matrix(n_spins, n_spins, sizeof(int));
 // every node has its own seed for the random numbers, this is important else
 // if one starts with the same seed, one ends with the same random numbers
 idum = -1-my_rank; // random starting point
 // Start Monte Carlo sampling by looping over T first
 for ( double temperature = initial_temp; temperature <= final_temp;
    temperature+=temp_step){
  //   initialise energy and magnetization
  E = M = 0.;
  // initialise array for expectation values
  initialize(n_spins, spin_matrix, E, M);
  // 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/temperature);
  for( int i = 0; i < 5; i++) average[i] = 0.;
  for( int i = 0; i < 5; i++) total_average[i] = 0.;
  // start Monte Carlo computation
  for (int cycles = myloop_begin; cycles <= myloop_end; cycles++){
   Metropolis(n_spins, idum, spin_matrix, E, M, w);
   // update expectation values for local node
   average[0] += E; average[1] += E*E;
   average[2] += M; average[3] += M*M; average[4] += fabs(M);
  }
  // Find total average
  for( int i =0; i < 5; i++){
   MPI_Reduce(&average[i], &total_average[i], 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  }
  // print results
  if ( my_rank == 0) {
   output(n_spins, mcs, temperature, total_average);
  }
 }
 free_matrix((void **) spin_matrix); // free memory
 ofile.close(); // close output file
 // End MPI
 MPI_Finalize ();
 return 0;
}
```

13.6 Selected Results for the Ising Model

In Figs. 13.5-13.8 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. 13.5 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 $T_C \approx 2.269$.

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. 13.6 shows the absolute value of the magnetisation as function of the number of spins.

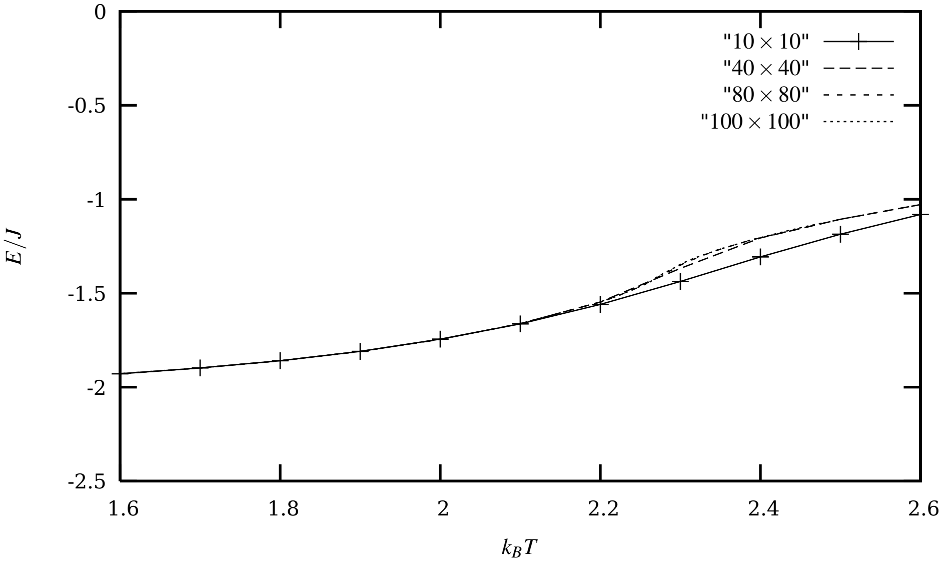


Fig. 13.5 Average energy per spin as function of the lattice size for the two-dimensional Ising model.

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 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.

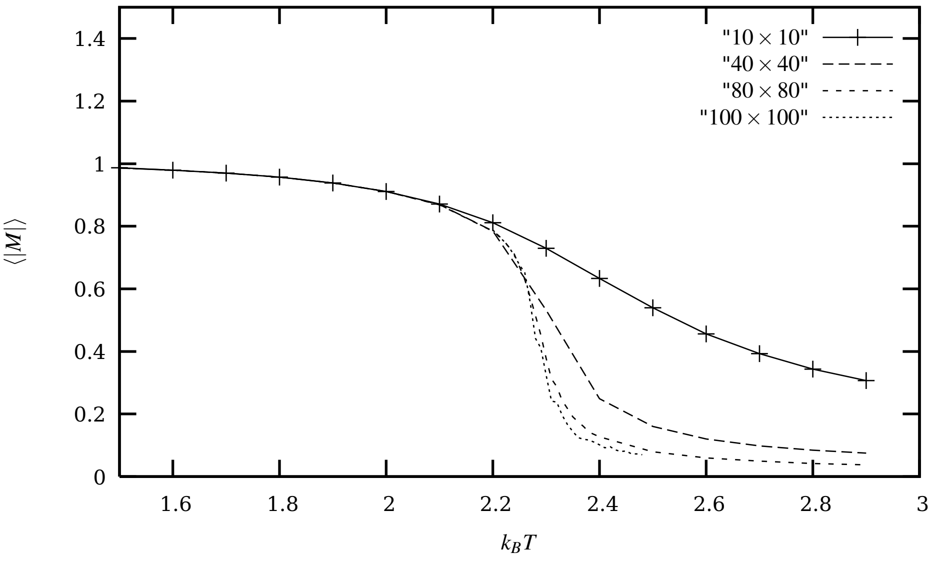


Fig. 13.6 Absolute value of the average magnetization per spin as function of the lattice size for the two-dimensional Ising model.

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 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 closed-form results

$$\left[1-\frac{(1-tanh^2(\beta J))^4}{16tanh^4(\beta J)}\right]^{1/8},$$

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. 13.7 we show the corresponding specific heat.

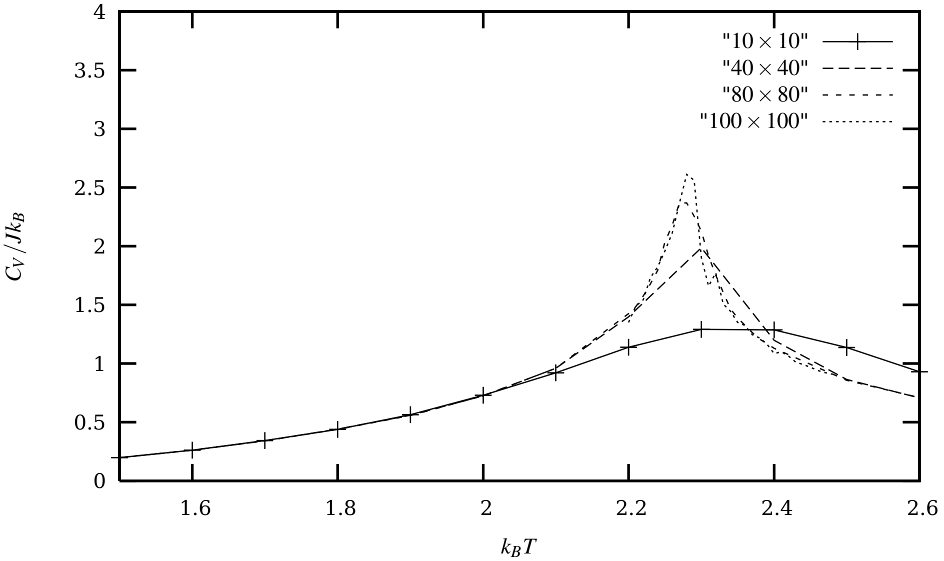


Fig. 13.7 Heat capacity per spin as function of the lattice size for the two-dimensional Ising model.

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 behavior is seen for the susceptibility as well, with an even sharper peak, as can be seen from Fig. 13.8.

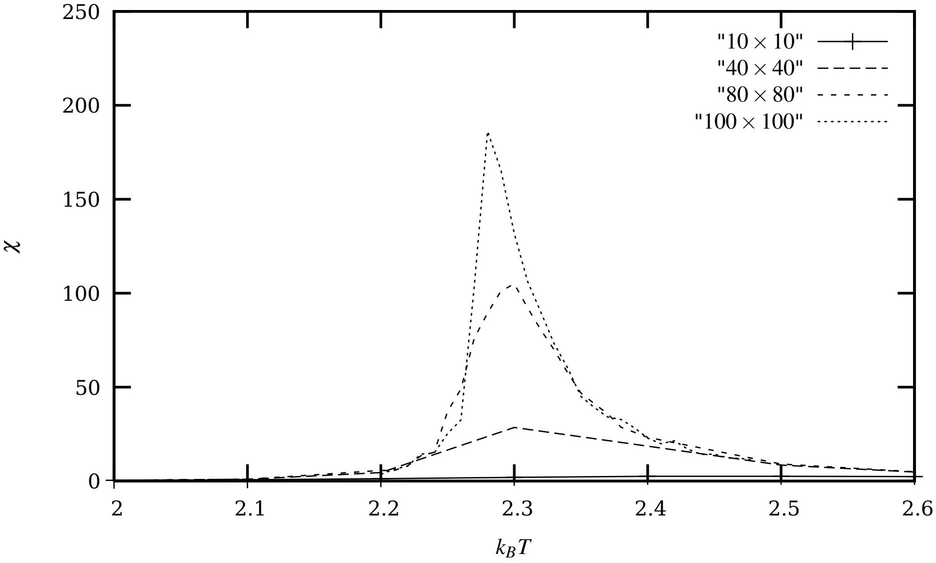


Fig. 13.8 Susceptibility per spin as function of the lattice size for the two-dimensional Ising model. Note that we have computed the susceptibility as $\xi = (\langle M^2\rangle - \langle|M|\rangle^2)/k_bT$.

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 [84, 85] are much more efficient in simulating properties near the critical temperature. For spin models like the class of higher-order Potts models discussed in section 13.8, the efficiency of the Metropolis algorithm is simply inadequate. These topics are discussed in depth in the textbooks of Newman and Barkema [79] and Landau and Binder [80].

13.7 Correlation Functions and Further Analysis of the Ising Model

13.7.1 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 11. 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. 13.9 and 13.10. In Fig. 13.9 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.

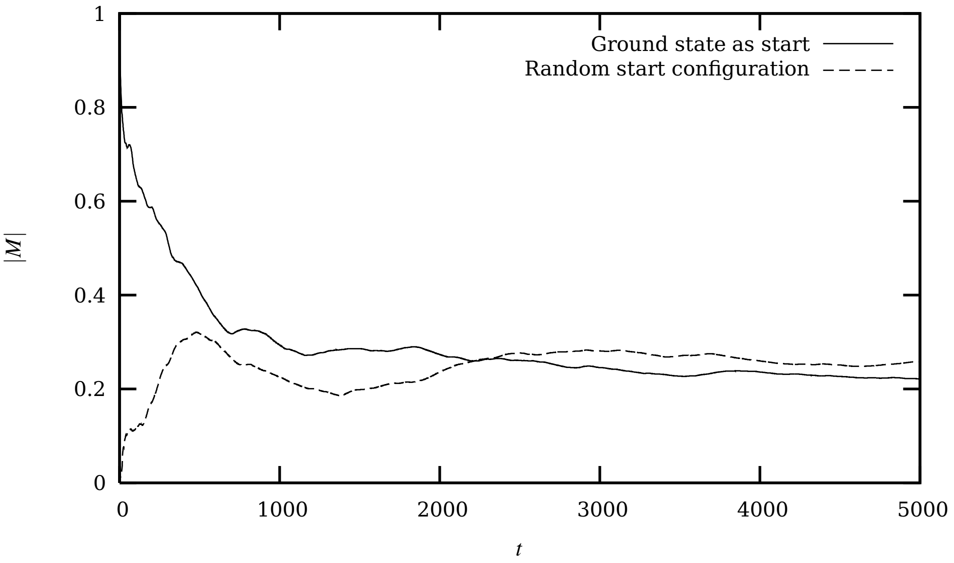


Fig. 13.9 Absolute value of the mean magnetisation as function of time t. Time is represented by the number of Monte Carlo cycles. The calculations have been performed with a 40×40 lattice for a temperature $k_BT/J = 2.4$. Two start configurations were used, one with a random orientation of the spins and one with an ordered orientation, which corresponds to the ground state of the system.

Fig. 13.10 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

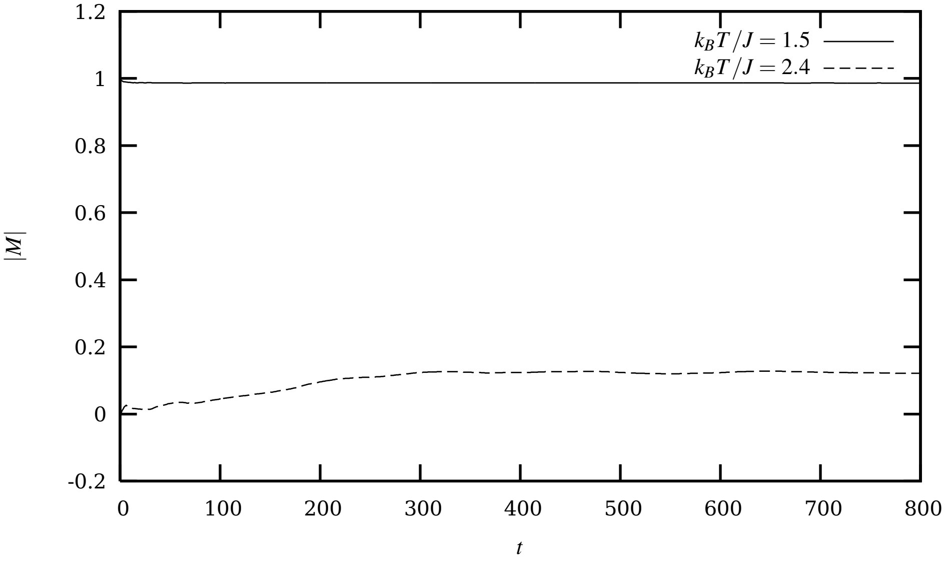


Fig. 13.10 Absolute value of the mean magnetisation as function of time t. Time is represented by the number of Monte Carlo cycles. The calculations were performed with a 100×100 lattice for temperatures $k_BT/J = 1.5$ and $k_BT/J = 2.4$. For the lowest temperature, an ordered start configuration was chosen, while for the temperature close to T_C, a disordered configuration was used.

the temperature close to $k_BT_C/J \approx 2.269$ 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. It is normal to set the equilibration time $\tau = t_{\mathrm{eq}}$. 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 [79], 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 $z = 2.1665 \pm 0.0012$ [86]. 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.
figure=figures/pict4.ps,width=height=6cm figure=figures/pict2.ps,width=height=6cm

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. figure=figures/pict1.ps,width=height=6cm figure=figures/pict6.ps,width=height=6cm

Since all spins point in the same direction, the chance of performing the flip

$$
\begin{array}{ccccc}
 & \uparrow & & & \uparrow \\
E=-4J & \uparrow\uparrow\uparrow & \Longrightarrow & E=4J & \uparrow\downarrow\uparrow \\
 & \uparrow & & & \uparrow
\end{array}
$$

leads to an energy difference of $\Delta E = 8J$. Using the exact critical temperature $k_BT_C/J \approx 2.269$, 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 [87] and the Swendsen-Wang [88] 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.

13.7.2 Time-correlation Function

The so-called time-displacement autocorrelation $\phi(t)$ for the magnetization is given by[1]

$$\phi(t) = \int dt' \left[\mathscr{M}(t') - \langle\mathscr{M}\rangle\right]\left[\mathscr{M}(t'+t) - \langle\mathscr{M}\rangle\right],$$

which can be rewritten as

$$\phi(t) = \int dt' \left[\mathscr{M}(t')\mathscr{M}(t'+t) - \langle\mathscr{M}\rangle^2\right],$$

where $\langle\mathscr{M}\rangle$ is the average value of the magnetization and $\mathscr{M}(t)$ its instantaneous value. We can discretize this function as follows, where we used our set of computed values $\mathscr{M}(t)$ for a set of discretized times (our Monte Carlo cycles corresponding to a sweep over the lattice)

$$\phi(t) = \frac{1}{t_{\max}-t}\sum_{t'=0}^{t_{\max}-t}\mathscr{M}(t')\mathscr{M}(t'+t) - \frac{1}{t_{\max}-t}\sum_{t'=0}^{t_{\max}-t}\mathscr{M}(t') \times \frac{1}{t_{\max}-t}\sum_{t'=0}^{t_{\max}-t}\mathscr{M}(t'+t). \tag{13.8}$$

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 $\mathscr{M}(t)$ can become large. Note also that we could replace the magnetization with the mean energy, or any other expectation values of interest.

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 fluc-

[1] We follow closely chapter 3 of Ref. [79].

tuating 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. (13.8), 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. 13.11 exhibits the time-correlation function for the magnetization for the same lattice and temperatures discussed in Fig. 13.10.

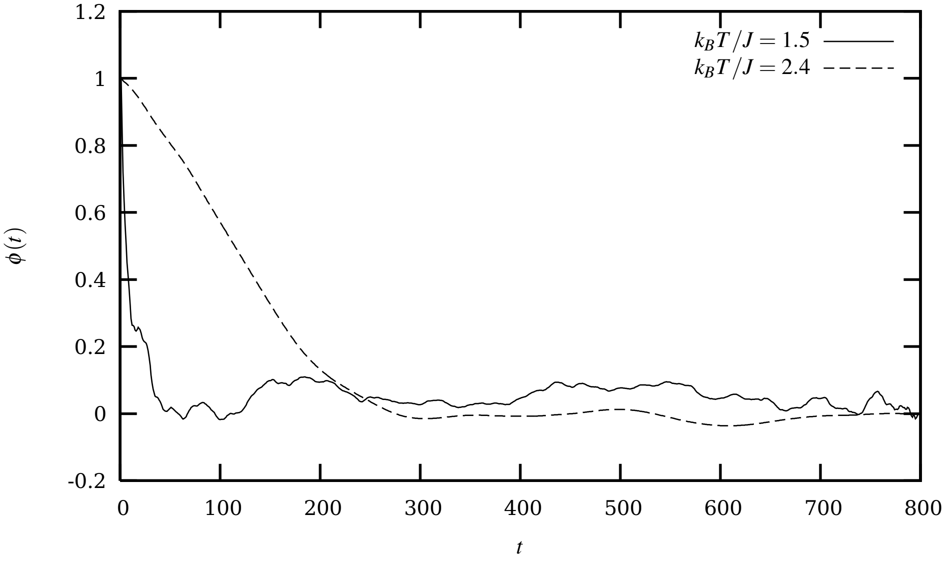


Fig. 13.11 Time-autocorrelation function with time t as number of Monte Carlo cycles. It has been normalized with $\phi(0)$. The calculations have been performed for a 100×100 lattice at $k_BT/J = 2.4$ with a disordered state as starting point and at $k_BT/J = 1.5$ with an ordered state as starting point.

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. 13.10. Similarly, for $k_BT/J = 1.5$ the correlation function reaches zero quickly, in good agreement again with the results of Fig. 13.10. 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 12 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

$$\hat{\mathbf{w}}(0) = \sum_i \alpha_i\hat{\mathbf{v}}_i,$$

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 $i^{\rm th}$ 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 $t\rightarrow\infty$, $\hat{\mathbf{w}}(t)$ 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

$$\langle \mathscr{M}(t)\rangle = \sum_{\mu}\hat{\mathbf{w}}(t)_{\mu}\mathscr{M}_{\mu},$$

or as the scalar of a vector product

$$\langle \mathscr{M}(t)\rangle = \hat{\mathbf{w}}(t)\mathbf{m},$$

with $\mathbf{m}$ being the vector whose elements are the values of $\mathscr{M}_{\mu}$ in its various microstates μ. We rewrite this relation as

$$\langle \mathscr{M}(t)\rangle = \hat{\mathbf{w}}(t)\mathbf{m} = \sum_{i}\lambda_i^t\alpha_i\hat{\mathbf{v}}_i\mathbf{m}_i.$$

If we define $m_i=\hat{\mathbf{v}}_i\mathbf{m}_i$ as the expectation value of $\mathscr{M}$ in the $i^{\rm th}$ eigenstate we can rewrite the last equation as

$$\langle \mathscr{M}(t)\rangle = \sum_{i}\lambda_i^t\alpha_i m_i.$$

Since we have that in the limit $t\rightarrow\infty$ the mean magnetization is dominated by the the largest eigenvalue λ_0, we can rewrite the last equation as

$$\langle \mathscr{M}(t)\rangle = \langle \mathscr{M}(\infty)\rangle + \sum_{i\neq 0}\lambda_i^t\alpha_i m_i.$$

We define the quantity

$$\tau_i = -\frac{1}{log\lambda_i},$$

and rewrite the last expectation value as

$$\langle \mathscr{M}(t)\rangle = \langle \mathscr{M}(\infty)\rangle + \sum_{i\neq 0}\alpha_i m_i e^{-t/\tau_i}. \tag{13.9}$$

The quantities τ_i are the correlation times for the system. They control also the autocorrelation 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 $\langle \mathscr{M}(t)\rangle$. Recalling that the mean value is equal to $\langle \mathscr{M}(\infty)\rangle$ we arrive at the expectation values

$$\phi(t) = \langle \mathscr{M}(0)-\mathscr{M}(\infty)\rangle\langle \mathscr{M}(t)-\mathscr{M}(\infty)\rangle,$$

and using Eq. (13.9) we arrive at

$$\phi(t) = \sum_{i,j\neq 0} m_i\alpha_i m_j\alpha_j e^{-t/\tau_i},$$

which is appropriate for all times.

13.8 The 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 $1, 2, \dots, q$. The Kronecker delta function δ_{s_l,s_k} equals unity if the spins are equal and is zero otherwise. The variable 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.

For references on the Potts Models see Refs. [89–92]

Compared with the two-dimensional Ising model, the Potts model can take only four possible values for ΔE, as shown in the following part of code

```
void Energy(double T,double *Boltzmann){
 Boltzmann[0] = exp(-J/T) ;
 Boltzmann[1] = exp(-2*J/T);
 Boltzmann[2] = exp(-3*J/T);
 Boltzmann[3] = exp(-4*J/T);
}//Energy
```

However, when we run the Potts model we must choose the new value of q randomly. The following functions encodes the Metropolis algorithm for the Potts model.

```
void Metropolis(int q,double *Boltzmann,int **Spin,long& seed,double& E){

 int SpinFlip, LocalEnergy0, LocalEnergy, x, y, dE;

  for(int i = 0; i < N; i++){
   for(int j = 0; j < N; j++){
    x = (int) (ran1(&seed)*N);
    y = (int) (ran1(&seed)*N);
    LocalEnergy0 = 0;
    LocalEnergy = 0;
    dE = 0;
    if(Spin[x][y] == Spin[x][periodic(y,N,-1)])
     LocalEnergy0 --;
    if(Spin[x][y] == Spin[periodic(x,N,-1)][y])
     LocalEnergy0 --;
    if(Spin[x][y] == Spin[x][periodic(y,N,1)])
     LocalEnergy0 --;
    if(Spin[x][y] == Spin[periodic(x,N,1)][y])
     LocalEnergy0 --;
    do{
    SpinFlip = (int)(ran1(&seed)*(q)+1);
    }while(SpinFlip == Spin[x][y]);

    if(SpinFlip == Spin[x][periodic(y,N,-1)])
```

```
        LocalEnergy --;
      if(SpinFlip == Spin[periodic(x,N,-1)][y])
        LocalEnergy --;
      if(SpinFlip == Spin[x][periodic(y,N,1)])
        LocalEnergy --;
      if(SpinFlip == Spin[periodic(x,N,1)][y])
        LocalEnergy --;

      dE = LocalEnergy - LocalEnergy0;

      if(dE<=0){
        Spin[x][y] = SpinFlip;
        E += J*dE;
      }
      else if(ran1(&seed)<Boltzmann[dE-1]){
        Spin[x][y] = SpinFlip;
        E += J*dE;
```

In the calculation of the energy difference from one spin configuration to the other, we have for the $q = 2$ Potts two possible values only. When we change one of the values such as flipping a spin we start with an energy $E = -4J$. Now we flip this spin as shown below. The energy of the new configuration is $E = 0J$, yielding $\Delta E = 4J$.

$$E = -4J \qquad \begin{array}{c} \uparrow \\ \uparrow\uparrow\uparrow \\ \uparrow \end{array} \qquad \Longrightarrow \qquad E = 4J \qquad \begin{array}{c} \uparrow \\ \uparrow\downarrow\uparrow \\ \uparrow \end{array}$$

However, when q becomes large the standard Metropolis algorithm becomes inefficient. Assume that $q = 100$. At high T the acceptance probability is close to 1 and our algorithm is efficient.

When we cool down the system $T \to T_C$, more and more 'spins' will take the same value and we build up cluster/domains with equally valued 'spins'. If the spins are aligned with its neigbours it has lower energy and thereby larger weight $e^{-\beta E}$.

The problem comes when q is large. If our value is one of the other 96 values, we need on average $100/4 = 25$ steps to find a desired state. This can result in a very long time to find state with lower energy.

If we start at low temperatures, there is an extra cost to excite, leading to smaller acceptance probability. We can easily end up in situation where we have almost 96 out 100 moves rejected. This means that we need a better algorithm. Such improvements are discussed in the chapter on advanced statistical physics problems (not available in this version).

13.9 Exercises

13.1. Convince yourself that the values listed in Table 13.4 are correct.

13.2. 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 $N \to \infty$.

13.3. 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 $\mathscr{B} = 0$.

1. 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.

2. 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.
3. 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.
4. 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?

13.4. Here we will simulate the two-dimensional Ising model.

1. Assume that the number of spins in the x and y directions are two, viz $L = 2$. Find the closed-form expression for the partition function and the corresponding mean values for E, $\mathscr{M}$, the capacity C_V and the suceptibility χ as function of T using periodic boundary conditions.
2. Write your own code for the two-dimensional Ising model with periodic boundary conditions and zero external field $\mathscr{B}$. Set $L = 2$ and compare your numerical results with the closed-form 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.
3. 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 $\langle \mathscr{M} \rangle$, 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 $\langle E \rangle$, $\langle \mathscr{M} \rangle$, C_V and χ as functions of T. Can you see an indication of a phase transition?
4. Use Eq. (13.5) 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 $L = 80$.
5. 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 $\mathscr{M}$ at the exact value $T = T_C$ for $L = 10$, $L = 20$, $L = 40$ and $L = 80$. Plot $log_{10}\, \mathscr{M}$ and χ som funksjon av $log_{10}\, L$ and use the scaling relations 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 $\gamma = 7/4$.
6. 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. (13.1) and (13.3). 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.

13.5. 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 $1,2,\dots,q$. 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.

1. 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.
2. 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 above-mentioned 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.
3. 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 14
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.*

Abstract 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 and the helium atom. Systems with many interacting fermions and bosons such as liquid ^{4}He and Bose Einstein condensation of atoms are discussed in chapters 16 and 17.

14.1 Introduction

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 of a given operator $\widehat{\mathbf{O}}$ for a system of N particles as

$$\langle\widehat{\mathbf{O}}\rangle = \frac{\int d\mathbf{R}_1 d\mathbf{R}_2\dots d\mathbf{R}_N \Psi^*(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)\widehat{\mathbf{O}}(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)\Psi(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)}{\int d\mathbf{R}_1 d\mathbf{R}_2\dots d\mathbf{R}_N \Psi^*(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)\Psi(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)}, \tag{14.1}$$

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 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

$$2^A \times \binom{A}{Z}$$

coupled second-order differential equations in $3A$ dimensions. For a nucleus like ^{16}O, with eight protons and eight neutrons this number is 8.4×10^8. This is a truely challenging many-body problem.

Equation (14.1) 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. (14.1).

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 16 and 17.

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 examples and applications from electronic systems with few electrons.

14.2 Postulates of Quantum Mechanics

14.2.1 Mathematical Properties of the Wave Functions

Schrödinger's equation for a one-dimensional onebody problem reads

$$-\frac{\hbar^2}{2m}\nabla^2\Psi(x,t) + V(x,t)\Psi(x,t) = \imath\hbar\frac{\partial\Psi(x,t)}{\partial t},$$

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]

$$P(x,t)dx = \Psi(x,t)^*\Psi(x,t)dx,$$

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,

$$\Psi(x,t) = R(x,t) + \imath I(x,t),$$

[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.

yielding

$$\Psi(x,t)^*\Psi(x,t) = (R-\imath I)(R+\imath I) = R^2+I^2.$$

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

$$\frac{\partial\Psi(x,\tau)}{\partial\tau} = \frac{\hbar^2}{2m}\frac{\partial^2\Psi(x,\tau)}{\partial x^2} - V(x,\tau)\Psi(x,\tau).$$

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 spin being a multiplum of $1/2$. Examples of such particles are various leptons such as 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
$$\int_{-\infty}^{\infty} P(x,t)dx = \int_{-\infty}^{\infty}\Psi(x,t)^*\Psi(x,t)dx = 1,$$
meaning that
$$\int_{-\infty}^{\infty}\Psi(x,t)^*\Psi(x,t)dx < \infty.$$
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.

14.2.2 Important Postulates

We list here some of the postulates that we will use in our discussion, see for example [93] for further discussions.

14.2.2.1 Postulate I

Any physical quantity $A(r,p)$ which depends on position r and momentum p has a corresponding quantum mechanical operator by replacing p $-i\hbar\bigtriangledown$, yielding the quantum mechanical operator

$$\widehat{\mathbf{A}} = A(r,-i\hbar\bigtriangledown).$$

Quantity	Classical definition	Quantum mechanical operator
Position	r	$\widehat{r} = r$
Momentum	p	$\widehat{p} = -i\hbar\bigtriangledown$
Orbital momentum	$L = r\times p$	$\widehat{L} = r\times(-i\hbar\bigtriangledown)$
Kinetic energy	$T = (p)^2/2m$	$\widehat{\mathbf{T}} = -(\hbar^2/2m)(\bigtriangledown)^2$
Total energy	$H = (p^2/2m)+V(r)$	$\widehat{\mathbf{H}} = -(\hbar^2/2m)(\bigtriangledown)^2+V(r)$

14.2.2.2 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.

14.2.2.3 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 $A(r,p)$ is given by

$$\langle A \rangle = \int (\Phi)^* \widehat{\mathbf{A}}(r, -i\hbar\bigtriangledown)\Phi d\tau.$$

We have assumed that Φ has been normalized, viz., $\int (\Phi)^*\Phi d\tau = 1$. Else

$$\langle A \rangle = \frac{\int (\Phi)^* \widehat{\mathbf{A}} \Phi d\tau}{\int (\Phi)^* \Phi d\tau}.$$

14.2.2.4 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.

14.3 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 pertinent physics 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 $\langle H \rangle$

$$\langle H \rangle = \frac{\int d\mathbf{R}\Psi_T^*(\mathbf{R})H(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\Psi_T(\mathbf{R})}, \tag{14.2}$$

is an upper bound to the true ground state energy E_0 of the Hamiltonian H, that is

$$E_0 \le \langle H \rangle.$$

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,

$$\Psi_T(\mathbf{R}) = \sum_i a_i \Psi_i(\mathbf{R}),$$

and assuming the set of eigenfunctions to be normalized, insertion of the latter equation in Eq. (14.2) results in

$$\langle H \rangle = \frac{\sum_{mn} a_m^* a_n \int d\mathbf{R}\Psi_m^*(\mathbf{R})H(\mathbf{R})\Psi_n(\mathbf{R})}{\sum_{mn} a_m^* a_n \int d\mathbf{R}\Psi_m^*(\mathbf{R})\Psi_n(\mathbf{R})} = \frac{\sum_{mn} a_m^* a_n \int d\mathbf{R}\Psi_m^*(\mathbf{R})E_n(\mathbf{R})\Psi_n(\mathbf{R})}{\sum_n a_n^2},$$

which can be rewritten as

$$\frac{\sum_n a_n^2 E_n}{\sum_n a_n^2} \ge E_0.$$

In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. Traditional integration methods like Gaussian-quadrature discussed in chapter 5 will not be adequate for say the computation of the energy of a many-body system.

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 $\alpha = (\alpha_1,\dots,\alpha_m)$.
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 allows for an in principle exact solution to the many-body Schrödinger equation. 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.

14.4 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 11 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 defines the quantum-mechanical probability distribution

$$P(\mathbf{R};\alpha) = \frac{|\psi_T(\mathbf{R};\alpha)|^2}{\int |\psi_T(\mathbf{R};\alpha)|^2 d\mathbf{R}}.$$

This is our new probability distribution function.

The expectation value of the Hamiltonian is given by

$$\langle \widehat{\mathbf{H}} \rangle = \frac{\int d\mathbf{R}\Psi^*(\mathbf{R})H(\mathbf{R})\Psi(\mathbf{R})}{\int d\mathbf{R}\Psi^*(\mathbf{R})\Psi(\mathbf{R})},$$

where Ψ is the exact eigenfunction. Using our trial wave function we define a new operator, the so-called local energy

$$\widehat{\mathbf{E}}_L(\mathbf{R};\alpha) = \frac{1}{\psi_T(\mathbf{R};\alpha)}\widehat{\mathbf{H}}\psi_T(\mathbf{R};\alpha), \tag{14.3}$$

which, together with our trial probability distribution function allows us to compute the expectation value of the local energy

$$\langle E_L(\alpha)\rangle = \int P(\mathbf{R};\alpha)\widehat{\mathbf{E}}_L(\mathbf{R};\alpha)d\mathbf{R}. \tag{14.4}$$

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 $\langle \widehat{\mathbf{H}} \rangle$. Equation (14.4) 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

$$\widehat{\mathbf{H}} = -\frac{\hbar^2}{2m}\sum_{i=1}^{N}\nabla_i^2 + \sum_{i=1}^{N}V_{\mathrm{onebody}}(\mathbf{r}_i) + \sum_{i<j}^{N}V_{\mathrm{int}}(\mid \mathbf{r}_i - \mathbf{r}_j \mid). \tag{14.5}$$

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

$$\widehat{\mathbf{E}}_L(\mathbf{R};\alpha)=\frac{1}{\psi_T(\mathbf{R};\alpha)}\left(-\frac{\hbar^2}{2m}\sum_{i=1}^{N}\nabla_i^2+\sum_{i=1}^{N}V_{\mathrm{onebody}}(\mathbf{r}_i)+\sum_{i<j}^{N}V_{\mathrm{int}}(\mid\mathbf{r}_i-\mathbf{r}_j\mid)\right)\psi_T(\mathbf{R};\alpha),$$

resulting in

$$\widehat{\mathbf{E}}_L(\mathbf{R};\alpha)=\frac{1}{\psi_T(\mathbf{R};\alpha)}\left(-\frac{\hbar^2}{2m}\sum_{i=1}^{N}\nabla_i^2\right)\psi_T(\mathbf{R};\alpha)+\sum_{i=1}^{N}V_{\mathrm{onebody}}(\mathbf{r}_i)+\sum_{i<j}^{N}V_{\mathrm{int}}(\mid\mathbf{r}_i-\mathbf{r}_j\mid).$$

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 spatial dependence only, adds 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 resides in the form of the probability distribution function . The main test to be performed by the Metropolis algorithm is a ratio of probabilities, as discussed in chapter 12. Suppose we are attempting to move from position $\mathbf{R}$ to a new position $\mathbf{R}'$. We need to 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. $$r\leq\frac{P(\mathbf{R}';\alpha)}{P(\mathbf{R};\alpha)},$$
where r is random number generated with uniform probability distribution function 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 between probabilities.

When writing a variational Monte Carlo program, one should always prepare in advance the required formulae for the local energy E_L in Eq. (14.4) 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. (14.4), 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.

[2] This corresponds to the partition function Z in statistical physics.

- 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 $\mathbf{R}_p = \mathbf{R} + r * \Delta\mathbf{R}$ where r is a random variable $r \in [0,1]$ and $\Delta\mathbf{R}$ a user-chosen step length.
  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 $w \geq s$, where s is a random number $s \in [0,1]$, the new position is accepted, else we stay at the same place.
  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 $\mathbf{R}_p = \mathbf{R} + r * \Delta\mathbf{R}$ 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 and importance sampling later in this chapter. Here we note that the above algorithm will depend on the chosen value of $\Delta\mathbf{R}$. Normally, $\Delta\mathbf{R}$ is chosen in order to accept approximately 50% of the proposed moves. One refers often to this algorithm as the brute force Metropolis algorithm.

14.4.1 First illustration of Variational Monte Carlo Methods

The harmonic oscillator in one dimension lends itself nicely for illustrative purposes. The Hamiltonian is

$$H = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{1}{2}kx^2, \tag{14.6}$$

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

$$H = -\frac{d^2}{dx^2} + x^2,$$

The energy of the ground state is then $E_0 = 1$. The exact wave function for the ground state is

$$\Psi_0(x) = \frac{1}{\pi^{1/4}}e^{-x^2/2},$$

but since we wish to illustrate the use of Monte Carlo methods, we choose the trial function

$$\Psi_T(x) = \frac{\sqrt{\alpha}}{\pi^{1/4}}e^{-x^2\alpha^2/2}.$$

Inserting this function in the expression for the local energy in Eq. (14.3), we obtain the following expression for the local energy

$$E_L(x) = \alpha^2 + x^2(1-\alpha^4),$$

with the expectation value for the Hamiltonian of Eq. (14.4) given by

$$\langle E_L \rangle = \int_{-\infty}^{\infty} |\psi_T(x)|^2 E_L(x) dx,$$

which reads with the above trial wave function

$$\langle E_L \rangle = \frac{\int_{-\infty}^{\infty} dx e^{-x^2\alpha^2} \alpha^2 + x^2(1-\alpha^4)}{\int_{-\infty}^{\infty} dx e^{-x^2\alpha^2}}.$$

Using the fact that

$$\int_{-\infty}^{\infty} dx e^{-x^2\alpha^2} = \sqrt{\frac{\pi}{\alpha^2}},$$

we obtain

$$\langle E_L \rangle = \frac{\alpha^2}{2} + \frac{1}{2\alpha^2}.$$

and the variance

$$\sigma^2 = \frac{(\alpha^4-1)^2}{2\alpha^4}. \tag{14.7}$$

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, as seen in this listing

```
... 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 a gaussian probability distribution function (which resembles the ansatz for our trial wave function $\sim |\Psi_T|^2$) and for each random number we compute the local energy according to the approximation

$$\langle \widehat{\mathbf{E}}_L \rangle = \int P(\mathbf{R}) \widehat{\mathbf{E}}_L(\mathbf{R}) d\mathbf{R} \approx \frac{1}{N}\sum_{i=1}^{N} E_L(x_i),$$

and the energy squared through

$$\langle \widehat{\mathbf{E}}_L^2 \rangle = \int P(\mathbf{R}) \widehat{\mathbf{E}}_L^2(\mathbf{R}) d\mathbf{R} \approx \frac{1}{N}\sum_{i=1}^{N} E_L^2(x_i).$$

In a certain sense, this is nothing but the importance Monte Carlo sampling discussed in chapter 11. 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. The expectation value of the kinetic energy is

$$-\frac{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\nabla^2\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\Psi_T(\mathbf{R})}, \tag{14.8}$$

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. For quantum mechanical systems, as discussed below, the exact wave function leads to a variance which is exactly zero.

Another shortcut we could think of is to transform the numerator in the latter equation to

$$\int d\mathbf{R}\Psi_T^*(\mathbf{R})\nabla^2\Psi_T(\mathbf{R}) = -\int d\mathbf{R}(\nabla\Psi_T^*(\mathbf{R}))(\nabla\Psi_T(\mathbf{R})), \tag{14.9}$$

using integration by parts and the relation

$$\int d\mathbf{R}\nabla(\Psi_T^*(\mathbf{R})\nabla\Psi_T(\mathbf{R})) = 0,$$

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

$$\int d\mathbf{R}(\nabla\Psi_T^*(\mathbf{R}))(\nabla\Psi_T(\mathbf{R})) + \int d\mathbf{R}\Psi_T^*(\mathbf{R})\nabla^2\Psi_T(\mathbf{R})) = 0.$$

The right-hand side of Eq. (14.9) involves only first derivatives. However, in case the wave function is the exact one, or rather close to the exact one, the left-hand side 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 parts for the harmonic oscillator case, the new local energy is

$$E_L(x) = x^2(1+\alpha^4),$$

and the variance

$$\sigma^2 = \frac{(\alpha^4+1)^2}{2\alpha^4},$$

which is larger than the variance of Eq. (14.7).

14.5 Variational Monte Carlo for atoms

The Hamiltonian for an N-electron atomic system consists of two terms

$$\hat{H}(\mathbf{R}) = \hat{T}(\mathbf{R}) + \hat{V}(\mathbf{R}), \tag{14.10}$$

the kinetic and the potential energy operator. Here $\mathbf{R} = \{\mathbf{r}_1, \mathbf{r}_2, \dots \mathbf{r}_N\}$ represents 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 ($p_k \to -i\hbar\partial/\partial x_k$)

$$\hat{T}(\mathbf{R}) = -\frac{\hbar^2}{2M}\nabla_0^2 - \sum_{i=1}^{N}\frac{\hbar^2}{2m}\nabla_i^2. \qquad (14.11)$$

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

$$\hat{V}(\mathbf{R}) = -\sum_{i=1}^{N}\frac{Ze^2}{(4\pi\varepsilon_0)r_i} + \sum_{i=1,i<j}^{N}\frac{e^2}{(4\pi\varepsilon_0)r_{ij}}, \qquad (14.12)$$

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 this approximation, the motion of the nucleus is disregarded.

14.5.1 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. (14.10), (14.11) and (14.12) 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

$$\hat{T}(\mathbf{R}) = -\frac{\hbar^2}{2(M+Nm)}\nabla_{CM}^2 - \frac{\hbar^2}{2\mu}\sum_{i=1}^{N}\nabla_i^2 - \frac{\hbar^2}{M}\sum_{i>j}^{N}\nabla_i\cdot\nabla_j, \qquad (14.13)$$

while the potential energy operator remains unchanged. Note that the Laplace operators ∇_i^2 now are in the center of mass reference system.

The first term of Eq. (14.13) 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 $1/1839$. We can therefore approximate the nucleus as stationary with respect to the electrons. Taking the limit $M \to \infty$ in Eq. (14.13), the kinetic energy operator reduces to

$$\hat{T} = -\sum_{i=1}^{N}\frac{\hbar^2}{2m}\nabla_i^2$$

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 closed-form 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. (14.12) is the nucleus-electron potential and the second term is the electron-electron potential. The inter-electronic potential is the main problem in atomic physics. Because of this term, 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. Popular methods in this direction are Hartree-Fock theory and Density Functional theory. These approaches are actually very efficient, and about 99% or more of the electronic energies are obtained for most Hartree-Fock calculations. Other observables are usually obtained to an accuracy of about $90-95\%$ (ref. [94]). We discuss these methods in chapter 15, where also systems with more than two electrons are discussed in more detail. Here we limit ourselves to systems with at most two electrons. Relevant systems are neutral helium with two electrons, the hydrogen molecule or two electrons confined in a two-dimensional harmonic oscillator trap.

14.5.2 The Hydrogen Atom

The spatial Schrödinger equation for the three-dimensional hydrogen atom can be solved in a closed form, see for example Ref. [93] for details. To achieve this, we rewrite the equation in terms of spherical coordinates using

$$x = r\sin\theta\cos\phi,$$

$$y = r\sin\theta\sin\phi,$$

and

$$z = r\cos\theta.$$

The reason we introduce spherical coordinates is due to the spherical symmetry of the Coulomb potential

$$\frac{e^2}{4\pi\varepsilon_0 r} = \frac{e^2}{4\pi\varepsilon_0\sqrt{x^2+y^2+z^2}},$$

where we have used $r=\sqrt{x^2+y^2+z^2}$. It is not possible to find a separable solution of the type

$$\psi(x,y,z) = \psi(x)\psi(y)\psi(z).$$

as we can with the harmonic oscillator in three dimensions. However, with spherical coordinates we can find a solution of the form

$$\psi(r,\theta,\phi) = R(r)P(\theta)F(\phi) = RPF.$$

These three coordinates yield in turn three quantum numbers which determine the energy of the system. We obtain three sets of ordinary second-order differential equations [93],

$$\frac{1}{F}\frac{\partial^2 F}{\partial\phi^2} = -C^2_\phi,$$

$$C_r\sin^2(\theta)P + \sin(\theta)\frac{\partial}{\partial\theta}(\sin(\theta)\frac{\partial P}{\partial\theta}) = C^2_\phi P,$$

and

$$\frac{1}{R}\frac{\partial}{\partial r}(r^2\frac{\partial R}{\partial r})+\frac{2mrke^2}{\hbar^2}+\frac{2mr^2}{\hbar^2}E=C_r, \tag{14.14}$$

where C_r and C_ϕ are constants. The angle-dependent differential equations result in the so-called spherical harmonic functions as solutions, with quantum numbers l and m_l. These functions are given by

$$Y_{lm_l}(\theta,\phi)=P(\theta)F(\phi)=\sqrt{\frac{(2l+1)(l-m_l)!}{4\pi(l+m_l)!}}P_l^{m_l}(\cos(\theta))\exp(im_l\phi),$$

with $P_l^{m_l}$ being the associated Legendre polynomials. They can be rewritten as

$$Y_{lm_l}(\theta,\phi)=\sin^{|m_l|}(\theta)\times(\mathrm{polynom}(\cos\theta))\exp(im_l\phi),$$

with the following selected examples

$$Y_{00}=\sqrt{\frac{1}{4\pi}},$$

for $l=m_l=0$,

$$Y_{10}=\sqrt{\frac{3}{4\pi}}\cos(\theta),$$

for $l=1$ og $m_l=0$,

$$Y_{1\pm 1}=\mp 1\sqrt{\frac{3}{8\pi}}\sin(\theta)\exp(\pm i\phi),$$

for $l=1$ og $m_l=\pm 1$, and

$$Y_{20}=\sqrt{\frac{5}{16\pi}}(3\cos^2(\theta)-1)$$

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 $l\geq 0$, $l=0,1,2,\dots$ and $m_l=-l,-l+1,\dots,l-1,l$. The spherical harmonics for $l\leq 3$ are listed in Table 14.1.

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		$-\frac{1}{2}(\frac{3}{2\pi})^{1/2}\sin\theta e^{+i\phi}$	$-\frac{1}{2}(\frac{15}{2\pi})^{1/2}\cos\theta\sin\theta e^{+i\phi}$	$-\frac{1}{8}(\frac{21}{2\pi})^{1/2}(5\cos^2\theta-1)\sin\theta e^{+i\phi}$
0	$\frac{1}{2\pi^{1/2}}$	$\frac{1}{2}(\frac{3}{\pi})^{1/2}\cos\theta$	$\frac{1}{4}(\frac{5}{\pi})^{1/2}(3\cos^2\theta-1)$	$\frac{1}{4}(\frac{7}{\pi})^{1/2}(2-5\sin^2\theta)\cos\theta$
-1		$+\frac{1}{2}(\frac{3}{2\pi})^{1/2}\sin\theta e^{-i\phi}$	$+\frac{1}{2}(\frac{15}{2\pi})^{1/2}\cos\theta\sin\theta e^{-i\phi}$	$+\frac{1}{8}(\frac{21}{2\pi})^{1/2}(5\cos^2\theta-1)\sin\theta e^{-i\phi}$
-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 14.1 Spherical harmonics Y_{lm_l} for the lowest l and m_l values.

We focus now on the radial equation, which can be rewritten as

$$-\frac{\hbar^2 r^2}{2m}\left(\frac{\partial}{\partial r}(r^2\frac{\partial R(r)}{\partial r})\right)-\frac{ke^2}{r}R(r)+\frac{\hbar^2 l(l+1)}{2mr^2}R(r)=ER(r).$$

Introducing the function $u(r)=rR(r)$, we can rewrite the last equation as

$$-\frac{\hbar^2}{2m}\frac{\partial^2 u(r)}{\partial r^2}-\left(\frac{ke^2}{r}-\frac{\hbar^2 l(l+1)}{2mr^2}\right)u(r)=Eu(r), \tag{14.15}$$

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

$$V_{\mathrm{eff}}(r)=-\frac{ke^2}{r}+\frac{\hbar^2 l(l+1)}{2mr^2}.$$

The radial equation yield closed form solutions resulting in the quantum number n, in addition to lm_l. The solution R_{nl} to the radial equation is given by the Laguerre polynomials [93]. The closed-form solutions are given by

$$\psi_{nlm_l}(r,\theta,\phi)=\psi_{nlm_l}=R_{nl}(r)Y_{lm_l}(\theta,\phi)=R_{nl}Y_{lm_l}$$

The ground state is defined by $n=1$ and $l=m_l=0$ and reads

$$\psi_{100}=\frac{1}{a_0^{3/2}\sqrt{\pi}}\exp(-r/a_0),$$

where we have defined the Bohr radius a_0

$$a_0=\frac{\hbar^2}{mke^2},$$

with length $a_0=0.05$ nm. The first excited state with $l=0$ is

$$\psi_{200}=\frac{1}{4a_0^{3/2}\sqrt{2\pi}}\left(2-\frac{r}{a_0}\right)\exp(-r/2a_0).$$

For states with with $l=1$ and $n=2$, we can have the following combinations with $m_l=0$

$$\psi_{210}=\frac{1}{4a_0^{3/2}\sqrt{2\pi}}\left(\frac{r}{a_0}\right)\exp(-r/2a_0)\cos(\theta),$$

and $m_l=\pm 1$

$$\psi_{21\pm1}=\frac{1}{8a_0^{3/2}\sqrt{\pi}}\left(\frac{r}{a_0}\right)\exp(-r/2a_0)\sin(\theta)\exp(\pm i\phi).$$

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 14.2.

When solving equations numerically, it is often convenient to rewrite the equation in terms of dimensionless variables. This leads to an equation in dimensionless form which is easier to code, sparing one for eventual 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

$$-\frac{1}{2}\frac{\partial^2 u(\rho)}{\partial\rho^2}-\frac{mke^2\beta}{\hbar^2\rho}u(\rho)+\frac{l(l+1)}{2\rho^2}u(\rho)=\frac{m\beta^2}{\hbar^2}Eu(\rho). \tag{14.16}$$

Hydrogen-like atomic radial functions

$l\backslash n$	1	2	3
0	$\exp(-Zr)$	$(2-r)\exp(-Zr/2)$	$(27-18r+2r^2)\exp(-Zr/3)$
1		$r\exp(-Zr/2)$	$r(6-r)\exp(-Zr/3)$
2			$r^2\exp(-Zr/3)$

Table 14.2 The first few radial functions of the hydrogen-like atoms.

We can determine β by simply requiring[3]

$$\frac{mke^2\beta}{\hbar^2}=1 \qquad (14.17)$$

With this choice, the constant β becomes the famous Bohr radius $a_0 = 0.05$ nm $a_0 = \beta = \hbar^2/mke^2$. We list here the standard units used in atomic physics and molecular physics calculations. It is common to scale atomic units by setting $m = e = \hbar = 4\pi\varepsilon_0 = 1$, see Table 14.3. We

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, $4\pi\varepsilon_0$	$1.113\cdot 10^{-10}$ C^2 J^{-1} m^{-1}	1
Energy, $\frac{e^2}{4\pi\varepsilon_0 a_0}$	27.211 eV	1
Length, $a_0 = \frac{4\pi\varepsilon_0\hbar^2}{me^2}$	$0.529\cdot 10^{-10}$ m	1

Table 14.3 Scaling from SI units to atomic units.

introduce thereafter the variable λ

$$\lambda = \frac{m\beta^2}{\hbar^2}E,$$

and inserting β and the exact energy $E = E_0/n^2$, with $E_0 = 13.6$ eV, we have that

$$\lambda = -\frac{1}{2n^2},$$

n being the principal quantum number. The equation we are then going to solve numerically is now

$$-\frac{1}{2}\frac{\partial^2 u(\rho)}{\partial\rho^2} - \frac{u(\rho)}{\rho} + \frac{l(l+1)}{2\rho^2}u(\rho) - \lambda u(\rho) = 0, \qquad (14.18)$$

with the Hamiltonian

$$H = -\frac{1}{2}\frac{\partial^2}{\partial\rho^2} - \frac{1}{\rho} + \frac{l(l+1)}{2\rho^2}.$$

[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. (14.18) is

$$u(\rho) = \rho e^{-\rho},$$

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

$$u_T^{\alpha}(\rho) = \alpha\rho e^{-\alpha\rho}. \qquad (14.19)$$

Inserting this wave function into the expression for the local energy E_L of Eq. (14.3) yields

$$E_L(\rho) = -\frac{1}{\rho} - \frac{\alpha}{2}\left(\alpha - \frac{2}{\rho}\right). \qquad (14.20)$$

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. (14.4) now reads

$$\langle H \rangle = \int P(\mathbf{R})E_L(\mathbf{R})d\mathbf{R} = \int_0^{\infty} \alpha^2\rho^2 e^{-2\alpha\rho} E_L(\rho)\rho^2 d\rho,$$

since $\rho \in [0,\infty)$. In this case we would use the exponential distribution instead of the normal distrubution, and our code could contain the following program statements

```
... 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 need to generate a large number N of random numbers corresponding to the exponential probability distribution function $\alpha^2\rho^2 e^{-2\alpha\rho}$ and for each random number we compute the local energy and variance.

14.5.3 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 $x \in (-\infty,\infty)$. In the calculations below we have used a uniform distribution to generate the various positions. This means that we employ a shifted uniform distribution where the integration regions beyond a given value of ρ and x are omitted. This is obviously an approximation and techniques like importance sampling discussed in chapter 11 should be used. Using a uniform distribution is normally refered to as brute force Monte Carlo or brute force Metropolis sampling. From a practical point of view, this means that the random variables are multiplied by a given step length λ. To better understand this, consider the above dimensionless radius $\rho \in [0,\infty)$.

The new position can then be modelled as

$$\rho_{\text{new}} = \rho_{\text{old}} + \lambda \times r,$$

with r being a random number drawn from the uniform distribution in a region $r \in [0,\Lambda]$, with $\Lambda < \infty$, a cutoff large enough in order to have a contribution to the integrand close to zero. The step length λ is chosen to give approximately an acceptance ratio of 50% for all proposed moves. This is nothing but a simple rule of thumb. In this chapter we will stay with this brute force Metropolis algorithm. All results discussed here have been obtained with this approach. Importance sampling and further improvements will be discussed in chapter 15. In Figs. 14.1 and 14.2 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 14.4 and 14.5. In these tables we list the variance and the standard deviation as well. We note that at $\alpha = 1$ for the hydrogen atom, we obtain

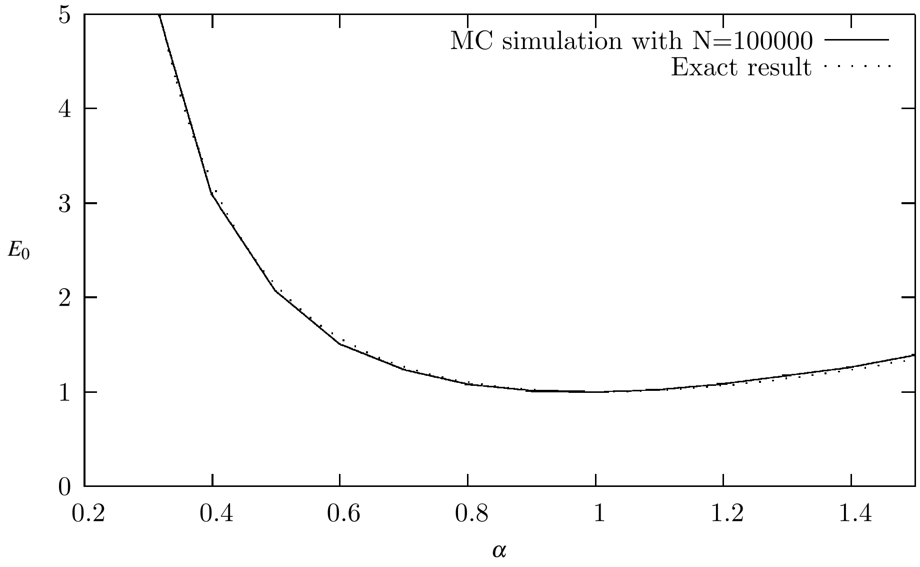


Fig. 14.1 Result for ground state energy of the harmonic oscillator as function of the variational parameter α. The exact result is for $\alpha = 1$ with an energy $E = 1$. See text for further details.

the exact result, and the variance is zero, as it should. The reason is that we have used the exact wave function, and the action of the hamiltionan on the wave function

$$H\psi = \text{constant} \times \psi,$$

yields just a constant. The integral which defines various expectation values involving moments of the Hamiltonian becomes then

$$\langle H^n \rangle = \frac{\int d\mathbf{R}\Psi_T^*(\mathbf{R})H^n(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\Psi_T(\mathbf{R})} = \text{constant} \times \frac{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\Psi_T(\mathbf{R})} = \text{constant}.$$

This explains why the variance is zero for $\alpha = 1$. However, the hydrogen atom and the har-

Table 14.4 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 $E = 1$. We list the energy and the variance σ^2 as well. 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
5.00000E-01	2.06479E+00	5.78739E+00
6.00000E-01	1.50495E+00	2.32782E+00
7.00000E-01	1.23264E+00	9.82479E-01
8.00000E-01	1.08007E+00	3.44857E-01
9.00000E-01	1.01111E+00	7.24827E-02
1.00000E-00	1.00000E+00	0.00000E+00
1.10000E+00	1.02621E+00	5.95716E-02
1.20000E+00	1.08667E+00	2.23389E-01
1.30000E+00	1.17168E+00	4.78446E-01
1.40000E+00	1.26374E+00	8.55524E-01
1.50000E+00	1.38897E+00	1.30720E+00

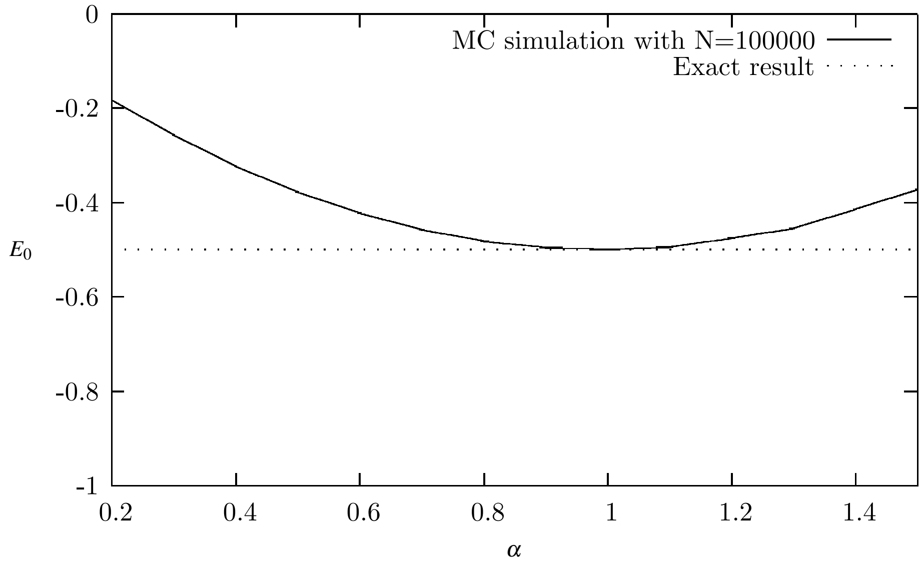


Fig. 14.2 Result for ground state energy of the hydrogen atom as function of the variational parameter α. The exact result is for $\alpha = 1$ with an energy $E = -1/2$. See text for further details.

monic oscillator are some of the few cases where we can use a trial wave function proportional to the exact one. These two systems offer some of the few examples 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.

Table 14.5 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 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
5.00000E-01	-3.76740E-01	6.10503E-02
6.00000E-01	-4.21744E-01	5.22322E-02
7.00000E-01	-4.57759E-01	4.51201E-02
8.00000E-01	-4.81461E-01	3.05736E-02
9.00000E-01	-4.95899E-01	8.20497E-03
1.00000E-00	-5.00000E-01	0.00000E+00
1.10000E+00	-4.93738E-01	1.16989E-02
1.20000E+00	-4.75563E-01	8.85899E-02
1.30000E+00	-4.54341E-01	1.45171E-01
1.40000E+00	-4.13220E-01	3.14113E-01
1.50000E+00	-3.72241E-01	5.45568E-01

14.5.4 The Helium Atom

Most physical problems of interest in atomic, molecular and solid state physics consist of many interacting electrons and ions. The total number of particles N is usually sufficiently large that an exact solution cannot be found. 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 of the Schrödinger equation for systems of interacting electrons and ions is the Born-Oppenheimer approximation discussed above. But even this simplified electronic Hamiltonian remains very difficult to solve. No closed-form solutions exist for general systems with more than one electron.

To set up the problem, we start by labelling 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

$$-\frac{2ke^2}{r_1} - \frac{2ke^2}{r_2},$$

and if we add the repulsion arising from the two interacting electrons, we obtain the potential energy

$$V(r_1,r_2) = -\frac{2ke^2}{r_1} - \frac{2ke^2}{r_2} + \frac{ke^2}{r_{12}},$$

with the electrons separated at a distance $r_{12} = |\mathbf{r}_1 - \mathbf{r}_2|$. The Hamiltonian becomes then

$$\widehat{\mathbf{H}} = -\frac{\hbar^2\nabla_1^2}{2m} - \frac{\hbar^2\nabla_2^2}{2m} - \frac{2ke^2}{r_1} - \frac{2ke^2}{r_2} + \frac{ke^2}{r_{12}},$$

and Schrödingers equation reads

$$\widehat{\mathbf{H}}\psi = E\psi.$$

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

$$V(r_1,r_2) \approx -\frac{Zke^2}{r_1} - \frac{Zke^2}{r_2}.$$

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 central 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

$$\widehat{\mathbf{h}}_1 = -\frac{\hbar^2\nabla_1^2}{2m} - \frac{2ke^2}{r_1},$$

with pertinent wave function and eigenvalue E_a

$$\widehat{\mathbf{h}}_1\psi_a = E_a\psi_a,$$

where $a = \{n_a l_a m_{l_a}\}$ are the relevant quantum numbers needed to describe the system. We assume here that we can use the hydrogen-like solutions, but with Z not necessarily equal to one. The energy E_a is

$$E_a = -\frac{Z^2E_0}{n_a^2}.$$

In a similar way, we obtain for electron 2

$$\widehat{\mathbf{h}}_2 = -\frac{\hbar^2\nabla_2^2}{2m} - \frac{2ke^2}{r_2},$$

with wave function ψ_b, $b = \{n_b l_b m_{l_b}\}$ and energy

$$E_b = \frac{Z^2E_0}{n_b^2}.$$

Since the electrons do not interact, the ground state wave function of the helium atom is given by

$$\psi = \psi_a\psi_b,$$

resulting in the following approximation to Schrödinger's equation

$$\left(\widehat{\mathbf{h}}_1 + \widehat{\mathbf{h}}_2\right)\psi = \left(\widehat{\mathbf{h}}_1 + \widehat{\mathbf{h}}_2\right)\psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2) = E_{ab}\psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2).$$

The energy becomes then

$$\left(\widehat{\mathbf{h}}_1 \psi_a(\mathbf{r}_1)\right) \psi_b(\mathbf{r}_2) + \left(\widehat{\mathbf{h}}_2 \psi_b(\mathbf{r}_2)\right) \psi_a(\mathbf{r}_1) = (E_a + E_b)\, \psi_a(\mathbf{r}_1) \psi_b(\mathbf{r}_2),$$

yielding

$$E_{ab} = Z^2 E_0 \left(\frac{1}{n_a^2} + \frac{1}{n_b^2}\right).$$

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

$$E_{ab} = 8E_0 = -108.8 \text{ eV},$$

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.

14.5.4.1 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

$$P(\mathbf{R}) = \frac{|\psi_T(\mathbf{R})|^2}{\int |\psi_T(\mathbf{R})|^2 d\mathbf{R}}.$$

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, the gradient and the 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

$$E_L(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})} H \psi_T(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})} \left(-\frac{1}{2}\nabla_1^2 - \frac{Z}{r_1}\right) \psi_T(\mathbf{R}) + \text{finite terms.}$$

Writing out the kinetic energy term in the spherical coordinates of electron 1, we arrive at the following expression for the local energy

$$E_L(R) = \frac{1}{\mathscr{R}_T(r_1)} \left(-\frac{1}{2}\frac{d^2}{dr_1^2} - \frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1}\right) \mathscr{R}_T(r_1) + \text{finite terms,}$$

where $\mathscr{R}_T(r_1)$ 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

$$\lim_{r_1 \to 0} E_L(R) = \frac{1}{\mathscr{R}_T(r_1)} \left(-\frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1} \right) \mathscr{R}_T(r_1),$$

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

$$\frac{1}{\mathscr{R}_T(r_1)} \frac{d\mathscr{R}_T(r_1)}{dr_1} = -Z,$$

implying that

$$\mathscr{R}_T(r_1) \propto e^{-Zr_1}.$$

A similar condition applies to electron 2 as well. For orbital momenta $l > 0$ it is rather straightforward to show that

$$\frac{1}{\mathscr{R}_T(r)} \frac{d\mathscr{R}_T(r)}{dr} = -\frac{Z}{l+1}.$$

Another constraint on the wave function is found when the two electrons are 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.

To find an ansatz for the correlated part of the wave function, it is useful to rewrite the two-particle local energy in terms of the relative and center-of-mass motion. Let us denote the distance between the two electrons as r_{12}. We omit the center-of-mass motion since we are only interested in the case when $r_{12} \to 0$. The contribution from the center-of-mass (CoM) variable $\mathbf{R}_{\mathrm{CoM}}$ gives only a finite contribution. We focus only on the terms that are relevant for r_{12}. The relevant local energy becomes then

$$\lim_{r_{12} \to 0} E_L(R) = \frac{1}{\mathscr{R}_T(r_{12})} \left(2\frac{d^2}{dr_{ij}^2} + \frac{4}{r_{ij}}\frac{d}{dr_{ij}} + \frac{2}{r_{ij}} - \frac{l(l+1)}{r_{ij}^2} + 2E \right) \mathscr{R}_T(r_{12}) = 0,$$

where l is now equal 0 if the spins of the two electrons are anti-parallel and 1 if they are parallel. Repeating the argument for the electron-nucleus cusp with the factorization of the leading r-dependency, we get the similar cusp condition:

$$\frac{d\mathscr{R}_T(r_{12})}{dr_{12}} = -\frac{1}{2(l+1)} \mathscr{R}_T(r_{12}) \qquad r_{12} \to 0$$

resulting in

$$\mathscr{R}_T \propto \begin{cases} \exp(r_{ij}/2) & \text{for anti-parallel spins, } l = 0 \\ \\ \exp(r_{ij}/4) & \text{for parallel spins, } l = 1 \end{cases}.$$

This is so-called cusp condition for the relative motion, resulting in a minimal requirement for the correlation part of the wave fuction. 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

$$\psi_T(\mathbf{R}) = e^{-\alpha(r_1+r_2)}, \tag{14.21}$$

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

$$\psi_T(\mathbf{R}) = e^{-\alpha(r_1+r_2)}e^{r_{12}/2}. \tag{14.22}$$

The last equation can be generalized to

$$\psi_T(\mathbf{R}) = \phi(\mathbf{r}_1)\phi(\mathbf{r}_2)\dots\phi(\mathbf{r}_N)\prod_{i<j} f(r_{ij}),$$

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 the ground state of 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. One way to account for this is by introducing the so-called Slater determinant (to be discussed in more detail in chapter 15). This determinant is written in terms of the various single-particle wave functions.

If we consider the helium atom with two electrons in the $1s$ state, we can write the total Slater determinant as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\alpha,\beta) = \frac{1}{\sqrt{2}}\begin{vmatrix} \psi_\alpha(\mathbf{r}_1) & \psi_\alpha(\mathbf{r}_2) \\ \psi_\beta(\mathbf{r}_1) & \psi_\beta(\mathbf{r}_2) \end{vmatrix},$$

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 $\beta = nlm_lsm_s = (1001/2\downarrow)$. It is normal to skip the two quantum numbers sm_s of the one-electron spin. We introduce therefore the shorthand $nlm_l\uparrow$ or $nlm_l\downarrow$) for a particular state where an arrow pointing upward represents $m_s = 1/2$ and a downward arrow stands for $m_s = -1/2$. Writing out the Slater determinant

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\alpha,\beta) = \frac{1}{\sqrt{2}}\left[\psi_\alpha(\mathbf{r}_1)\psi_\beta(\mathbf{r}_2) - \psi_\beta(\mathbf{r}_1)\psi_\gamma(\mathbf{r}_2)\right],$$

we see that the Slater determinant is antisymmetric with respect to the permutation of two particles, that is

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\alpha,\beta) = -\Phi(\mathbf{r}_2,\mathbf{r}_1,\alpha,\beta).$$

The Slater determinant obeys the cusp condition for the two electrons and combined with the correlation part we could write the ansatz for the wave function as

$$\psi_T(\mathbf{R}) = \frac{1}{\sqrt{2}}\left[\psi_\alpha(\mathbf{r}_1)\psi_\beta(\mathbf{r}_2) - \psi_\beta(\mathbf{r}_1)\psi_\gamma(\mathbf{r}_2)\right] f(r_{12}),$$

Several forms of the correlation function $f(r_{ij})$ exist in the literature and we will mention only a selected few to give the general idea of how they are constructed. A form given by Hylleraas that had great success for the helium atom was the series expansion

$$f(r_{ij}) = \exp(\varepsilon s)\sum_k c_k r^{l_k} s^{m_k} t^{n_k}$$

where the inter-particle separation r_{ij} for simplicity is written as r. In addition $s = r_i + r_i$ and $t = r_i - r_i$ with r_i and r_j being the two electron-nucleus distances. All the other quantities are free parameters. Notice that the cusp condition is satisfied by the exponential. Unfortunately the convergence of this function turned out to be quite slow. For example, to pinpoint the He-energy to the fourth decimal digit a nine term function would suffice. To double the number of digits, one needed almost 1100 terms.

The so called Padé-Jastrow form, however, is more suited for larger systems. It is based on an exponential function with a rational exponent:

$$f(r_{ij}) = \exp(U)$$

In its general form, U is a potential series expansion on both the absolute particle coordinates r_i and the inter-particle coordinates r_{ij}:

$$U = \sum_{i<j}^{N} \left(\frac{\sum_k \alpha_k r_i^k}{1 + \sum_k \alpha'_k r_i^k} \right) + \sum_{i}^{N} \left(\frac{\sum_k \beta_k r_{ij}^k}{1 + \sum_k \beta'_k r_{ij}^k} \right)$$

A typical Padé-Jastrow function used for quantum mechanical Monte Carlo calculations of molecular and atomic systems is

$$\exp\left(\frac{a r_{ij}}{(1 + \beta r_{ij})} \right)$$

where β is a variational parameter and a dependes on the spins of the interacting particles.

14.5.5 Program Example for Atomic Systems

The variational Monte Carlo algorithm consists of two distinct parts. 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 part, 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 15.

http://folk.uio.no/compphys/programs/chapter14/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;
```

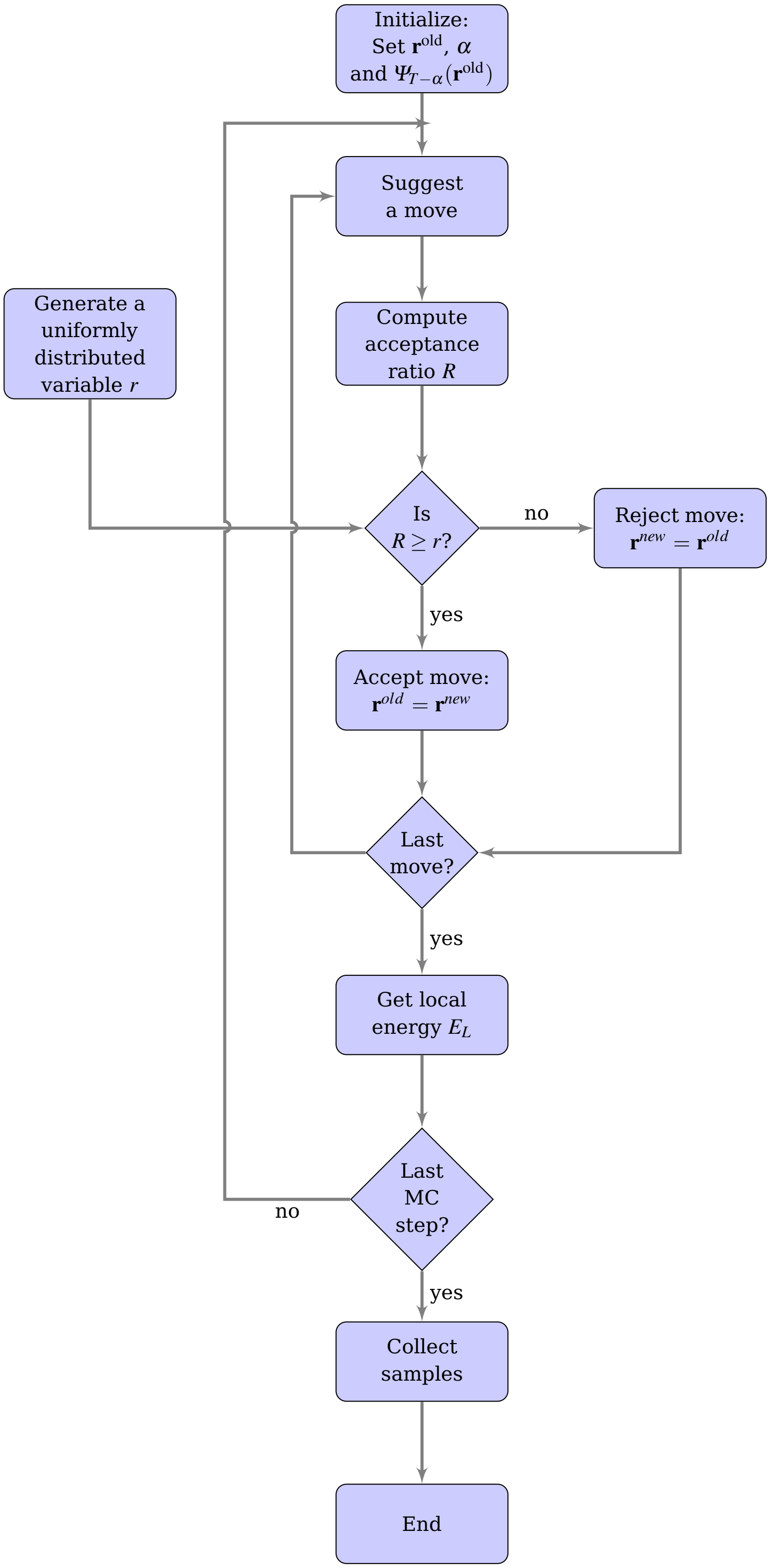


Fig. 14.3 Chart flow for the Quantum Varitional Monte Carlo algorithm.

```
// 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

$$-\frac{1}{2\psi_T(\mathbf{R})}\nabla^2\psi_T(\mathbf{R}). \tag{14.23}$$

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 a closed-form 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.

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. (14.21) for the trial wave function, we obtain an energy minimum at $\alpha =$

1.6875[4]. 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. (14.22), could improve our agreement with experiment.

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. 14.4 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 in the next section and in chapter 17.

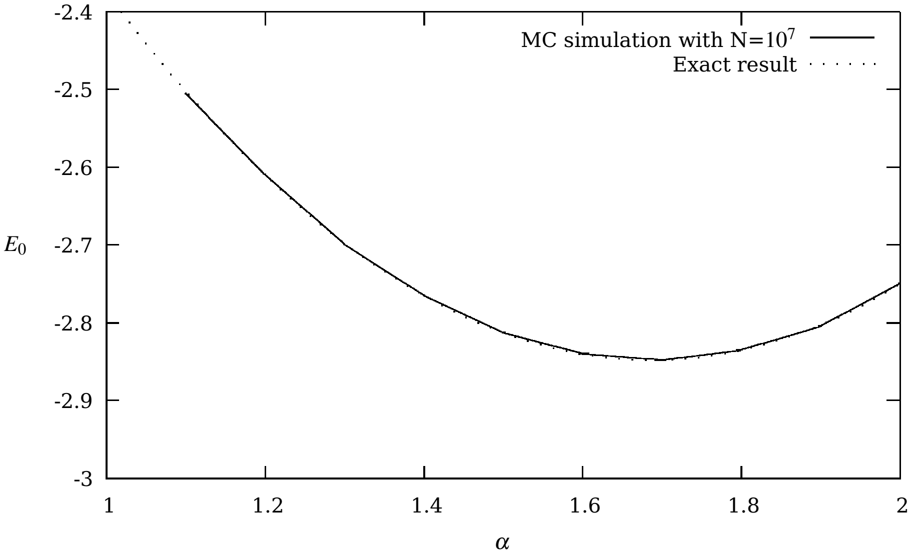


Fig. 14.4 Result for ground state energy of the helium atom using Eq. (14.21) for the trial wave function. A total of 10^7 Monte Carlo moves were used with a step length of 1 Bohr radius. Approximately 50% of all proposed moves were accepted. The variance at the minimum is 1.026, reflecting the fact that we do not have the exact wave function. The variance has a minimum at value of α different from the energy minimum. The numerical results are compared with the exact result $E[Z] = Z^2 - 4Z + \frac{5}{8}Z$.

14.5.6 Importance sampling

As mentioned in connection with the generation of random numbers, sequential correlations must be given thorough attention as it may lead to bad error estimates of our numerical results.

There are several things we need to keep in mind in order to keep the correlation low. First of all, the transition acceptance must be kept as high as possible. Otherwise, a walker will dwell at the same spot in state space for several iterations at a time, which will clearly lead to high correlation between nearby succeeding measurements.

[4] With hydrogen like wave functions for the $1s$ state one can easily calculate the energy of the ground state for the helium atom as function of the charge Z. The results is $E[Z] = Z^2 - 4Z + \frac{5}{8}Z$, and taking the derivative with respect to Z to find the minumum we get $Z = 2 - \frac{5}{16} = 1.6875$. This number represents an optimal effective charge.

Secondly, when using the simple symmetric form of $\omega(r_{\rm old}, r_{\rm new})$, one has to keep in mind the random walk nature of the algorithm. Transitions will be made between points that are relatively close to each other in state space, which also clearly contributes to increase correlation. The seemingly obvious way to deal with this would be just to increase the step size, allowing the walkers to cover more of the state space in fewer steps (thus requiring fewer steps to reach ergodicity). But unfortunately, long before the step length becomes desirably large, the algorithm breaks down. When proposing moves symmetrically and uniformly around $r_{\rm old}$, the step acceptance becomes directly dependent on the step length in such a way that a too large step length reduces the acceptance. The reason for this is very simple. As the step length increases, a walker will more likely be given a move proposition to areas of very low probability, particularly if the governing trial wave function describes a localized system. In effect, the effective movement of the walkers again becomes too small, resulting in large correlation. For optimal results we therefore have to balance the step length with the acceptance.

With a transition suggestion rule ω as simple as the uniform symmetrical one emphasized so far, the usual rule of thumb is to keep the acceptance around 0.5. But the optimal interval varies a lot from case to case. We therefore have to treat each numerical experiment with care.

By choosing a better ω, we can still improve the efficiency of the step length versus acceptance. Recall that ω may be chosen arbitrarily as long as it fulfills ergodicity, meaning that it has to allow the walker to reach any point of the state space in a finite number of steps. What we basically want is an ω that pushes the ratio towards unity, increasing the acceptance. The theoretical situation of ω exactly equal to p itself:

$$\omega(r_{\rm new}, r_{\rm old}) = \omega(r_{\rm new}) = p(r_{\rm new})$$

would give the maximal acceptance of 1. But then we would already have solved the problem of producing points distributed according to p. One typically settles on modifying the symmetrical ω so that the walkers move more towards areas of the state space where the distribution is large. One such procedure is the Fokker-Planck formalism where the walkers are moved according to the gradient of the distribution. The formalism “pushes” the walkers in a “desirable” direction. The idea is to propose moves similarly to an isotropic diffusion process with a drift. A new position $x_{\rm new}$ is calculated from the old one, $x_{\rm old}$, as follows:

$$r_{\rm new} = r_{\rm old} + \chi + DF(r_{\rm old})\delta t \qquad (14.24)$$

Here χ is a Gaussian pseudo-random number with mean equal zero and variance equal $2D\delta t$. It accounts for the diffusion part of the transition. The third term on the left hand side accounts for the drift. F is a drift velocity dependent on the position of the walker and is derived from the quantum mechanical wave function ψ. The constant D, being the diffusion constant of χ, also adjusts the size of the drift. δt is a time step parameter whose presence will be clarified shortly.

It can be shown that the ω corresponding to the move proposition rule in Eq. (14.24) becomes (in non-normalized form):

$$\omega(r_{\rm old}, r_{\rm new}) = \exp\left(-\frac{(r_{\rm new} - r_{\rm old} - D\delta t F(r_{\rm old}))^2}{4D\delta t}\right) \qquad (14.25)$$

which, as expected, is a Gaussian with variance $2D\delta t$ centered slightly off $r_{\rm old}$ due to the drift term $DF(r_{\rm old})\delta t$.

What is the optimal choice for the drift term? From statistical mechanics we know that a simple isotropic drift diffusion process obeys a Fokker-Planck equation of the form:

$$\frac{\partial f}{\partial t} = \sum_i D \frac{\partial}{\partial x_i}\left(\frac{\partial}{\partial x_i} - F_i(F)\right) f \tag{14.26}$$

where f is the continuous distribution of walkers. Equation (14.24) is a discretized realization of such a process where δt is the discretized time step. In order for the solution f to converge to the desired distribution p, it can be shown that the drift velocity has to be chosen as follows:

$$F = \frac{1}{f}\nabla f$$

where the operator ∇ is the vector of first derivatives of all spatial coordinates. Convergence for such a diffusion process is only guaranteed when the time step approaches zero. But in the Metropolis algorithm, where drift diffusion is used just as a transition proposition rule, this bias is corrected automatically by the rejection mechanism. In our application, the desired probability distribution function being the square absolute of the wave function, $f = |\psi|^2$, the drift velocity becomes:

$$F = 2\frac{1}{\psi}\nabla\psi \tag{14.27}$$

As expected, the walker is "pushed" along the gradient of the wave function.

When dealing with many-particle systems, we should also consider whether to move only one particle at a time at each transition or all at once. The former method may often be more efficient. A movement of only one particle will restrict the accessible space a walker can move to in a single transition even more, thus introducing correlation. But on the other hand, the acceptance is increased so that each particle can be moved further than it could in a standard all-particle move. It is also computationally far more efficient to do one-particle transitions particularly when dealing with complicated distributions governing many-dimensional anti-symmetrical fermionic systems.

Alternatively, we can treat the sequence of all one-particle transitions as one total transition of all particles. This gives a larger effective step length thus reducing the correlation. From a computational point of view, we may not gain any speed by summing up the individual one-particle transitions as opposed to doing an all-particle transition. But the reduced correlation increases the total efficiency. We are able to do fewer calculations in order to reach the same numerical accuracy.

Another way to acquire some control over the correlation is to do a so called blocking procedure on our set of numerical measurements. This is discussed in chapter 16.

14.6 Exercises

14.1. The aim of this problem 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

$$\psi_T(\mathbf{R}) = \phi(\mathbf{r}_1)\phi(\mathbf{r}_2)f(r_{12}). \tag{14.28}$$

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

$$\psi_{T1}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}}) = \exp\left(-\alpha(r_1 + r_2)\right), \tag{14.29}$$

where α is the variational parameter,

$$\psi_{T2}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}}) = \exp\left(-\alpha(r_1+r_2)\right)(1+\beta r_{12}), \tag{14.30}$$

with β as a new variational parameter and

$$\psi_{T3}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}}) = \exp\left(-\alpha(r_1+r_2)\right)\exp\left(\frac{r_{12}}{2(1+\beta r_{12})}\right). \tag{14.31}$$

a) Find the closed-form 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

$$\langle\widehat{\mathbf{H}}\rangle = \frac{\int d\mathbf{R}\Psi^*_T(\mathbf{R})\widehat{\mathbf{H}}(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi^*_T(\mathbf{R})\Psi_T(\mathbf{R})}, \tag{14.32}$$

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 closed-form expression

$$\langle\widehat{\mathbf{H}}\rangle = \frac{\hbar^2}{m_e}\alpha^2 - \frac{27}{32}\frac{e^2}{\pi\varepsilon_0}\alpha. \tag{14.33}$$

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.

14.2. 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

$$\left\{-\frac{\hbar^2\nabla^2_r}{2m_e} - \frac{ke^2}{|\mathbf{r}-\mathbf{R}/2|} - \frac{ke^2}{|\mathbf{r}+\mathbf{R}/2|} + \frac{ke^2}{R}\right\}\psi(\mathbf{r},\mathbf{R}) = E\psi(\mathbf{r},\mathbf{R}), \tag{14.34}$$

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.

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

$$\psi_{100}(r) = \left(\frac{1}{\pi a_0^3}\right)^{1/2} e^{-r/a_0}. \tag{14.35}$$

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

$$\psi_1(\mathbf{r},\mathbf{R}) = \left(\frac{1}{\pi a_0^3}\right)^{1/2} e^{-|\mathbf{r}-\mathbf{R}/2|/a_0}, \tag{14.36}$$

and

$$\psi_2(\mathbf{r},\mathbf{R}) = \left(\frac{1}{\pi a_0^3}\right)^{1/2} e^{-|\mathbf{r}+\mathbf{R}/2|/a_0}. \tag{14.37}$$

Based on these two wave functions, which represent where the electron can be, we attempt at the following linear combination

$$\psi_{\pm}(\mathbf{r},\mathbf{R}) = C_{\pm}\left(\psi_1(\mathbf{r},\mathbf{R}) \pm \psi_2(\mathbf{r},\mathbf{R})\right), \tag{14.38}$$

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 14.3.

14.3. 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

$$-\frac{\nabla_1^2}{2} - \frac{\nabla_2^2}{2}. \tag{14.39}$$

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

$$+\frac{1}{r_{12}}. \tag{14.40}$$

In a similar way we obtain a repulsive contribution from the interaction between the two protons given by

$$+\frac{1}{|\mathbf{R}|}, \tag{14.41}$$

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

$$\mathbf{r}_{1p1} = \mathbf{r}_1 + \mathbf{R}/2, \tag{14.42}$$

and

$$\mathbf{r}_{1p2} = \mathbf{r}_1 - \mathbf{R}/2, \tag{14.43}$$

from proton 2. Similarly, for electron 2 we obtain

$$\mathbf{r}_{2p1} = \mathbf{r}_2 + \mathbf{R}/2, \tag{14.44}$$

and

$$\mathbf{r}_{2p2} = \mathbf{r}_2 - \mathbf{R}/2. \tag{14.45}$$

These four distances define the attractive contributions to the potential energy

$$-\frac{1}{r_{1p1}} - \frac{1}{r_{1p2}} - \frac{1}{r_{2p1}} - \frac{1}{r_{2p2}}. \tag{14.46}$$

We can then write the total Hamiltonian as

$$\widehat{\mathbf{H}} = -\frac{\nabla_1^2}{2} - \frac{\nabla_2^2}{2} - \frac{1}{r_{1p1}} - \frac{1}{r_{1p2}} - \frac{1}{r_{2p1}} - \frac{1}{r_{2p2}} + \frac{1}{r_{12}} + \frac{1}{|\mathbf{R}|}, \tag{14.47}$$

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

$$\psi_T(\mathbf{r_1}, \mathbf{r_2}, \mathbf{R}) = \psi(\mathbf{r}_1, \mathbf{R})\psi(\mathbf{r}_2, \mathbf{R}) \exp\left(\frac{r_{12}}{2(1+\beta r_{12})}\right), \tag{14.48}$$

with the following trial wave function

$$\psi(\mathbf{r}_1, \mathbf{R}) = \left(\exp\left(-\alpha r_{1p1}\right) + \exp\left(-\alpha r_{1p2}\right)\right), \tag{14.49}$$

for electron 1 and

$$\psi(\mathbf{r}_2, \mathbf{R}) = \left(\exp\left(-\alpha r_{2p1}\right) + \exp\left(-\alpha r_{2p2}\right)\right). \tag{14.50}$$

The variational parameters are α and β.

One can show that in the limit where all distances approach zero that

$$\alpha = 1 + \exp\left(-R/\alpha\right), \tag{14.51}$$

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 4.

1. Find the local energy as function of R.
2. 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.
3. 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.
4. We modify now the approximation for the wave functions of electrons 1 and 2 by subtracting the two terms instead of adding up, viz

$$\psi(\mathbf{r}_1, \mathbf{R}) = \left(\exp\left(-\alpha r_{1p1}\right) - \exp\left(-\alpha r_{1p2}\right)\right), \tag{14.52}$$

for electron 1

$$\psi(\mathbf{r}_2, \mathbf{R}) = \left(\exp\left(-\alpha r_{2p1}\right) - \exp\left(-\alpha r_{2p2}\right)\right), \tag{14.53}$$

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.

Part V
Advanced topics

The last part of this book contains several project oriented advanced topics. Each of these topics can serve as a regular one-semester course based on the solution of the pertinent projects. We present several topics, mainly within applications to quantum mechanical systems and statistical mechanics. We discuss Hartree-Fock and density-functional theory calculations for electronic systems, variational and diffusion Monte Carlo calculations of many-particle systems (atoms, quantum dots and Bose-Einstein condensation), large molecular-dynamics calculations of solids, percolation and critical phenomena and advanced simulations of phase transitions.

Chapter 15
Many-body approaches to studies of electronic systems: Hartree-Fock theory and Density Functional Theory

Abstract This chapter presents the Hartree-Fock method with an emphasis on computing the energies of selected closed-shell atoms.

15.1 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 (on closed form) 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 [95–99], various types of Monte Carlo methods [100–102], perturbative many-body methods [103–105], Green's function methods [106, 107], the density-matrix renormalization group [108, 109], density functional theory [110] and ab initio density functional theory [111–113], and large-scale diagonalization methods [114–116], 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 three 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. We limit the attention to non-relativistic quantum mechanics.

In this chapter we limit ourselves to studies of electronic systems such atoms, molecules and quantum dots, as discussed partly in chapter 14 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 solution on a closed form and must resort to numerical efforts. In this chapter we will examine Hartree-Fock theory applied to the atomic problem. However, the machinery we expose can easily be extended to studies of molecules or two-dimensional systems like quantum dots.

For atoms and molecules, the electron-electron interaction is rather weak compared with the attraction from the nucleus. An independent particle picture is therefore 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 discussed in this chapter.

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. (14.15 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 14 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 14 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 the next 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.

In the next chapter we focus on the variational Monte Carlo method as a way to improve upon the Hartree-Fock results. 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. [117].

15.2 Hartree-Fock theory

Hartree-Fock theory [97, 118] 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[1]

$$\psi_{nlm_lsm_s} = \phi_{nlm_l}(\mathbf{r})\xi_{m_s}(s).$$

We use hereafter the shorthand $\psi_{nlm_lsm_s}(\mathbf{r}) = \psi_\alpha(\mathbf{r})$, where α now contains all the quantum numbers needed to specify a particular single-particle orbital.

The Slater determinant can then be written as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\ldots,\mathbf{r}_N,\alpha,\beta,\ldots,\nu) = \frac{1}{\sqrt{N!}}\left|\begin{array}{cccc} \psi_\alpha(\mathbf{r}_1) & \psi_\alpha(\mathbf{r}_2) & \ldots & \psi_\alpha(\mathbf{r}_N) \\ \psi_\beta(\mathbf{r}_1) & \psi_\beta(\mathbf{r}_2) & \ldots & \psi_\beta(\mathbf{r}_N) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_\nu(\mathbf{r}_1) & \psi_\beta(\mathbf{r}_2) & \ldots & \psi_\beta(\mathbf{r}_N) \end{array}\right|. \tag{15.1}$$

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. 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

[1] We limit ourselves to a restricted Hartree-Fock approach and assume that all the lowest-lying orbits are filled. This constitutes an approach suitable for systems with filled shells. The theory we outline is therefore applicable to systems which exhibit so-called magic numbers like the noble gases, closed-shell nuclei like ^{16}O and ^{40}Ca and quantum dots with magic number fillings.

of $2(2l+1)2(2\cdot 1+1) = 6$. This leads to ten possible values for $\alpha,\beta,\dots,\nu$. Fig. 15.1 shows the possible quantum numbers which the ten first elements can have.

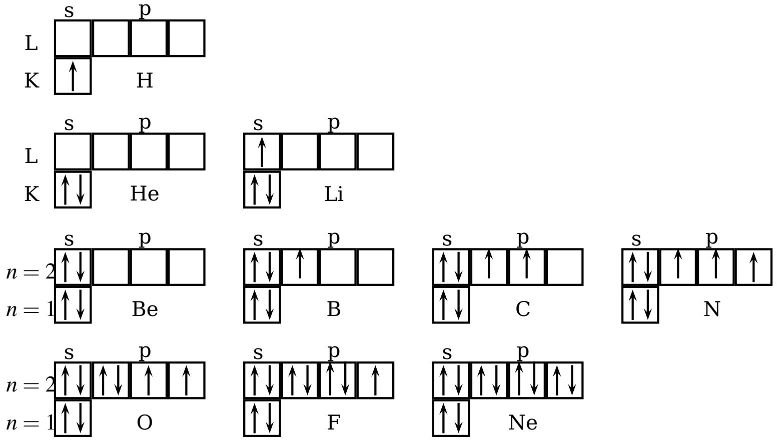


Fig. 15.1 The electronic configurations for the ten first elements. We let an arrow which points upward to represent a state with $m_s = 1/2$ while an arrow which points downwards has $m_s = -1/2$.

If we consider the helium atom with two electrons in the $1s$ state, we can write the total Slater determinant as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\alpha,\beta) = \frac{1}{\sqrt{2}}\begin{vmatrix} \psi_\alpha(\mathbf{r}_1) & \psi_\alpha(\mathbf{r}_2) \\ \psi_\beta(\mathbf{r}_1) & \psi_\beta(\mathbf{r}_2) \end{vmatrix}, \tag{15.2}$$

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 $\beta = nlm_lsm_s = (1001/2\downarrow)$. It is normal to skip the quantum number of the one-electron spin. We introduce therefore the shorthand $nlm_l\uparrow$ or $nlm_l\downarrow)$ for a particular state. Writing out the Slater determinant

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\alpha,\beta) = \frac{1}{\sqrt{2}}\left[\psi_\alpha(\mathbf{r}_1)\psi_\beta(\mathbf{r}_2) - \psi_\beta(\mathbf{r}_1)\psi_\gamma(\mathbf{r}_2)\right], \tag{15.3}$$

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

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\mathbf{r}_3,\alpha,\beta,\gamma) = \frac{1}{\sqrt{3!}}\begin{vmatrix} \psi_\alpha(\mathbf{r}_1) & \psi_\alpha(\mathbf{r}_2) & \psi_\alpha(\mathbf{r}_3) \\ \psi_\beta(\mathbf{r}_1) & \psi_\beta(\mathbf{r}_2) & \psi_\beta(\mathbf{r}_3) \\ \psi_\gamma(\mathbf{r}_1) & \psi_\gamma(\mathbf{r}_2) & \psi_\gamma(\mathbf{r}_3) \end{vmatrix}. \tag{15.4}$$

Computing the determinant gives

$$\begin{aligned}\Phi(\mathbf{r}_1,\mathbf{r}_2,\mathbf{r}_3,\alpha,\beta,\gamma) = \tfrac{1}{\sqrt{3!}}\big[&\psi_\alpha(\mathbf{r}_1)\psi_\beta(\mathbf{r}_2)\psi_\gamma(\mathbf{r}_3) + \psi_\beta(\mathbf{r}_1)\psi_\gamma(\mathbf{r}_2)\psi_\alpha(\mathbf{r}_3) + \psi_\gamma(\mathbf{r}_1)\psi_\alpha(\mathbf{r}_2)\psi_\beta(\mathbf{r}_3) - \\ &\psi_\gamma(\mathbf{r}_1)\psi_\beta(\mathbf{r}_2)\psi_\alpha(\mathbf{r}_3) - \psi_\beta(\mathbf{r}_1)\psi_\alpha(\mathbf{r}_2)\psi_\gamma(\mathbf{r}_3) - \psi_\alpha(\mathbf{r}_1)\psi_\gamma(\mathbf{r}_2)\psi_\beta(\mathbf{r}_3)\big].\end{aligned} \tag{15.5}$$

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 $\alpha,\beta,\dots,\nu$)

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\dots,\mathbf{r}_i,\dots,\mathbf{r}_j,\dots\mathbf{r}_N) = -\Phi(\mathbf{r}_1,\mathbf{r}_2,\dots,\mathbf{r}_j,\dots,\mathbf{r}_i,\dots\mathbf{r}_N).$$

As another example, consider the Slater determinant for the ground state of beryllium. This system is made up of four electrons and we assume that these electrons fill the $1s$ and $2s$ hydrogen-like orbits. The radial part of the single-particle could also be represented by other single-particle wave functions such as those given by the harmonic oscillator.

The ansatz for the Slater determinant can then be written as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) = \frac{1}{\sqrt{4!}}\begin{vmatrix} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) & \psi_{100\uparrow}(\mathbf{r}_3) & \psi_{100\uparrow}(\mathbf{r}_4) \\ \psi_{100\downarrow}(\mathbf{r}_1) & \psi_{100\downarrow}(\mathbf{r}_2) & \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) & \psi_{200\uparrow}(\mathbf{r}_3) & \psi_{200\uparrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_1) & \psi_{200\downarrow}(\mathbf{r}_2) & \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{vmatrix}.$$

We choose an ordering where columns represent the spatial positions of various electrons while rows refer to specific quantum numbers.

Note that the Slater determinant as written is zero since the spatial wave functions for the spin up and spin down states are equal. However, we can rewrite it as the product of two Slater determinants, one for spin up and one for spin down. In general we can rewrite it as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) = Det\uparrow(1,2)Det\downarrow(3,4) - Det\uparrow(1,3)Det\downarrow(2,4)$$

$$-Det\uparrow(1,4)Det\downarrow(3,2) + Det\uparrow(2,3)Det\downarrow(1,4) - Det\uparrow(2,4)Det\downarrow(1,3)$$

$$+Det\uparrow(3,4)Det\downarrow(1,2),$$

where we have defined

$$Det\uparrow(1,2) = \left|\frac{1}{\sqrt{2}}\begin{matrix} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) \end{matrix}\right|,$$

and

$$Det\downarrow(3,4) = \left|\frac{1}{\sqrt{2}}\begin{matrix} \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{matrix}\right|.$$

The total determinant is still zero! In our variational Monte Carlo calculations this will obviously cause problems.

We want to avoid to sum over spin variables, in particular when the interaction does not depend on spin. It can be shown, see for example Moskowitz *et al* [119, 120], that for the variational energy we can approximate the Slater determinant as the product of a spin up and a spin down Slater determinant

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta) \propto Det\uparrow(1,2)Det\downarrow(3,4),$$

or more generally as

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\dots\mathbf{r}_N) \propto Det\uparrow Det\downarrow,$$

where we have the Slater determinant as the product of a spin up part involving the number of electrons with spin up only (two in beryllium and five in neon) and a spin down part involving the electrons with spin down.

This ansatz is not antisymmetric under the exchange of electrons with opposite spins but it can be shown that it gives the same expectation value for the energy as the full Slater determinant as long as the Hamiltonian is spin independent. It is left as an exercise to the reader to show this. However, before we can prove this need to set up the expectation value of a given two-particle Hamiltonian using a Slater determinant.

15.3 Expectation value of the Hamiltonian with a given Slater determinant

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

$$\hat{H} = \hat{H}_0 + \hat{H}_I = \sum_{i=1}^{N} \hat{h}_i + \sum_{i<j=1}^{N} \frac{1}{r_{ij}}, \tag{15.6}$$

where

$$\hat{h}_i = -\frac{1}{2}\nabla_i^2 - \frac{Z}{r_i}. \tag{15.7}$$

The first term of Eq. (15.6), 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

$$E_0 \le E[\Phi] = \int \Phi^* \hat{H} \Phi d\tau \tag{15.8}$$

where Φ is a trial function which we assume to be normalized

$$\int \Phi^* \Phi d\tau = 1, \tag{15.9}$$

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. (15.1) which can be rewritten as

$$\Psi(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N, \alpha, \beta, \dots, \nu) = \frac{1}{\sqrt{N!}} \sum_P (-)^P P \psi_\alpha(\mathbf{r}_1) \psi_\beta(\mathbf{r}_2) \dots \psi_\nu(\mathbf{r}_N) = \sqrt{N!} \mathscr{A} \Phi_H, \tag{15.10}$$

where we have introduced the anti-symmetrization operator $\mathscr{A}$ defined by the summation over all possible permutations of two eletrons. It is defined as

$$\mathscr{A} = \frac{1}{N!} \sum_P (-)^P P, \tag{15.11}$$

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)

$$\Phi_H(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N, \alpha, \beta, \dots, \nu) = \psi_\alpha(\mathbf{r}_1) \psi_\beta(\mathbf{r}_2) \dots \psi_\nu(\mathbf{r}_N). \tag{15.12}$$

Both $\hat{H}_0$ and $\hat{H}_I$ are invariant under electron permutations, and hence commute with $\mathscr{A}$

$$[H_0, \mathscr{A}] = [H_I, \mathscr{A}] = 0. \tag{15.13}$$

Furthermore, $\mathscr{A}$ satisfies

$$\mathscr{A}^2 = \mathscr{A}, \tag{15.14}$$

since every permutation of the Slater determinant reproduces it. The expectation value of $\hat{H}_0$

$$\int \Phi^* \hat{H}_0 \Phi d\tau = N! \int \Phi_H^* \mathscr{A} \hat{H}_0 \mathscr{A} \Phi_H d\tau$$

is readily reduced to

$$\int \Phi^*\hat{H}_0\Phi d\tau = N!\int \Phi_H^*\hat{H}_0\mathscr{A}\Phi_H d\tau,$$

where we have used eqs. (15.13) and (15.14). The next step is to replace the anti-symmetry operator by its definition eq. (15.10) and to replace $\hat{H}_0$ with the sum of one-body operators

$$\int \Phi^*\hat{H}_0\Phi d\tau = \sum_{i=1}^{N}\sum_{P}(-)^P\int \Phi_H^*\hat{h}_i P\Phi_H d\tau. \tag{15.15}$$

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

$$\int \Phi^*\hat{H}_0\Phi d\tau = \sum_{i=1}^{N}\int \Phi_H^*\hat{h}_i\Phi_H d\tau. \tag{15.16}$$

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

$$\int \Phi^*\hat{H}_0\Phi d\tau = \sum_{\mu=1}^{N}\int \psi_\mu^*(\mathbf{r}_i)\hat{h}_i\psi_\mu(\mathbf{r}_i)d\mathbf{r}_i. \tag{15.17}$$

The expectation value of the two-body Hamiltonian is obtained in a similar manner. We have

$$\int \Phi^*\hat{H}_I\Phi d\tau = N!\int \Phi_H^*\mathscr{A}\hat{H}_I\mathscr{A}\Phi_H d\tau, \tag{15.18}$$

which reduces to

$$\int \Phi^*\hat{H}_I\Phi d\tau = \sum_{i\le j=1}^{N}\sum_{P}(-)^P\int \Phi_H^*\frac{1}{r_{ij}}P\Phi_H d\tau, \tag{15.19}$$

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

$$\int \Phi^*\hat{H}_I\Phi d\tau = \sum_{i<j=1}^{N}\int \Phi_H^*\frac{1}{r_{ij}}(1-P_{ij})\Phi_H d\tau. \tag{15.20}$$

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

$$\int \Phi^*\hat{H}_I\Phi d\tau = \frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\left[\int \psi_\mu^*(\mathbf{r}_i)\psi_\nu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\mu(\mathbf{r}_i)\psi_\nu(\mathbf{r}_j)d\mathbf{x_i}d\mathbf{x_j} - \int \psi_\mu^*(\mathbf{r}_i)\psi_\nu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\nu(\mathbf{r}_i)\psi_\mu(\mathbf{r}_i)d\mathbf{x_i}d\mathbf{x_j}\right]. \tag{15.21}$$

The first term is the so-called direct term or Hartree term, while the second is due to the Pauli principle and is called the exchange term or the Fock term. The factor $1/2$ is introduced because we now run over all pairs twice.

Combining Eqs. (15.17) and (15.21) we obtain the functional

$$E[\Phi] = \sum_{\mu=1}^{N}\int \psi_\mu^*(\mathbf{r}_i)\hat{h}_i\psi_\mu(\mathbf{r}_i)d\mathbf{r}_i + \frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\left[\int \psi_\mu^*(\mathbf{r}_i)\psi_\nu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\mu(\mathbf{r}_i)\psi_\nu(\mathbf{r}_j)d\mathbf{r}_i d\mathbf{r}_j - \right. \tag{15.22}$$

$$\left. - \int \psi_\mu^*(\mathbf{r}_i)\psi_\nu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\nu(\mathbf{r}_i)\psi_\mu(\mathbf{r}_j)d\mathbf{r}_i d\mathbf{r}_j\right].$$

15.4 Derivation of the Hartree-Fock equations

Having obtained the functional $E[\Phi]$, we now proceed to the second step of the calculation. With the given functional, we can embark on at least two types of variational strategies:

- We can vary the Slater determinant by changing the spatial part of the single-particle wave functions themselves.
- We can expand the single-particle functions in a known basis and vary the coefficients, that is, the new single-particle wave function $|a\rangle$ is written as a linear expansion in terms of a fixed chosen orthogonal basis (for the example harmonic oscillator, or Laguerre polynomials etc)
$$\psi_a = \sum_{\lambda} C_{a\lambda} \psi_{\lambda}.$$
In this case we vary the coefficients $C_{a\lambda}$.

We will derive the pertinent Hartree-Fock equations and discuss the pros and cons of the two methods. Both cases lead to a new Slater determinant which is related to the previous one via a unitary transformation.

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 [53].

We have already met the variational principle in chapter 14. We give here a brief reminder on the calculus of variations.

15.4.1 Reminder on 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

$$E[\Phi] = \int_a^b f(\Phi(x), \frac{\partial \Phi}{\partial \mathbf{r}}, \mathbf{r}) d\mathbf{r},$$

where E is the quantity which is sought minimized or maximized. The problem is that although f is a function of the variables Φ, $\partial\Phi/\partial\mathbf{r}$ and $\mathbf{r}$, the exact dependence of Φ on $\mathbf{r}$ 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.

The condition for a stationary value is given by a partial differential equation, which we here write in terms of one variable x

$$\frac{\partial f}{\partial \Phi} - \frac{d}{dx}\frac{\partial f}{\partial \Phi_x} = 0,$$

This equation is better better known as Euler's equation and it can easily be generalized to more variables.

As an example 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

$$df+\lambda d\phi=(\frac{\partial f}{\partial z}+\lambda\frac{\partial \phi}{\partial x})dx+(\frac{\partial f}{\partial y}+\lambda\frac{\partial \phi}{\partial y})dy+(\frac{\partial f}{\partial z}+\lambda\frac{\partial \phi}{\partial z})dz=0,$$

where our multiplier is chosen so that

$$\frac{\partial f}{\partial z}+\lambda\frac{\partial \phi}{\partial z}=0.$$

However, since we took dx and dy to be arbitrary 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, there is no need to determine λ. 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

$$\hat{H}=-\frac{1}{2}\nabla^2+V(x,y,z).$$

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 (using $V(x,y,z)=V$) we obtain

$$\delta E=\delta\left\{\int dxdydz\left(\frac{1}{2}\nabla\psi^*\nabla\psi+V\psi^*\psi\right)\right\}=0.$$

The requirement that the wave functions should be orthogonal gives

$$\int dxdydz\psi^*\psi=\text{constant},$$

and multiplying it with a Lagrangian multiplier λ and taking the variational minimum we obtain the final variational equation

$$\delta\left\{\int dxdydz\left(\frac{1}{2}\nabla\psi^*\nabla\psi+V\psi^*\psi-\lambda\psi^*\psi\right)\right\}=0.$$

We introduce the function f

$$f=\frac{1}{2}\nabla\psi^*\nabla\psi+V\psi^*\psi-\lambda\psi^*\psi=\frac{1}{2}(\psi_x^*\psi_x+\psi_y^*\psi_y+\psi_z^*\psi_z)+V\psi^*\psi-\lambda\psi^*\psi.$$

In our notation here we have dropped the dependence on x,y,z and introduced the shorthand ψ_x, ψ_y and ψ_z for the various first 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. 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.

15.4.2 Varying the single-particle wave functions

If we generalize the Euler-Lagrange equations to more variables and introduce N^2 Lagrange multipliers which we denote by $\varepsilon_{\mu\nu}$, we can write the variational equation for the functional of Eq. (15.23) as

$$\delta E - \sum_{\mu=1}^{N}\sum_{\nu=1}^{N} \varepsilon_{\mu\nu}\delta \int \psi_\mu^* \psi_\nu = 0. \tag{15.23}$$

For the orthogonal wave functions ψ_μ this reduces to

$$\delta E - \sum_{\mu=1}^{N} \varepsilon_{\mu}\delta \int \psi_\mu^* \psi_\mu = 0. \tag{15.24}$$

Variation with respect to the single-particle wave functions ψ_μ yields then

$$\begin{aligned}\sum_{\mu=1}^{N} \int \delta\psi_\mu^* \hat{h}_i \psi_\mu d\mathbf{x_i} + \frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\left[\int \delta\psi_\mu^*\psi_\nu^* \frac{1}{r_{ij}}\psi_\mu\psi_\nu d(\mathbf{x_i}\mathbf{x_j}) - \int \delta\psi_\mu^*\psi_\nu^* \frac{1}{r_{ij}}\psi_\nu\psi_\mu d\mathbf{r}_i d\mathbf{r}_j\right] \\ + \sum_{\mu=1}^{N} \int \psi_\mu^* \hat{h}_i \delta\psi_\mu d\mathbf{r}_i + \frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\left[\int \psi_\mu^*\psi_\nu^* \frac{1}{r_{ij}}\delta\psi_\mu\psi_\nu d\mathbf{r}_i d\mathbf{r}_j - \int \psi_\mu^*\psi_\nu^* \frac{1}{r_{ij}}\psi_\nu\delta\psi_\mu d\mathbf{r}_i d\mathbf{r}_j\right] \\ - \sum_{\mu=1}^{N} E_\mu \int \delta\psi_\mu^*\psi_\mu d\mathbf{x_i} - \sum_{\mu=1}^{N} E_\mu \int \psi_\mu^*\delta\psi_\mu d\mathbf{r}_i = 0.\end{aligned} \tag{15.25}$$

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

$$\begin{aligned}\left[-\frac{1}{2}\nabla_i^2 - \frac{Z}{r_i} + \sum_{\nu=1}^{N}\int \psi_\nu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\nu(\mathbf{r}_j)d\mathbf{r}_j\right]\psi_\mu(\mathbf{x_i}) \\ - \left[\sum_{\nu=1}^{N}\int \psi_\nu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\mu(\mathbf{r}_j)d\mathbf{r}_j\right]\psi_\nu(\mathbf{r}_i) = \varepsilon_\mu\psi_\mu(\mathbf{r}_i).\end{aligned} \tag{15.26}$$

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 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 the 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 the following operators

$$V_\mu^d(\mathbf{r}_i) = \int \psi_\mu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\mu(\mathbf{r}_j)d\mathbf{r}_j \tag{15.27}$$

and

$$V_\mu^{ex}(\mathbf{r}_i)g(\mathbf{r}_i) = \left(\int \psi_\mu^*(\mathbf{r}_j)\frac{1}{r_{ij}}g(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_\mu(\mathbf{r}_i), \tag{15.28}$$

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

$$V_\mu^{ex}(\mathbf{r}_i)\psi_\nu(\mathbf{r}_i) = \left(\int \psi_\mu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\nu(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_\mu(\mathbf{r}_i). \tag{15.29}$$

We may then rewrite the Hartree-Fock equations as

$$H_i^{HF}\psi_\nu(\mathbf{r}_i) = \varepsilon_\nu\psi_\nu(\mathbf{r}_i), \tag{15.30}$$

with

$$H_i^{HF} = h_i + \sum_{\mu=1}^{N} V_\mu^d(\mathbf{r}_i) - \sum_{\mu=1}^{N} V_\mu^{ex}(\mathbf{r}_i), \tag{15.31}$$

and where h_i is defined by equation (15.7).

15.4.3 Detailed solution of the Hartree-Fock equations

We show here the explicit form of the Hartree-Fock for helium and beryllium

Let us introduce

$$\psi_{nlm_lsm_s} = \phi_{nlm_l}(\mathbf{r})\xi_{m_s}(s)$$

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 and $u_{nl} = rR_{nl}$. We have for helium

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,\alpha,\beta) = \frac{1}{\sqrt{2}}\phi_{100}(\mathbf{r}_1)\phi_{100}(\mathbf{r}_2)\left[\xi_\uparrow(1)\xi_\downarrow(2) - \xi_\uparrow(2)\xi_\downarrow(1)\right],$$

The direct term acts on

$$\frac{1}{\sqrt{2}}\phi_{100}(\mathbf{r}_1)\phi_{100}(\mathbf{r}_2)\xi_\uparrow(1)\xi_\downarrow(2)$$

while the exchange term acts on

$$-\frac{1}{\sqrt{2}}\phi_{100}(\mathbf{r}_1)\phi_{100}(\mathbf{r}_2)\xi_\uparrow(2)\xi_\downarrow(1).$$

How do these terms get translated into the Hartree and the Fock terms?

The Hartree term

$$V_\mu^d(\mathbf{r}_i) = \int \psi_\mu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\mu(\mathbf{r}_j)d\mathbf{r}_j,$$

acts on $\psi_\lambda(\mathbf{r}_i) = \phi_{nlm_l}(\mathbf{r}_i)\xi_{m_s}(s_i)$, that is it results in

$$V_\mu^d(\mathbf{r}_i)\psi_\lambda(\mathbf{r}_i) = \left(\int \psi_\mu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\mu(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_\lambda(\mathbf{r}_i),$$

and accounting for spins we have

$$V_{nlm_l\uparrow}^d(\mathbf{r}_i)\psi_\lambda(\mathbf{r}_i) = \left(\int \psi_{nlm_l\uparrow}^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_{nlm_l\uparrow}(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_\lambda(\mathbf{r}_i),$$

and

$$V_{nlm_l\downarrow}^d(\mathbf{r}_i)\psi_\lambda(\mathbf{r}_i) = \left(\int \psi_{nlm_l\downarrow}^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_{nlm_l\downarrow}(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_\lambda(\mathbf{r}_i),$$

If the state we act on has spin up, we obtain two terms from the Hartree part,

$$\sum_{\mu=1}^{N} V_\mu^d(\mathbf{r}_i),$$

and since the interaction does not depend on spin we end up with a total contribution for helium

$$\sum_{\mu=1}^{N} V_\mu^d(\mathbf{r}_i)\psi_\lambda(\mathbf{r}_i) = \left(2\int \phi_{100}^*(\mathbf{r}_j)\frac{1}{r_{ij}}\phi_{100}(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_\lambda(\mathbf{r}_i),$$

one from spin up and one from spin down. Since the energy for spin up or spin down is the same we can then write the action of the Hartree term as

$$\sum_{\mu=1}^{N} V_\mu^d(\mathbf{r}_i)\psi_\lambda(\mathbf{r}_i) = \left(2\int \phi_{100}^*(\mathbf{r}_j)\frac{1}{r_{ij}}\phi_{100}(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_{100\uparrow}(\mathbf{r}_i).$$

(the spin in $\psi_{100\uparrow}$ is irrelevant)

What we need to code for helium is then

$$\Phi(r_i)u_{10} = 2V_{10}^d(r_i)u_{10}(r_i) = 2\int_0^\infty |u_{10}(r_j)|^2\frac{1}{r_>}dr_j)u_{10}(r_i).$$

with $r_> = max(r_i, r_j)$. What about the exchange or Fock term

$$V_\mu^{ex}(\mathbf{r}_i)\psi_\lambda(\mathbf{r}_i) = \left(\int \psi_\mu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\lambda(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_\mu(\mathbf{r}_i)?$$

We must be careful here with

$$V_\mu^{ex}(\mathbf{r}_i)\psi_\lambda(\mathbf{r}_i) = \left(\int \psi_\mu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\lambda(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_\mu(\mathbf{r}_i),$$

because the spins of μ and λ have to be the same due to the constraint

$$\langle s_\mu m_s^\mu | s_\lambda m_s^\lambda\rangle = \delta_{m_s^\mu, m_s^\lambda}.$$

This means that if $m_s^\mu = \uparrow$ then $m_s^\lambda = \uparrow$ and if $m_s^\mu = \downarrow$ then $m_s^\lambda = \downarrow$. That is

$$V_\mu^{ex}(\mathbf{r}_i)\psi_\lambda(\mathbf{r}_i) = \delta_{m_s^\mu, m_s^\lambda}\left(\int \psi_\mu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\lambda(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_\mu(\mathbf{r}_i),$$

The consequence is that for the $1s\uparrow$ (and the same for $1s\downarrow$) state we get only one contribution from the Fock term, namely

$$\sum_{\mu=1}^{N} V_\mu^{ex}(\mathbf{r}_i)\psi_{100\uparrow}(\mathbf{r}_i) = \delta_{m_s^\mu,\uparrow}\left(\int \psi_\mu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_{100\uparrow}(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_\mu(\mathbf{r}_i),$$

resulting in

$$\sum_{\mu=1}^{N} V_\mu^{ex}(\mathbf{r}_i)\psi_{100\uparrow}(\mathbf{r}_i) = \left(\int \psi_{100\uparrow}^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_{100\uparrow}(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_{100\uparrow}(\mathbf{r}_i).$$

The final Fock term for helium is then

$$\sum_{\mu=1}^{N} V_\mu^{ex}(\mathbf{r}_i)\psi_{100\uparrow}(\mathbf{r}_i) = \left(\int \psi_{100\uparrow}^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_{100\uparrow}(\mathbf{r}_j)d\mathbf{r}_j\right)\psi_{100\uparrow}(\mathbf{r}_i),$$

which is exactly the same as the Hartree term except for a factor of 2. Else the integral is the same. We can then write the differential equation

$$\left(-\frac{1}{2}\frac{d^2}{dr^2}+\frac{l(l+1)}{2r^2}-\frac{2}{r}+\Phi_{nl}(r)-F_{nl}(r)\right)u_{nl}(r)=e_{nl}u_{nl}(r).$$

as

$$\left(-\frac{1}{2}\frac{d^2}{dr^2}+\frac{l(l+1)}{2r^2}-\frac{2}{r}+2V_{10}^{d}(r)\right)u_{10}(r)-V_{10}^{ex}(r)=e_{10}u_{10}(r),$$

or

$$\left(-\frac{1}{2}\frac{d^2}{dr^2}-\frac{2}{r}+V_{10}^{d}(r)\right)u_{10}(r)=e_{10}u_{10}(r),$$

since $l=0$. The shorthand $V_{10}^{ex}(r)$ contains the $1s$ wave function.

The expression we have obtained are independent of the spin projections and we have skipped them in the equations.

For beryllium the Slater determinant takes the form

$$\Phi(\mathbf{r}_1,\mathbf{r}_2,,\mathbf{r}_3,\mathbf{r}_4,\alpha,\beta,\gamma,\delta)=\frac{1}{\sqrt{4!}}\begin{vmatrix}\psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) & \psi_{100\uparrow}(\mathbf{r}_3) & \psi_{100\uparrow}(\mathbf{r}_4)\\ \psi_{100\downarrow}(\mathbf{r}_1) & \psi_{100\downarrow}(\mathbf{r}_2) & \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4)\\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) & \psi_{200\uparrow}(\mathbf{r}_3) & \psi_{200\uparrow}(\mathbf{r}_4)\\ \psi_{200\downarrow}(\mathbf{r}_1) & \psi_{200\downarrow}(\mathbf{r}_2) & \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4)\end{vmatrix},$$

When we now spell out the Hartree-Fock equations we get two coupled differential equations, one for u_{10} and one for u_{20}.

The $1s$ wave function has the same Hartree-Fock contribution as in helium for the $1s$ state, but the $2s$ state gives two times the Hartree term and one time the Fock term. We get

$$\sum_{\mu=1}^{N}V_{\mu}^{d}(\mathbf{r}_i)\psi_{100\uparrow}(\mathbf{r}_i)=2\int_0^{\infty}d\mathbf{r}_j\left(\phi_{100}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}\phi_{100}(\mathbf{r}_j)+\phi_{200}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}\phi_{200}(\mathbf{r}_j)\right)\psi_{100\uparrow}(\mathbf{r}_i)$$

$$=(2V_{10}^{d}(\mathbf{r}_i)+2V_{20}^{d}(\mathbf{r}_i))\psi_{100\uparrow}(\mathbf{r}_i)$$

for the Hartree part.

For the Fock term we get (we fix the spin)

$$\sum_{\mu=1}^{N}V_{\mu}^{ex}(\mathbf{r}_i)\psi_{100\uparrow}(\mathbf{r}_i)=\int_0^{\infty}d\mathbf{r}_j\phi_{100}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}\phi_{100}(\mathbf{r}_j)\psi_{100\uparrow}(\mathbf{r}_i)+$$

$$\int_0^{\infty}d\mathbf{r}_j\phi_{200}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}\phi_{100}(\mathbf{r}_j)\psi_{200\uparrow}(\mathbf{r}_i)=V_{10}^{ex}(\mathbf{r}_i)+V_{20}^{ex}(\mathbf{r}_i).$$

The first term is the same as we have for the Hartree term with $1s$ except the factor of two. The final differential equation is

$$\left(-\frac{1}{2}\frac{d^2}{dr^2}-\frac{4}{r}+V_{10}^{d}(r)+2V_{20}^{d}(r)\right)u_{10}(r)-V_{20}^{ex}(r)=e_{10}u_{10}(r).$$

Note again that the $V_{20}^{ex}(r)$ contains the $1s$ function in the integral, that is

$$V_{20}^{ex}(r)=\int_0^{\infty}d\mathbf{r}_j\phi_{200}^{*}(\mathbf{r}_j)\frac{1}{r-r_j}\phi_{100}(\mathbf{r}_j)\psi_{200\uparrow}(\mathbf{r}).$$

The $2s$ wave function obtains the following Hartree term (recall that the interaction has no spin dependence)

$$\sum_{\mu=1}^{N} V_{\mu}^{d}(\mathbf{r}_i)\psi_{200\uparrow}(\mathbf{r}_i) = 2\int_0^{\infty} d\mathbf{r}_j \left(\phi_{100}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}\phi_{100}(\mathbf{r}_j) + \phi_{200}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}\phi_{200}(\mathbf{r}_j)\right)\psi_{200\uparrow}(\mathbf{r}_i) =$$

$$(2V_{10}^{d}(\mathbf{r}_i) + 2V_{20}^{d}(\mathbf{r}_i))\psi_{200\uparrow}(\mathbf{r}_i)$$

For the Fock term we get

$$\sum_{\mu=1}^{N} V_{\mu}^{ex}(\mathbf{r}_i)\psi_{200\uparrow}(\mathbf{r}_i) = \int_0^{\infty} d\mathbf{r}_j \phi_{100}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}\phi_{200}(\mathbf{r}_j)\psi_{100\uparrow}(\mathbf{r}_i) +$$

$$\int_0^{\infty} d\mathbf{r}_j \phi_{200}^{*}(\mathbf{r}_j)\frac{1}{r_{ij}}\phi_{200}(\mathbf{r}_j)\psi_{200\uparrow}(\mathbf{r}_i) = V_{10}^{ex}(\mathbf{r}_i) + V_{20}^{ex}(\mathbf{r}_i)$$

The second term is the same as we have for the Hartree term with $2s$. The final differential equation is

$$\left(-\frac{1}{2}\frac{d^2}{dr^2} - \frac{4}{r} + 2V_{10}^{d}(r) + V_{20}^{d}(r)\right)u_{20}(r) - V_{10}^{ex}(r) = e_{20}u_{20}(r).$$

Note again that $V_{10}^{ex}(r)$ contains the $2s$ function in the integral, that is

$$V_{10}^{ex}(r) = \int_0^{\infty} d\mathbf{r}_j \phi_{100}^{*}(\mathbf{r}_j)\frac{1}{r - r_j}\phi_{200}(\mathbf{r}_j)\psi_{100\uparrow}(\mathbf{r}).$$

We have two coupled differential equations

$$\left(-\frac{1}{2}\frac{d^2}{dr^2} - \frac{4}{r} + V_{10}^{d}(r) + 2V_{20}^{d}(r)\right)u_{10}(r) - V_{20}^{ex}(r) = e_{10}u_{10}(r),$$

and

$$\left(-\frac{1}{2}\frac{d^2}{dr^2} - \frac{4}{r} + 2V_{10}^{d}(r) + V_{20}^{d}(r)\right)u_{20}(r) - V_{10}^{ex}(r) = e_{20}u_{20}(r).$$

Recall again that the interaction does not depend on spin. This means that the single-particle energies and single-particle function u do not depend on spin.

15.4.4 Hartree-Fock by variation of basis function coefficients

Another possibility is to expand the single-particle functions in a known basis and vary the coefficients, that is, the new single-particle wave function is written as a linear expansion in terms of a fixed chosen orthogonal basis (for example harmonic oscillator, Laguerre polynomials etc)

$$\psi_a = \sum_{\lambda} C_{a\lambda}\psi_{\lambda}. \tag{15.32}$$

In this case we vary the coefficients $C_{a\lambda}$.

The single-particle wave functions $\psi_{\lambda}(\mathbf{r})$, defined by the quantum numbers λ and $\mathbf{r}$ are defined as the overlap

$$\psi_{\alpha}(\mathbf{r}) = \langle \mathbf{r}|\alpha\rangle.$$

We will omit the radial dependence of the wave functions and introduce first the following shorthands for the Hartree and Fock integrals

$$\langle \mu\nu|V|\mu\nu\rangle = \int \psi_{\mu}^{*}(\mathbf{r}_i)\psi_{\nu}^{*}(\mathbf{r}_j)V(r_{ij})\psi_{\mu}(\mathbf{r}_i)\psi_{\nu}(\mathbf{r}_j)d\mathbf{r}_i\mathbf{r}_j,$$

and

$$\langle\mu\nu|V|\nu\mu\rangle = \int \psi_\mu^*(\mathbf{r}_i)\psi_\nu^*(\mathbf{r}_j)V(r_{ij})\psi_\nu(\mathbf{r}_i)\psi_\mu(\mathbf{r}_i)d\mathbf{r}_i\mathbf{r}_j.$$

Since the interaction is invariant under the interchange of two particles it means for example that we have

$$\langle\mu\nu|V|\mu\nu\rangle = \langle\nu\mu|V|\nu\mu\rangle,$$

or in the more general case

$$\langle\mu\nu|V|\sigma\tau\rangle = \langle\nu\mu|V|\tau\sigma\rangle.$$

The direct and exchange matrix elements can be brought together if we define the antisymmetrized matrix element

$$\langle\mu\nu|V|\mu\nu\rangle_{AS} = \langle\mu\nu|V|\mu\nu\rangle - \langle\mu\nu|V|\nu\mu\rangle,$$

or for a general matrix element

$$\langle\mu\nu|V|\sigma\tau\rangle_{AS} = \langle\mu\nu|V|\sigma\tau\rangle - \langle\mu\nu|V|\tau\sigma\rangle.$$

It has the symmetry property

$$\langle\mu\nu|V|\sigma\tau\rangle_{AS} = -\langle\mu\nu|V|\tau\sigma\rangle_{AS} = -\langle\nu\mu|V|\sigma\tau\rangle_{AS}.$$

The antisymmetric matrix element is also hermitian, implying

$$\langle\mu\nu|V|\sigma\tau\rangle_{AS} = \langle\sigma\tau|V|\mu\nu\rangle_{AS}.$$

With these notations we rewrite Eq. (15.21) as

$$\int \Phi^*\hat{H}_0\Phi d\tau = \frac{1}{2}\sum_{\mu=1}^{A}\sum_{\nu=1}^{A}\langle\mu\nu|V|\mu\nu\rangle_{AS}. \tag{15.33}$$

Combining Eqs. (15.17) and (15.33) we obtain the energy functional

$$E[\Phi] = \sum_{\mu=1}^{N}\langle\mu|h|\mu\rangle + \frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\langle\mu\nu|V|\mu\nu\rangle_{AS}. \tag{15.34}$$

which we will use as our starting point for the Hartree-Fock calculations.

If we vary the above energy functional with respect to the basis functions $|\mu\rangle$, this corresponds to what was done in the previous subsection. We are however interested in defining a new basis defined in terms of a chosen basis as defined in Eq. (15.32). We can then rewrite the energy functional as

$$E[\Psi] = \sum_{a=1}^{N}\langle a|h|a\rangle + \frac{1}{2}\sum_{ab}^{N}\langle ab|V|ab\rangle_{AS}, \tag{15.35}$$

where Ψ is the new Slater determinant defined by the new basis of Eq. (15.32). Using Eq. (15.32) we can rewrite Eq. (15.35) as

$$E[\Psi] = \sum_{a=1}^{N}\sum_{\alpha\beta}C_{a\alpha}^*C_{a\beta}\langle\alpha|h|\beta\rangle + \frac{1}{2}\sum_{ab}\sum_{\alpha\beta\gamma\delta}C_{a\alpha}^*C_{b\beta}^*C_{a\gamma}C_{b\delta}\langle\alpha\beta|V|\gamma\delta\rangle_{AS}. \tag{15.36}$$

We wish now to minimize the above functional. We introduce again a set of Lagrange multipliers, noting that since $\langle a|b\rangle = \delta_{a,b}$ and $\langle\alpha|\beta\rangle = \delta_{\alpha,\beta}$, the coefficients $C_{a\gamma}$ obey the relation

$$\langle a|b\rangle = \delta_{a,b} = \sum_{\alpha\beta}C_{a\alpha}^*C_{a\beta}\langle\alpha|\beta\rangle = \sum_{\alpha}C_{a\alpha}^*C_{a\alpha},$$

which allows us to define a functional to be minimized that reads

$$E[\Psi] - \sum_a \varepsilon_a \sum_\alpha C^*_{a\alpha} C_{a\alpha}. \tag{15.37}$$

Minimizing with respect to $C^*_{k\alpha}$, remembering that $C^*_{k\alpha}$ and $C_{k\alpha}$ are independent, we obtain

$$\frac{d}{dC^*_{k\alpha}} \left[E[\Psi] - \sum_a \varepsilon_a \sum_\alpha C^*_{a\alpha} C_{a\alpha} \right] = 0, \tag{15.38}$$

which yields for every single-particle state k the following Hartree-Fock equations

$$\sum_\gamma C_{k\gamma} \langle \alpha | h | \gamma \rangle + \frac{1}{2} \sum_a \sum_{\beta\gamma\delta} C^*_{a\beta} C_{a\delta} C_{k\gamma} \langle \alpha\beta | V | \gamma\delta \rangle_{AS} = \varepsilon_k C_{k\alpha}. \tag{15.39}$$

We can rewrite this equation as

$$\sum_{\gamma=1}^{N} \left\{ \langle \alpha | h | \gamma \rangle + \frac{1}{2} \sum_a^N \sum_{\beta\delta}^N C^*_{a\beta} C_{a\delta} \langle \alpha\beta | V | \gamma\delta \rangle_{AS} \right\} C_{k\gamma} = \varepsilon_k C_{k\alpha}. \tag{15.40}$$

Defining

$$h^{HF}_{\alpha\gamma} = \langle \alpha | h | \gamma \rangle + \frac{1}{2} \sum_a^N \sum_{\beta\delta}^N C^*_{a\beta} C_{a\delta} \langle \alpha\beta | V | \gamma\delta \rangle_{AS},$$

we can rewrite the new equations as

$$\sum_{\gamma=1}^{N} h^{HF}_{\alpha\gamma} C_{k\gamma} = \varepsilon_k C_{k\alpha}. \tag{15.41}$$

The advantage of this approach is that we can calculate and tabulate the matrix elements $\alpha|h|\gamma\rangle$ and $\langle \alpha\beta | V | \gamma\delta \rangle_{AS}$ once and for all. If the basis $|\alpha\rangle$ is chosen properly, then the matrix elements can also serve as a good starting point for a Hartree-Fock calculation. Eq. (15.41) is nothing but an eigenvalue problem. The eigenvectors are defined by the coefficients $C_{k\gamma}$.

The size of the matrices to diagonalize are seldomly larger than 1000×1000 and can be solved by the standard eigenvalue methods that we discussed in chapter 7.

For closed shell atoms it is natural to consider the spin-orbitals as paired. For example, two $1s$ orbitals with different spin have the same spatial wave-function, but orthogonal spin functions. For open-shell atoms two procedures are commonly used; the *restricted Hartree-Fock* (RHF) and *unrestricted Hartree-Fock* (UHF). In RHF all the electrons except those occupying open-shell orbitals are forced to occupy doubly occupied spatial orbitals, while in UHF all orbitals are treated independently. The UHF, of course, yields a lower variational energy than the RHF formalism. One disadvantage of the UHF over the RHF, is that whereas the RHF wave function is an eigenfunction of S^2, the UHF function is not; that is, the total spin angular momentum is not a well-defined quantity for a UHL wave-function. Here we limit our attention to closed shell RHF's, and show how the coupled HF equations may be turned into a matrix problem by expressing the spin-orbitals using known sets of basis functions.

In principle, a complete set of basis functions must be used to represent spin-orbitals exactly, but this is not computationally feasible. A given finite set of basis functions is, due to the incompleteness of the basis set, associated with a *basis-set truncation error*. The limiting HF energy, with truncation error equal to zero, will be referred to as the *Hartree-Fock limit*.

The computational time depends on the number of basis-functions and of the difficulty in computing the integrals of both the Fock matrix and the overlap matrix. Therefore we wish to keep the number of basis functions as low as possible and choose the basis-functions cleverly.

By cleverly we mean that the truncation error should be kept as low as possible, and that the computation of the matrix elements of both the overlap and the Fock matrices should not be too time consuming.

One choice of basis functions are the so-called *Slater type orbitals* (STO), see for example Ref. [121]). They are defined as

$$\Psi_{nlm_l}(r,\theta,\phi) = \mathcal{N} r^{n_{eff}-1} e^{\frac{Z_{eff}\rho}{n_{eff}}} Y_{lm_l}(\theta,\phi). \tag{15.42}$$

Here $\mathcal{N}$ is a normalization constant that for the purpose of basis set expansion may be put into the unknown $c_{i\mu}$'s, Y_{lm_l} is a spherical harmonic and $\rho = r/a_0$.

The normalization constant of the spherical harmonics may of course also be put into the expansion coefficients $c_{i\mu}$. The effective principal quantum number n_{eff} is related to the true principal quantum number N by the following mapping (ref. [121])

$$n \to n_{eff} : 1 \to 1 \quad 2 \to 2 \quad 3 \to 3 \quad 4 \to 3.7 \quad 5 \to 4.0 \quad 6 \to 4.2.$$

The effective atomic number Z_{eff} for the ground state orbitals of some neutral ground-state atoms are listed in table 15.1. The values in table 15.1 have been constructed by fitting STOs to numerically computed wave-functions [122].

Effective Atomic Number

	H							He
1s	1							1.6875
	Li	Be	B	C	N	O	F	Ne
1s	2.6906	3.6848	4.6795	5.6727	6.6651	7.6579	8.6501	9.6421
2s	1.2792	1.9120	2.5762	3.2166	3.8474	4.4916	5.1276	5.7584
2p			2.4214	3.1358	3.8340	4.4532	5.1000	5.7584
	Na	Mg	Al	Si	P	S	Cl	Ar
1s	10.6259	11.6089	12.5910	13.5754	14.5578	15.5409	16.5239	17.5075
2s	6.5714	7.3920	8.2136	9.0200	9.8250	10.6288	11.4304	12.2304
2p	6.8018	7.8258	8.9634	9.9450	10.9612	11.9770	12.9932	14.0082
3s	2.5074	3.3075	4.1172	4.9032	5.6418	6.3669	7.0683	7.7568
3p			4.0656	4.2852	4.8864	5.4819	6.1161	6.7641

Table 15.1 Values of Z_{eff} for neutral ground-state atoms [122].

15.5 Density Functional Theory

Hohenberg and Kohn [123] 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) [123, 124], see also Refs. [3, 110, 125–127]. DFT takes into account electron correlations but is less demanding computationally than for example full diagonalization oor many-body perturbation theory.

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* [123] 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

$$E[\rho] = -\frac{1}{2}\sum_{i=1}^{N}\int \Psi_i^*(\mathbf{r_1})\nabla_1^2\Psi_i(\mathbf{r_1})d\mathbf{r_1} - \int \frac{Z}{r_1}\rho(\mathbf{r_1})d\mathbf{r_1} + \frac{1}{2}\int \frac{\rho(\mathbf{r_1})\rho(\mathbf{r_2})}{r_{12}}d\mathbf{r_1}d\mathbf{r_2} + E_{EXC}[\rho]$$

with Ψ_i the *Kohn-Sham* (KS) *orbitals*. The ground-state charge density is given by

$$\rho(\mathbf{r}) = \sum_{i=1}^{N}|\Psi_i(\mathbf{r})|^2,$$

where the sum is over the occupied Kohn-Sham orbitals. The last term, $E_{EXC}[\rho]$, 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

$$\left\{-\frac{1}{2}\nabla_1^2 - \frac{Z}{r_1} + \int \frac{\rho(\mathbf{r_2})}{r_{12}}d\mathbf{r_2} + V_{EXC}(\mathbf{r_1})\right\}\Psi_i(\mathbf{r_1}) = \varepsilon_i\Psi_i(\mathbf{r_1}) \tag{15.43}$$

where ε_i are the KS orbital energies, and where the *exchange-correlation potential* is given by

$$V_{EXC}[\rho] = \frac{\delta E_{EXC}[\rho]}{\delta\rho}. \tag{15.44}$$

The KS equations are solved in a self-consistent fashion. A variety of basis set functions can be used, and the experience gained in Hartree-Fock calculations are often useful. The computational time needed for a Density function theory 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

$$E_{EXC} = \int \rho(\mathbf{r})\varepsilon_{EXC}[\rho(\mathbf{r})]d\mathbf{r}, \tag{15.45}$$

where $\varepsilon_{EXC}[\rho(\mathbf{r})]$ 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.

15.5.1 Hohenberg-Kohn Theorem

15.5.2 Derivation of the Kohn-Sham Equations

15.5.3 The Local Density Approximation and the Electron Gas

15.5.4 Applications and Code Examples

15.6 Exercises

15.1. The aim of this problem is to perform Hartree-Fock calculations in order to obtain an optimal basis for the single-particle wave functions Beryllium.

The Hartree-Fock functional is written as

$$E[\Phi] = \sum_{\mu=1}^{N} \int \psi_\mu^*(\mathbf{r}_i)\hat{h}_i\psi_\mu(\mathbf{r}_i)d\mathbf{r}_i + \frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\left[\int \psi_\mu^*(\mathbf{r}_i)\psi_\nu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\mu(\mathbf{r}_i)\psi_\nu(\mathbf{r}_j)d\mathbf{r}_i\mathbf{r}_j \right.$$

$$\left. - \int \psi_\mu^*(\mathbf{r}_i)\psi_\nu^*(\mathbf{r}_j)\frac{1}{r_{ij}}\psi_\nu(\mathbf{r}_i)\psi_\mu(\mathbf{r}_i)d\mathbf{r}_i\mathbf{r}_j\right].$$

The more compact version is

$$E[\Phi] = \sum_{\mu=1}^{N}\langle\mu|h|\mu\rangle + \frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\left[\langle\mu\nu|\frac{1}{r_{ij}}|\mu\nu\rangle - \langle\mu\nu|\frac{1}{r_{ij}}|\nu\mu\rangle\right].$$

With the given functional, we can perform at least two types of variational strategies.

- Vary the Slater determinant by changing the spatial part of the single-particle wave functions themselves.
- Expand the single-particle functions in a known basis and vary the coefficients, that is, the new function single-particle wave function $|a\rangle$ is written as a linear expansion in terms of a fixed basis ϕ (harmonic oscillator, Laguerre polynomials etc)

$$\psi_a = \sum_\lambda C_{a\lambda}\phi_\lambda,$$

Both cases lead to a new Slater determinant which is related to the previous via a unitary transformation. The second one is the one we will use in this project.

1. Consider a Slater determinant built up of single-particle orbitals ψ_λ, with $\lambda = 1, 2, \dots, N$.
The unitary transformation

$$\psi_a = \sum_\lambda C_{a\lambda}\phi_\lambda,$$

brings us into the new basis. Show that the new basis is orthonormal. Show that the new Slater determinant constructed from the new single-particle wave functions can be written as the determinant based on the previous basis and the determinant of the matrix C. Show that the old and the new Slater determinants are equal up to a complex constant with absolute value unity. (Hint, C is a unitary matrix).

2. Minimizing with respect to $C^*_{k\alpha}$, remembering that $C^*_{k\alpha}$ and $C_{k\alpha}$ are independent and defining

$$h^{HF}_{\alpha\gamma} = \langle\alpha|h|\gamma\rangle + \sum_{a=1}^{N}\sum_{\beta\delta} C^*_{a\beta}C_{a\delta}\langle\alpha\beta|V|\gamma\delta\rangle_{AS},$$

show that you can write the Hartree-Fock equations as

$$\sum_\gamma h^{HF}_{\alpha\gamma} C_{k\gamma} = \varepsilon_k C_{k\alpha}.$$

Explain the meaning of the different terms.
Set up the Hartree-Fock equations for the ground state beryllium with the electrons occupying the respective 'hydrogen-like' orbitals $1s$ and $2s$. There is no spin-orbit part in the two-body Hamiltonian.

3. As basis functions for our calculations we will use hydrogen-like single-particle functions. In the computations you will need to program the Coulomb interaction with matrix elements involving single-particle wave functions with $l=0$ only, so-called s-waves. We need only the radial part since the spherical harmonics for the s-waves are rather simple. Our radial wave functions are
$$R_{n0}(r) = \left(\frac{2Z}{n}\right)^{3/2}\sqrt{\frac{(n-1)!}{2n\times n!}} L^1_{n-1}(\frac{2Zr}{n})\exp{(-\frac{Zr}{n})},$$
with energies $-Z^2/2n^2$. A function for computing the generalized Laguerre polynomials $L^1_{n-1}(\frac{2Zr}{n})$ is provided at the webpage of the course under the link of project 2. We will use these functions to solve the Hartree-Fock problem for beryllium.
Show that you can simplify the direct term developed during the lectures
$$\int r_1^2 dr_1 \int r_2^2 dr_2 R^*_{n_\alpha 0}(r_1) R^*_{n_\beta 0}(r_2) \frac{1}{(r_>)} R_{n_\gamma 0}(r_1) R_{n_\delta 0}(r_2)$$
$$\int_0^\infty r_1^2 dr_1 R^*_{n_\alpha 0}(r_1) R_{n_\gamma 0}(r_1)\left[\frac{1}{(r_1)}\int_0^{r_1} r_2^2 dr_2 R^*_{n_\beta 0}(r_2) R_{n_\delta 0}(r_2) + \int_{r_1}^\infty r_2 dr_2 R^*_{n_\beta 0}(r_2) R_{n_\delta 0}(r_2)\right].$$
Find the corresponding expression for the exchange term.
4. With the above ingredients we are now ready to solve the Hartree-Fock equations for the beryllium atom. Write a program which solves the Hartree-Fock equations for beryllium. You will need methods to find eigenvalues (see chapter 7) and gaussian quadrature (chapter 5) to compute the integrals of the Coulomb interaction. Use as input for the first iteration the hydrogen-like single-particle wave function. Compare the results (make a plot of the $1s$ and the $2s$ functions) when self-consistency has been achieved with those obtained using the hydrogen-like wave functions only (first iteration). Parameterize thereafter your results in terms of the following Slater-type orbitals (STO)
$$R^{\mathrm{STO}}_{10}(r) = N_{10}\exp{(-\alpha_{10} r)}$$
and
$$R^{\mathrm{STO}}_{20}(r) = N_{20} r\exp{(-\alpha_{20} r/2)}$$
Find the coefficients α_{10} and α_{20} which reproduce best the Hartree-Fock solutions. These functions can then be used in a variational Monte Carlo calculation of the beryllium atom.

15.2. In this problem we will attempt to perform so-called density functional calculations.

1. The first step is to perform a Hartree-Fock calculation using the code developed in the previous exercise but omitting the exchange (Fock) term. Solve the Hartree equation for beryllium and find the total density determined in terms of the single-particle wave functions ψ_i as
$$\rho^H(\mathbf{r}) = \sum_{i=1}^N |\psi_i(\mathbf{r})|^2,$$

where the single-particle functions ψ_i are the solutions of the Hartree equations and the index H refers to the density obtained by solving the Hartree equations. Check that the density is normalised to

$$\int d^3r\rho^H(\mathbf{r}) = N.$$

Compare this density with the corresponding density $\rho^{\mathrm{HF}}(\mathbf{r})$ you get by solving the full Hartree-Fock equations. Compare both the Hartree and Hartree-Fock densities with those resulting from your best VMC calculations. Discuss your results.

2. A popular approximation to the exchange potential in the density functional is to approximate the contribution to this term by the corresponding result from the infinite electron gas model. The exchange term reads then

$$V_x(\mathbf{r}) = -\left(\frac{3}{\pi}\right)^{1/3}\rho^H(\mathbf{r}).$$

Use the Hartree results to compute the total ground state energy of beryllium with the above approximation to the exchange potential. Compare the resulting energy with the resulting Hartree-Fock energy.

15.3. We consider a system of electrons confined in a pure two-dimensional isotropic harmonic oscillator potential, with an idealized total Hamiltonian given by

$$\widehat{\mathbf{H}} = \sum_{i=1}^{N}\left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2\right) + \sum_{i<j}\frac{1}{r_{ij}},$$

where natural units ($\hbar = c = e = m_e = 1$) are used and all energies are in so-called atomic units a.u. We will study systems of many electrons N as functions of the oscillator frequency ω using the above Hamiltonian. The Hamiltonian includes a standard harmonic oscillator part

$$\widehat{\mathbf{H}}_0 = \sum_{i=1}^{N}\left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2\right),$$

and the repulsive interaction between two electrons given by

$$\widehat{\mathbf{H}}_1 = \sum_{i<j}\frac{1}{r_{ij}},$$

with the distance between electrons given by $r_{ij} = \sqrt{\mathbf{r}_1 - \mathbf{r}_2}$. We define the modulus of the positions of the electrons (for a given electron i) as $r_i = \sqrt{r_{i_x}^2 + r_{i_y}^2}$. We limit ourselves to quantum dots with $N = 2$ and $N = 6$ electrons only.

1. The first step is to develop a code that solves the Kohn-Sham equations for $N = 2$ and $N = 6$ quantum dot systems with frequencies $\omega = 0.01$, $\omega = 0.28$ and $\omega = 1.0$ ignoring the exchange contribution. This corresponds to solving the Hartree equations. Solve the Kohn-Sham equations with this approximation for these quantum dot systems and find the total density determined in terms of the single-particle wave functions ψ_i as

$$\rho^H(\mathbf{r}) = \sum_{i=1}^{N}|\psi_i(\mathbf{r})|^2,$$

where the single-particle functions ψ_i are the solutions of the approximated Kohn-Sham equations. Check that the density is normalised to

$$\int d^3r\rho^H(\mathbf{r}) = N.$$

Compare this density with the corresponding density you get by solving the VMC calculations. Discuss your results.

2. A popular approximation to the exchange potential in the density functional is to approximate the contribution to this term by the corresponding result from the infinite electron gas model in two dimensions. For the exchange interaction $V_x(\mathbf{r})$ we will use the local-density approximation of Rajagopal and Kimball, see Phys. Rev. B **15**, 2819 (1977). Use the Kohn-Sham equations to compute the total ground state energy of the same ground states as in 2a) with the above approximation to the exchange potential. Compare the resulting energy with that obtained by performing a Hartree-Fock calculation of these quantum dot systems..

Chapter 16
Improved Monte Carlo Approaches to Systems of Fermions

Abstract This chapter develops the equations and formalism that are necessary to study many-particle systems of fermions. The crucial part of any variational or diffusion Monte Carlo code for many particles is the Metropolis evaluation of the ratios between wave functions and the computation of the local energy. In particular, we develop efficient ways of computing the ratios between Slater determinants

16.1 Introduction

For fermions we need to pay particular attention to the way we treat the Slater determinant. The trial wave function, as discussed in chapter 14, consists of separate factors that incorporate different mathematical properties of the total wave function. There are three types that will be of concern to us: The Slater determinant, the product state, and the correlation factor. The two first are direct functions of the spatial coordinates of the particles, while the last one typically depends on the relative distance between particles.

The trial wave function plays a central role in quantum variational Monte Carlo simulations. Its importance lies in the fact that all the observables are computed with respect to the probability distribution function defined from the trial wave function. Moreover, it is needed in the Metropolis algorithm and in the evaluation of the quantum force term when importance sampling is applied. Computing a determinant of an $N \times N$ matrix by standard Gaussian elimination is of the order of $\mathscr{O}(N^3)$ calculations. As there are $N \cdot d$ independent coordinates we need to evaluate Nd Slater determinants for the gradient (quantum force) and $N \cdot d$ for the Laplacian (kinetic energy). Therefore, it is imperative to find alternative ways of computating quantities related to the trial wave function such that the computational perfomance can be improved.

16.2 Splitting the Slater Determinant

Following for example Ref. [128], assume that we wish to compute the expectation value of a spin-independent quantum mechanical operator $\widehat{\mathbf{O}}(\mathbf{r})$ using the spin-dependent state $\Psi(\mathbf{x})$, where $\mathbf{x} = (\mathbf{r}, \sigma)$ represents the space-spin coordinate par. Then,

$$\langle \widehat{\mathbf{O}} \rangle = \frac{\langle \Psi(\mathbf{x}) | \widehat{\mathbf{O}}(\mathbf{r}) | \Psi(\mathbf{x}) \rangle}{\langle \Psi(\mathbf{x}) | \Psi(\mathbf{x}) \rangle}.$$

If for each spin configuration $\sigma = (\sigma_1, \dots, \sigma_N)$ we replace the total antisymmetric wave function by a version with permuted arguments arranged such that the first $N_\uparrow$ arguments are spin up and the rest $N_\downarrow = N - N_\uparrow$ are spin down we get

$$\begin{aligned}\Psi(\mathbf{x_1}, \dots, \mathbf{x_N}) \rightarrow \Psi(\mathbf{x_{i1}}, \dots, \mathbf{x_{iN}}) & \\ &= \Psi(\{\mathbf{r_{i1}}, \uparrow\}, \dots, \{\mathbf{r_{iN_\uparrow}}, \uparrow\}, \{\mathbf{r_{iN_{\uparrow+1}}}, \downarrow\}, \dots, \{\mathbf{r_{iN}}, \downarrow\}) \\ &= \Psi(\{\mathbf{r_1}, \uparrow\}, \dots, \{\mathbf{r_{N_\uparrow}}, \uparrow\}, \{\mathbf{r_{1N_{\uparrow+1}}}, \downarrow\}, \dots, \{\mathbf{r_N}, \downarrow\}).\end{aligned}$$

Because the operator $\widehat{\mathbf{O}}$ is symmetric with respect to the exchange of labels in a pair of particles, each spin configuration gives an identical contribution to the expectation value. Hence,

$$\langle \widehat{\mathbf{O}} \rangle = \frac{\langle \Psi(\mathbf{r}) | \widehat{\mathbf{O}}(\mathbf{r}) | \Psi(\mathbf{r}) \rangle}{\langle \Psi(\mathbf{r}) | \Psi(\mathbf{r}) \rangle}$$

The new state is antisymmetric with respect to exchange of spatial coordinates of pairs of spin-up or spin-down electrons. Therefore, for spin-independent Hamiltonians, the Slater determinant can be splitted in a product of Slater determinants obtained from single particle orbitals with different spins. For electronic systems we get then

$$\Psi_D = D_\uparrow D_\downarrow,$$

where

$$D_\uparrow = |\mathbf{D}(\mathbf{r_1}, \mathbf{r_2}, \dots, \mathbf{r_{N/2}})|_\uparrow = \left| \begin{array}{cccc} \phi_1(\mathbf{r_1}) & \phi_2(\mathbf{r_1}) & \cdots & \phi_{N/2}(\mathbf{r_1}) \\ \phi_1(\mathbf{r_2}) & \phi_2(\mathbf{r_2}) & \cdots & \phi_{N/2}(\mathbf{r_2}) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\mathbf{r_{N/2}}) & \phi_2(\mathbf{r_{N/2}}) & \cdots & \phi_{N/2}(\mathbf{r_{N/2}}) \end{array} \right|_\uparrow . \qquad (16.1)$$

In a similar way, $D_\downarrow = |\mathbf{D}(\mathbf{r_{N/2+1}}, \mathbf{r_{N/2+2}}, \dots, \mathbf{r_N})|_\downarrow$. The normalization factor has been removed, since it cancels in the ratios needed by the variational Monte Carlo algorithm, as shown later. The new state $\Psi_D(\mathbf{r})$ gives in this case the same expectation value as $\Psi(\mathbf{x})$, but is more convenient in terms of computational cost.The Slater determinant can now be factorized as

$$\boxed{\Psi_T(\mathbf{x}) = D_\uparrow D_\downarrow \Psi_C.} \qquad (16.2)$$

16.3 Computational Optimization of the Metropolis/Hasting Ratio

In the Metropolis/Hasting algorithm, the *acceptance ratio* determines the probability for a particle to be accepted at a new position. The ratio of the trial wave functions evaluated at the new and current positions is given by

$$\boxed{R \equiv \frac{\Psi_T^{new}}{\Psi_T^{cur}} = \underbrace{\frac{|D|_\uparrow^{new}}{|D|_\uparrow^{cur}} \frac{|D|_\downarrow^{new}}{|D|_\downarrow^{cur}}}_{R_{SD}} \underbrace{\frac{\Psi_C^{new}}{\Psi_C^{cur}}}_{R_C} .} \qquad (16.3)$$

16.3.1 Evaluating the Determinant-determinant Ratio

Evaluating the determinant of an $N \times N$ matrix by Gaussian elimination takes of the order of $\mathcal{O}(N^3)$ operations, which is rather expensive for a many-particle quantum system. An alternative algorithm not requiring the separated evaluation of the determinants will be derived in the following. We start by defining a Slater matrix $\mathbf{D}$ with its corresponding $(i,j)-$entries given by

$$D_{ij} \equiv \phi_j(\mathbf{r_i}), \tag{16.4}$$

where $\phi_j(\mathbf{r_i})$ is the j^{th} single particle wave function evaluated for the particle at position $\mathbf{r_i}$.

The inverse of a (Slater) matrix is related to its adjoint (transpose matrix of cofactors) and its determinant by

$$\mathbf{D^{-1}} = \frac{adj\mathbf{D}}{|D|} \Rightarrow |D| = \frac{adj\mathbf{D}}{\mathbf{D^{-1}}}, \tag{16.5}$$

or

$$|D| = \sum_{j=1}^{N} \frac{C_{ji}}{D_{ij}^{-1}} = \sum_{j=1}^{N} D_{ij}C_{ji}, \tag{16.6}$$

i.e., the determinant of a matrix equals the scalar product of any column(row) of the matrix with the same column(row) of the matrix of cofactors.

In the particular case when only one particle is moved at the time (say particle at position $\mathbf{r_i}$), this changes only one row (or column)[1] of the Slater matrix. An efficient way of evaluating that ratio is as follows [94, 129].

We define the ratio of the new to the old determinants in terms of Eq. (16.6) such that

$$R_{SD} \equiv \frac{|\mathbf{D}(\mathbf{x^{new}})|}{|\mathbf{D}(\mathbf{x^{cur}})|} = \frac{\sum_{j=1}^{N} D_{ij}(\mathbf{x^{new}})C_{ji}(\mathbf{x^{new}})}{\sum_{j=1}^{N} D_{ij}(\mathbf{x^{cur}})C_{ji}(\mathbf{x^{cur}})}.$$

When the particle at position $\mathbf{r_i}$ is moved, the $i^{th}-$row of the matrix of cofactors remains unchanged, i.e., the row number i of the cofactor matrix are independent of the entries in the rows of its corresponding matrix $\mathbf{D}$. Therefore,

$$C_{ij}(\mathbf{x^{new}}) = C_{ij}(\mathbf{x^{cur}}),$$

and

$$R_{SD} = \frac{\sum_{j=1}^{N} D_{ij}(\mathbf{x^{new}})C_{ji}(\mathbf{x^{cur}})}{\sum_{j=1}^{N} D_{ij}(\mathbf{x^{cur}})C_{ji}(\mathbf{x^{cur}})} = \frac{\sum_{j=1}^{N} D_{ij}(\mathbf{x^{new}})D_{ji}^{-1}(\mathbf{x^{cur}})|D|(\mathbf{x^{cur}})}{\sum_{j=1}^{N} D_{ij}(\mathbf{x^{cur}})D_{ji}^{-1}(\mathbf{x^{cur}})|D|(\mathbf{x^{cur}})}. \tag{16.7}$$

The invertibility of D implies that

$$\sum_{k}^{N} D_{ik}D_{kj}^{-1} = \delta_{ij}. \tag{16.8}$$

Hence, the denominator in Eq. (16.7) is equal to unity. Then,

$$R_{SD} = \sum_{j=1}^{N} D_{ij}(\mathbf{x^{new}})D_{ji}^{-1}(\mathbf{x^{cur}}).$$

Substituting Eq. (16.4) we arrive at

$$\boxed{R_{SD} = \sum_{j=1}^{N} \phi_j(\mathbf{x_i^{new}})D_{ji}^{-1}(\mathbf{x^{cur}})} \tag{16.9}$$

[1] Some authors prefer to express the Slater matrix by placing the orbitals in a row wise order and the position of the particles in a column wise one.

which means that determining R_{SD} when only particle i has been moved, requires only the evaluation of the dot product between a vector containing orbitals (evaluated at the new position) and all the entries in the i^{th} column of the inverse Slater matrix (evaluated at the current position). This requires approximately $\mathscr{O}(N)$ operations.

Further optimizations can be done by noting that when only one particle is moved at the time, one of the two determinants in the numerator and denominator of Eq. (16.3) is unaffected, cancelling each other. This allows us to carry out calculations with only half of the total number of particles every time a move occurs, requiring only $(N/2)^d$ operations, where d is the number of spatial components of the problem, in systems with equal number of electrons with spin up and down. The total number of operations for a problem in three dimensions becomes $(N/2)^3 = N^3/8$, i.e., the total calculations are reduced up to by a factor of eight.

16.4 Optimizing the $\nabla\Psi_T/\Psi_T$ Ratio

Setting $\Psi_D = |D|_\uparrow |D|_\downarrow$ in Eq. (16.2) we get,

$$\begin{aligned}\frac{\nabla\Psi}{\Psi} &= \frac{\nabla(\Psi_D\Psi_C)}{\Psi_D\Psi_C} = \frac{\Psi_C\nabla\Psi_D + \Psi_D\nabla\Psi_C}{\Psi_D\Psi_C} = \frac{\nabla\Psi_D}{\Psi_D} + \frac{\nabla\Psi_C}{\Psi_C} \\ &= \frac{\nabla(|D|_\uparrow |D|_\downarrow)}{|D|_\uparrow |D|_\downarrow} + \frac{\nabla\Psi_C}{\Psi_C},\end{aligned}$$

or

$$\boxed{\frac{\nabla\Psi}{\Psi} = \frac{\nabla(|D|_\uparrow)}{|D|_\uparrow} + \frac{\nabla(|D|_\downarrow)}{|D|_\downarrow} + \frac{\nabla\Psi_C}{\Psi_C}.} \tag{16.10}$$

16.4.1 Evaluating the Gradient-determinant-to-determinant Ratio

The evaluation of Eq. (16.10) requires differentiating the N entries of the Slater matrix with respect to all the d spatial components. Since the evaluation of the Slater determinant scales as $\mathscr{O}(N^3)$ this would involve of the order of $N \cdot d \cdot \mathscr{O}(N^3) \approx \mathscr{O}(N^4)$ floating point operations. A cheaper algorithm can be derived by noting that when only one particle is moved at the time, only one row in the Slater matrix needs to be evaluated again. Thus, only the derivatives of that row with respect to the coordinates of the particle moved need to be updated. Obtaining the gradient-determinant ratio required in Eq. (16.10) becomes straigforward. It is analogous to the procedure used in deriving Eq. (16.3). From Eq. (16.9) and Eq. (16.3) we see that

$$\boxed{\frac{\nabla_{\mathbf{i}}|\mathbf{D}(\mathbf{x})|}{|\mathbf{D}(\mathbf{x})|} = \sum_{j=1}^{N} \nabla_{\mathbf{i}} D_{ij}(\mathbf{x}) D^{-1}_{ji}(\mathbf{x}) = \sum_{j=1}^{N} \nabla_{\mathbf{i}} \phi_j(\mathbf{x_i}) D^{-1}_{ji}(\mathbf{x}),} \tag{16.11}$$

which means that when one particle is moved at the time, the gradient-determinant ratio is given by the dot product between the gradient of the single-particle wave functions evaluated for the particle at position $\mathbf{r_i}$ and the inverse Slater matrix. A small modification has to be done when computing the gradient to determinant ratio after a move has been accepted. Denoting by $\mathbf{y}$ the vector containing the new spatial coordinates, by definition we get,

$$\frac{\nabla_{\mathbf{i}}|\mathbf{D}(\mathbf{y})|}{|\mathbf{D}(\mathbf{y})|} = \sum_{j=1}^{N} \nabla_{\mathbf{i}} D_{ij}(\mathbf{y}) D^{-1}_{ji}(\mathbf{y}) = \sum_{j=1}^{N} \nabla_{\mathbf{i}} \phi_j(\mathbf{y_i}) D^{-1}_{ji}(\mathbf{y}),$$

which can be expressed in terms of the transpose and inverse of the Slater matrix evaluated at the old positions [94] to get

$$\boxed{\frac{\nabla_{\mathbf{i}}|\mathbf{D}(\mathbf{y})|}{|\mathbf{D}(\mathbf{y})|} = \frac{1}{R} \sum_{j=1}^{N} \nabla_{\mathbf{i}} \phi_j(\mathbf{y_i}) D^{-1}_{ji}(\mathbf{x}).} \tag{16.12}$$

Computing a single derivative is an $\mathscr{O}(N)$ operation. Since there are dN derivatives, the total time scaling becomes $\mathscr{O}(dN^2)$.

16.5 Optimizing the $\nabla^2\Psi_T/\Psi_T$ Ratio

From the single-particle kinetic energy operator, the expectation value of the kinetic energy expressed in atomic units for electron i is

$$\langle \widehat{\mathbf{K}}_i \rangle = -\frac{1}{2}\frac{\langle \Psi|\nabla_i^2|\Psi\rangle}{\langle\Psi|\Psi\rangle}, \tag{16.13}$$

which is obtained by using Monte Carlo integration. The energy of each space configuration is cummulated after each Monte Carlo cycle. For each electron we evaluate

$$K_i = -\frac{1}{2}\frac{\nabla_i^2\Psi}{\Psi}. \tag{16.14}$$

Following a procedure similar to that of section 16.4, the term for the kinetic energy is obtained by

$$\begin{aligned}\frac{\nabla^2\Psi}{\Psi} &= \frac{\nabla^2(\Psi_D\Psi_C)}{\Psi_D\Psi_C} = \frac{\nabla\cdot[\nabla(\Psi_D\Psi_C)]}{\Psi_D\Psi_C} = \frac{\nabla\cdot[\Psi_C\nabla\Psi_D + \Psi_D\nabla\Psi_C]}{\Psi_D\Psi_C}\\ &= \frac{\nabla\Psi_C\cdot\nabla\Psi_D + \Psi_C\nabla^2\Psi_D + \nabla\Psi_D\cdot\nabla\Psi_C + \Psi_D\nabla^2\Psi_C}{\Psi_D\Psi_C}\end{aligned} \tag{16.15}$$

$$\begin{aligned}\frac{\nabla^2\Psi}{\Psi} &= \frac{\nabla^2\Psi_D}{\Psi_D} + \frac{\nabla^2\Psi_C}{\Psi_C} + 2\frac{\nabla\Psi_D}{\Psi_D}\cdot\frac{\nabla\Psi_C}{\Psi_C}\\ &= \frac{\nabla^2(|D|_\uparrow|D|_\downarrow)}{(|D|_\uparrow|D|_\downarrow)} + \frac{\nabla^2\Psi_C}{\Psi_C} + 2\frac{\nabla(|D|_\uparrow|D|_\downarrow)}{(|D|_\uparrow|D|_\downarrow)}\cdot\frac{\nabla\Psi_C}{\Psi_C},\end{aligned}$$

or

$$\boxed{\frac{\nabla^2\Psi}{\Psi} = \frac{\nabla^2|D|_\uparrow}{|D|_\uparrow} + \frac{\nabla^2|D|_\downarrow}{|D|_\downarrow} + \frac{\nabla^2\Psi_C}{\Psi_C} + 2\left[\frac{\nabla|D|_\uparrow}{|D|_\uparrow} + \frac{\nabla|D|_\downarrow}{|D|_\downarrow}\right]\cdot\frac{\nabla\Psi_C}{\Psi_C},} \tag{16.16}$$

where the *laplace-determinant-to-determinant ratio* is given by

$$\boxed{\frac{\nabla_i^2|\mathbf{D}(\mathbf{x})|}{|\mathbf{D}(\mathbf{x})|} = \sum_{j=1}^{N} \nabla_i^2 D_{ij}(\mathbf{x}) D^{-1}_{ji}(\mathbf{x}) = \sum_{j=1}^{N} \nabla_i^2 \phi_j(\mathbf{x_i}) D^{-1}_{ji}(\mathbf{x})} \tag{16.17}$$

for particle at $\mathbf{x_i}$ as deduced from Eq. (16.9) and Eq. (16.3). The comments given in section 16.4 on performance yields applies also to this case. Moreover, Eq. (16.17) is computed with the trial move only if it is accepted.

16.6 Updating the Inverse of the Slater Matrix

Computing the ratios in Eqs. (16.9), (16.11), (16.12) and (16.17) requires that we maintain the inverse of the Slater matrix evaluated at the current position. Each time a trial position is accepted, the row number i of the Slater matrix changes and updating its inverse has to be carried out. Getting the inverse of an $N \times N$ matrix by Gaussian elimination has a complexity of order of $\mathscr{O}(N^3)$ operations, a luxury that we cannot afford for each time a particle move is accepted. An alternative way of updating the inverse of a matrix when only a row/column is changed was suggested by Sherman and Morris. It has a time scaling of the order of $\mathscr{O}(N^2)$ [94,128,129] and is given by

$$\boxed{D^{-1}_{kj}(\mathbf{x^{new}}) = \begin{cases} D^{-1}_{kj}(\mathbf{x^{cur}}) - \frac{D^{-1}_{ki}(\mathbf{x^{cur}})}{R} \sum_{l=1}^{N} D_{il}(\mathbf{x^{new}}) D^{-1}_{lj}(\mathbf{x^{cur}}) & \text{if } j \neq i \\ \frac{D^{-1}_{ki}(\mathbf{x^{cur}})}{R} \sum_{l=1}^{N} D_{il}(\mathbf{x^{cur}}) D^{-1}_{lj}(\mathbf{x^{cur}}) & \text{if } j = i \end{cases}} \tag{16.18}$$

The evaluation of the determinant of an $N \times N$ matrix by standard Gaussian elimination requires $\mathscr{O}(N^3)$ calculations. As there are Nd independent coordinates we need to evaluate Nd Slater determinants for the gradient (quantum force) and Nd for the Laplacian (kinetic energy). With the updating algorithm we need only to invert the Slater determinant matrix once. This can be done by the LU decomposition discussed in chapter 6.

Table 16.1 summarizes the computational cost associated with the Slater determinant part of the trial wave function.

Operation	No optimization	With optimization
Evaluation of R	$\mathscr{O}(N^2)$	$\mathscr{O}\left(\frac{N^2}{2}\right)$
Updating inverse	$\mathscr{O}(N^3)$	$\mathscr{O}\left(\frac{N^3}{4}\right)$
Transition of one particle	$\mathscr{O}(N^2)+\mathscr{O}(N^3)$	$\mathscr{O}\left(\frac{N^2}{2}\right)+\mathscr{O}\left(\frac{N^3}{4}\right)$

Table 16.1 Comparison of the computational cost involved in the computation of the Slater determinant with and without optimization.

16.7 Reducing the Computational Cost of the Correlation Form

The total number of different relative distances r_{ij} is $N(N-1)/2$. In a matrix storage format, the set forms a strictly upper triangular matrix[2]

[2] In the implementation, however, we do not store the entries lying on the diagonal.

$$\mathbf{r} \equiv \begin{pmatrix} 0 & r_{1,2} & r_{1,3} & \cdots & r_{1,N} \\ \vdots & 0 & r_{2,3} & \cdots & r_{2,N} \\ \vdots & \vdots & 0 & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & r_{N-1,N} \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}. \tag{16.19}$$

This applies to $\mathbf{g} = \mathbf{g}(r_{ij})$ as well.

16.8 Computing the Correlation-to-correlation Ratio

For the case where all particles are moved simlutaneously, all the g_{ij} have to be reevaluated. The number of operations for getting R_C scales as $\mathscr{O}(N^2)$. When moving only one particle at a time, say the kth, only $N-1$ of the distances r_{ij} having k as one of their indices are changed. It means that the rest of the factors in the numerator of the Jastrow ratio has a similar counterpart in the denominator and cancel each other. Therefore, only $N-1$ factors of Ψ_C^{new} and Ψ_C^{cur} avoid cancellation and

$$\boxed{R_C = \frac{\Psi_C^{\mathrm{new}}}{\Psi_C^{\mathrm{cur}}} = \prod_{i=1}^{k-1} \frac{g_{ik}^{\mathrm{new}}}{g_{ik}^{\mathrm{cur}}} \prod_{i=k+1}^{N} \frac{g_{ki}^{\mathrm{new}}}{g_{ki}^{\mathrm{cur}}}}. \tag{16.20}$$

For the Padé-Jastrow form

$$\boxed{R_C = \frac{\Psi_C^{\mathrm{new}}}{\Psi_C^{\mathrm{cur}}} = \frac{e^{U_{new}}}{e^{U_{cur}}} = e^{\Delta U},} \tag{16.21}$$

where

$$\Delta U = \sum_{i=1}^{k-1} \left(f_{ik}^{\mathrm{new}} - f_{ik}^{\mathrm{cur}} \right) + \sum_{i=k+1}^{N} \left(f_{ki}^{\mathrm{new}} - f_{ki}^{\mathrm{cur}} \right) \tag{16.22}$$

One needs to develop a special algorithm that iterates only through the elements of the upper triangular matrix $\mathbf{g}$ that have k as an index.

16.9 Evaluating the $\nabla\Psi_C/\Psi_C$ Ratio

The expression to be derived in the following is of interest when computing the quantum force and the kinetic energy. It has the form

$$\frac{\nabla_{\mathbf{i}}\Psi_C}{\Psi_C} = \frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_i},$$

for all dimensions and with i running over all particles. From the discussion in section 16.8, for the first derivative only $N-1$ terms survive the ratio because the g-terms that are not differentiated cancel with their corresponding ones in the denominator. Then,

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1} \frac{1}{g_{ik}}\frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N} \frac{1}{g_{ki}}\frac{\partial g_{ki}}{\partial x_k}. \tag{16.23}$$

An equivalent equation is obtained for the exponential form after replacing g_{ij} by $\exp(g_{ij})$, yielding:

$$\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} + \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_k}, \tag{16.24}$$

with both expressions scaling as $\mathscr{O}(N)$.

Later, using the identity

$$\frac{\partial}{\partial x_i} g_{ij} = -\frac{\partial}{\partial x_j} g_{ij} \tag{16.25}$$

on the right hand side terms of Eq. (16.23) and Eq. (16.24), we get expressions where all the derivatives act on the particle are represented by the *second* index of g:

$$\boxed{\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{1}{g_{ik}}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\frac{\partial g_{ki}}{\partial x_i},} \tag{16.26}$$

and for the exponential case:

$$\boxed{\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\partial g_{ik}}{\partial x_k} - \sum_{i=k+1}^{N}\frac{\partial g_{ki}}{\partial x_i}.} \tag{16.27}$$

16.9.1 Special Case: Correlation Functions Depending on the Relative Distance

For correlation forms depending only on the scalar distances r_{ij}, we note that

$$\frac{\partial g_{ij}}{\partial x_j} = \frac{\partial g_{ij}}{\partial r_{ij}}\frac{\partial r_{ij}}{\partial x_j} = \frac{x_j - x_i}{r_{ij}}\frac{\partial g_{ij}}{\partial r_{ij}}, \tag{16.28}$$

after substitution in Eq. (16.26) and Eq. (16.27) we arrive at

$$\boxed{\frac{1}{\Psi_C}\frac{\partial \Psi_C}{\partial x_k} = \sum_{i=1}^{k-1}\frac{1}{g_{ik}}\frac{\mathbf{r_{ik}}}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\frac{\mathbf{r_{ki}}}{r_{ki}}\frac{\partial g_{ki}}{\partial r_{ki}}.} \tag{16.29}$$

Note that for the Padé-Jastrow form we can set $g_{ij} \equiv g(r_{ij}) = e^{f(r_{ij})} = e^{f_{ij}}$ and

$$\frac{\partial g_{ij}}{\partial r_{ij}} = g_{ij}\frac{\partial f_{ij}}{\partial r_{ij}}. \tag{16.30}$$

Therefore,

$$\boxed{\frac{1}{\Psi_{PJ}}\frac{\partial \Psi_{PJ}}{\partial x_k} = \sum_{i=1}^{k-1}\frac{\mathbf{r_{ik}}}{r_{ik}}\frac{\partial f_{ik}}{\partial r_{ik}} - \sum_{i=k+1}^{N}\frac{\mathbf{r_{ki}}}{r_{ki}}\frac{\partial f_{ki}}{\partial r_{ki}},} \tag{16.31}$$

where

$$\mathbf{r}_{ij} = |\mathbf{r}_j - \mathbf{r}_i| = (x_j - x_i)\hat{e}_1 + (y_j - y_i)\hat{e}_2 + (z_j - z_i)\hat{e}_3 \tag{16.32}$$

is the vectorial distance. When the correlation function is the *linear Padé-Jastrow*, we set

$$f_{ij} = \frac{a_{ij} r_{ij}}{(1 + \beta_{ij} r_{ij})}, \tag{16.33}$$

which yields the closed-form expression

$$\boxed{\frac{\partial f_{ij}}{\partial r_{ij}} = \frac{a_{ij}}{(1+\beta_{ij}r_{ij})^2}}. \tag{16.34}$$

16.10 Computing the $\nabla^2\Psi_C/\Psi_C$ Ratio

For deriving this expression we note first that Eq. (16.29) can be written as

$$\nabla_k\Psi_C = \sum_{i=1}^{k-1}\frac{1}{g_{ik}}\nabla_k g_{ik} + \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\nabla_k g_{ki}.$$

After multiplying by Ψ_C and taking the gradient on both sides we get,

$$\begin{aligned}\nabla_k^2\Psi_C &= \nabla_k\Psi_C\cdot\left(\sum_{i=1}^{k-1}\frac{1}{g_{ik}}\nabla_k g_{ik} + \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\nabla_k g_{ki}\right)\\ &\quad + \Psi_C\nabla_k\cdot\left(\sum_{i=k+1}^{N}\frac{1}{g_{ki}}\nabla_k g_{ki} + \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\nabla_k g_{ki}\right)\\ &= \Psi_C\left(\frac{\nabla_k\Psi_C}{\Psi_C}\right)^2 + \Psi_C\nabla_k\cdot\left(\sum_{i=k+1}^{N}\frac{1}{g_{ki}}\nabla_k g_{ki} + \sum_{i=k+1}^{N}\frac{1}{g_{ki}}\nabla_k g_{ki}\right).\end{aligned} \tag{16.35}$$

Now,

$$\begin{aligned}\nabla_k\cdot\left(\frac{1}{g_{ik}}\nabla_k g_{ik}\right) &= \nabla_k\left(\frac{1}{g_{ik}}\right)\cdot\nabla_k g_{ik} + \frac{1}{g_{ik}}\nabla_k\cdot\nabla_k g_{ik}\\ &= -\frac{1}{g_{ik}^2}\nabla_k g_{ik}\cdot\nabla_k g_{ik} + \frac{1}{g_{ik}}\nabla_k\cdot\left(\frac{\mathbf{r}_{ik}}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right)\\ &= -\frac{1}{g_{ik}^2}(\nabla_k g_{ik})^2\\ &\quad + \frac{1}{g_{ik}}\left[\nabla_k\left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right)\cdot\mathbf{r}_{ik} + \left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right)\nabla_k\cdot\mathbf{r}_{ik}\right]\\ &= -\frac{1}{g_{ik}^2}\left(\frac{\mathbf{r}_{ik}}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right)^2\\ &\quad + \frac{1}{g_{ik}}\left[\nabla_k\left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right)\cdot\mathbf{r}_{ik} + \left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right)d\right]\\ &= -\frac{1}{g_{ik}^2}\left(\frac{\partial g_{ik}}{\partial r_{ik}}\right)^2\\ &\quad + \frac{1}{g_{ik}}\left[\nabla_k\left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right)\cdot\mathbf{r}_{ik} + \left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right)d\right],\end{aligned} \tag{16.36}$$

with d being the number of spatial dimensions.

Moreover,

$$\begin{aligned}\nabla_k\left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right) &= \frac{\mathbf{r}_{ik}}{r_{ik}}\frac{\partial}{\partial r_{ik}}\left(\frac{1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}}\right)\\ &= \frac{\mathbf{r}_{ik}}{r_{ik}}\left(-\frac{1}{r_{ik}^2}\frac{\partial g_{ik}}{\partial r_{ik}} + \frac{1}{r_{ik}}\frac{\partial^2 g_{ik}}{\partial r_{ik}^2}\right).\end{aligned}$$

The substitution of the last result in Eq. (16.36) gives

$$\nabla_k \cdot \left(\frac{1}{g_{ik}}\nabla_k g_{ik}\right) = -\frac{1}{g_{ik}^2}\left(\frac{\partial g_{ik}}{\partial r_{ik}}\right)^2 + \frac{1}{g_{ik}}\left[\left(\frac{d-1}{r_{ik}}\right)\frac{\partial g_{ik}}{\partial r_{ik}} + \frac{\partial^2 g_{ik}}{\partial r_{ik}^2}\right].$$

Inserting the last expression in Eq. (16.35) and after division by Ψ_C we get,

$$\begin{aligned}\frac{\nabla_k^2\Psi_C}{\Psi_C} &= \left(\frac{\nabla_k\Psi_C}{\Psi_C}\right)^2 \\ &+ \sum_{i=1}^{k-1} -\frac{1}{g_{ik}^2}\left(\frac{\partial g_{ik}}{\partial r_{ik}}\right)^2 + \frac{1}{g_{ik}}\left[\left(\frac{d-1}{r_{ik}}\right)\frac{\partial g_{ik}}{\partial r_{ik}} + \frac{\partial^2 g_{ik}}{\partial r_{ik}^2}\right] \\ &+ \sum_{i=k+1}^{N} -\frac{1}{g_{ki}^2}\left(\frac{\partial g_{ki}}{\partial r_{ki}}\right)^2 + \frac{1}{g_{ki}}\left[\left(\frac{d-1}{r_{ki}}\right)\frac{\partial g_{ki}}{\partial r_{ki}} + \frac{\partial^2 g_{ki}}{\partial r_{ki}^2}\right].\end{aligned} \qquad (16.37)$$

For the exponential case we have

$$\begin{aligned}\frac{\nabla_k^2\Psi_{PJ}}{\Psi_{PJ}} &= \left(\frac{\nabla_k\Psi_{PJ}}{\Psi_{PJ}}\right)^2 \\ &+ \sum_{i=1}^{k-1} -\frac{1}{g_{ik}^2}\left(g_{ik}\frac{\partial f_{ik}}{\partial r_{ik}}\right)^2 + \frac{1}{g_{ik}}\left[\left(\frac{d-1}{r_{ik}}\right)g_{ik}\frac{\partial f_{ik}}{\partial r_{ik}} + \frac{\partial}{\partial r_{ik}}\left(g_{ik}\frac{\partial f_{ik}}{\partial r_{ik}}\right)\right] \\ &+ \sum_{i=k+1}^{N} -\frac{1}{g_{ki}^2}\left(g_{ik}\frac{\partial f_{ki}}{\partial r_{ki}}\right)^2 + \frac{1}{g_{ki}}\left[\left(\frac{d-1}{r_{ki}}\right)g_{ki}\frac{\partial f_{ki}}{\partial r_{ki}} + \frac{\partial}{\partial r_{ki}}\left(g_{ki}\frac{\partial f_{ki}}{\partial r_{ki}}\right)\right].\end{aligned}$$

Using

$$\begin{aligned}\frac{\partial}{\partial r_{ik}}\left(g_{ik}\frac{\partial f_{ik}}{\partial r_{ik}}\right) &= \frac{\partial g_{ik}}{\partial r_{ik}}\frac{\partial f_{ik}}{\partial r_{ik}} + g_{ik}\frac{\partial^2 f_{ik}}{\partial r_{ik}^2} \\ &= g_{ik}\frac{\partial f_{ik}}{\partial r_{ik}}\frac{\partial f_{ik}}{\partial r_{ik}} + g_{ik}\frac{\partial^2 f_{ik}}{\partial r_{ik}^2} \\ &= g_{ik}\left(\frac{\partial f_{ik}}{\partial r_{ik}}\right)^2 + g_{ik}\frac{\partial^2 f_{ik}}{\partial r_{ik}^2}\end{aligned}$$

and substituting this result into the equation above gives rise to the final expression,

$$\begin{aligned}\frac{\nabla_k^2\Psi_{PJ}}{\Psi_{PJ}} &= \left(\frac{\nabla_k\Psi_{PJ}}{\Psi_{PJ}}\right)^2 \\ &+ \sum_{i=1}^{k-1}\left[\left(\frac{d-1}{r_{ik}}\right)\frac{\partial f_{ik}}{\partial r_{ik}} + \frac{\partial^2 f_{ik}}{\partial r_{ik}^2}\right] + \sum_{i=k+1}^{N}\left[\left(\frac{d-1}{r_{ki}}\right)\frac{\partial f_{ki}}{\partial r_{ki}} + \frac{\partial^2 f_{ki}}{\partial r_{ki}^2}\right].\end{aligned} \qquad (16.38)$$

Again, for the *linear Padé-Jastrow*, we get in this case the closed-form result

$$\boxed{\frac{\partial^2 f_{ij}}{\partial r_{ij}^2} = -\frac{2a_{ij}\beta_{ij}}{(1+\beta_{ij}r_{ij})^3}}. \qquad (16.39)$$

16.11 Efficient Optimization of the Trial Wave Function

Energy minimization requires the evaluation of the derivative of the trial wave function with respect to the variational parameters. The computational cost of this operation depends, of course, on the algorithm selected. In practice, evaluating the derivatives of the trial wave function with respect to the variational parameters analitically is possible only for small systems (two to four electrons). On the other hand, the numerical solution needs the repetead evaluation of the trial wave function (the product of a Slater determinant by a Jastrow function) with respect to each variational parameter. As an example, consider using a central difference scheme to evaluate the derivative of the Slater determinant part with respect to a parameter α,

$$\frac{d\Psi_{SD}}{d\alpha} = \frac{\Psi_{SD}(\alpha+\Delta\alpha)-\Psi_{SD}(\alpha-\Delta\alpha)}{2\Delta\alpha} + \mathscr{O}(\Delta\alpha^2).$$

The reader should note that for the Slater determinant part we need to compute the expression above two times per Monte Carlo cycle per variational parameter. Computing a determinant is a highly costly operation. Moreover, the numerical accuracy in the solution will depend on the choice of the step size $\Delta\alpha$.

In the following we suggest a method to efficiently compute the derivative of the energy with respect to the variational parameters. It derives from the fact that the energy derivative is equivalent to

$$\frac{\partial E}{\partial c_m} = 2\left[\left\langle E_L \frac{\partial \ln \Psi_{T_{c_m}}}{\partial c_m}\right\rangle - E\left\langle \frac{\partial \ln \Psi_{T_{c_m}}}{\partial c_m}\right\rangle\right],$$

or more precisicely,

$$\boxed{\frac{\partial E}{\partial c_m} = 2\left\{\frac{1}{N}\sum_{i=1}^{N}\left[(E_L[c_m])_i\left(\frac{\partial \ln \Psi_{T_c}}{\partial c_m}\right)_i\right] - \frac{1}{N^2}\sum_{i=1}^{N}(E_L[c_m])_i\sum_{j=1}^{N}\left(\frac{\partial \ln \Psi_{T_c}}{\partial c_m}\right)_j\right\}}, \tag{16.40}$$

and because $\Psi_{T_{c_m}} = \Psi_{SD_{c_m}}\Psi_{J_{c_m}}$, we get that

$$\begin{aligned}\ln \Psi_{T_{c_m}} &= \ln(\Psi_{SD_{c_m}}\Psi_{J_{c_m}}) = \ln(\Psi_{SD_{c_m}}) + \ln(\Psi_{J_{c_m}}) \\ &= \ln(\Psi_{SD_{c_m}\uparrow}\Psi_{SD_{c_m}\downarrow}) + \ln(\Psi_{J_{c_m}}) \\ &= \ln(\Psi_{SD_{c_m}\uparrow}) + \ln(\Psi_{SD_{c_m}\downarrow}) + \ln(\Psi_{J_{c_m}}).\end{aligned}$$

Then,

$$\boxed{\frac{\partial \ln \Psi_{T_{c_m}}}{\partial c_m} = \frac{\partial \ln(\Psi_{SD_{c_m}\uparrow})}{\partial c_m} + \frac{\partial \ln(\Psi_{SD_{c_m}\downarrow})}{\partial c_m} + \frac{\partial \ln(\Psi_{J_{c_m}})}{\partial c_m}}, \tag{16.41}$$

which is a convenient expression in terms of implementation in an object oriented fashion because we can compute the contribution to the expression above in two separated classes independently, namely the Slater determinant and Jastrow classes.

Note also that for each of the derivatives of concerning the determinants above we have, in general, that

$$\frac{\partial \ln(\Psi_{SD_{c_m}})}{\partial c_m} = \frac{\frac{\partial \Psi_{SD_{c_m}}}{\partial c_m}}{\Psi_{SD_{c_m}\uparrow}}$$

For the derivative of the Slater determinant yields that if $\mathbf{A}$ is an invertible matrix which depends on a real parameter t, and if $\frac{\mathrm{d}\mathbf{A}}{\mathrm{d}t}$ exists, then

$$\frac{\mathrm{d}}{\mathrm{d}t}(||A) = (||A)\,\mathrm{tr}\left(\mathbf{A}^{-1}\frac{\mathrm{d}\mathbf{A}}{\mathrm{d}t}\right).$$

$$\boxed{\frac{d}{dt}\ln||A(t) = \mathrm{tr}\left(\mathbf{A}^{-1}\frac{d\mathbf{A}}{dt}\right) = \sum_{i=1}^{N}\sum_{j=1}^{N}A_{ij}^{-1}\dot{A}_{ji}}, \tag{16.42}$$

where N is the number of entries in a row. What we have here is the expression for computing the derivative of each of the determinants appearing in Eq. (16.41). Furthemore, note that the specialization of this expression to the current problem implies that the term $\mathbf{A}^{-1}$ appearing on the right hand side is the inverse of the Slater matrix, already available after finishing each Monte Carlo cycle as deduced from the algorithms discussed in the previous sections. It means that the only thing we have to do is to take the derivative of each single wave function in the Slater matrix with respect to its variational parameter and taking the trace of $\Psi_{SD}(\alpha)^{-1}\dot{\Psi}_{SD}(\alpha)$. The implementation of this expression and its computation using analytical derivatives for the single state wave functions is straighforward. The flow chart for the Quantum Variational Monte Carlo method with optimization of the trial wave function is shown in figure 16.1.

16.12 Exercises

16.1. 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

$$-\frac{2}{r_1}-\frac{2}{r_2}, \tag{16.43}$$

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

$$V(r_1,r_2) = -\frac{2}{r_1}-\frac{2}{r_2}+\frac{1}{r_{12}}, \tag{16.44}$$

yielding the total Hamiltonian

$$\widehat{\mathbf{H}} = -\frac{\nabla_1^2}{2}-\frac{\nabla_2^2}{2}-\frac{2}{r_1}-\frac{2}{r_2}+\frac{1}{r_{12}}, \tag{16.45}$$

and Schrödinger's equation reads

$$\widehat{\mathbf{H}}\psi = E\psi. \tag{16.46}$$

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..

1. 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**.
2. 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

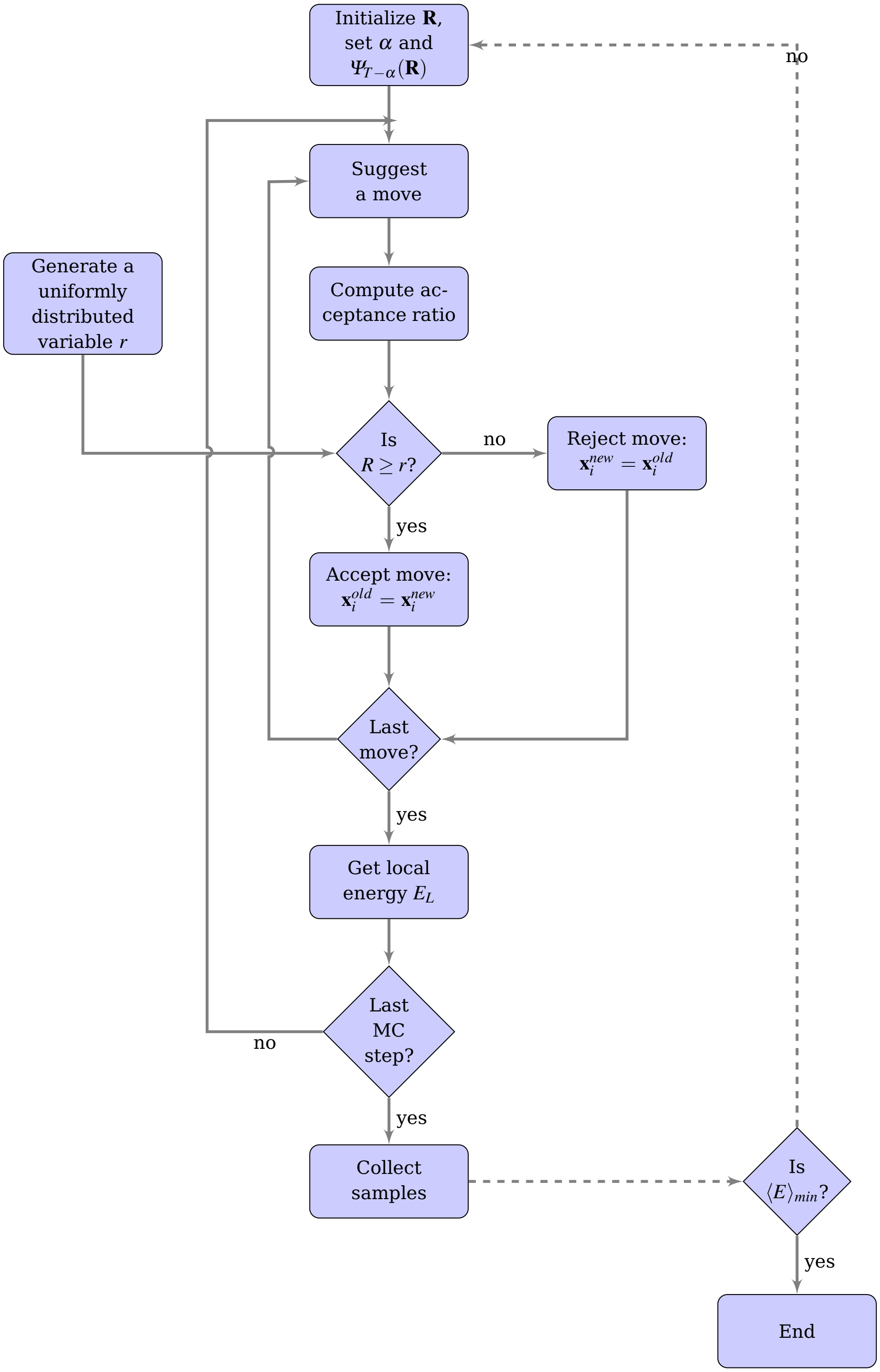

Initialize **R**, set α and $\Psi_{T-\alpha}(\mathbf{R})$
Suggest a move
Compute acceptance ratio
Generate a uniformly distributed variable r
Is $R \geq r$?
no
Reject move: $\mathbf{x}_i^{new} = \mathbf{x}_i^{old}$
yes
Accept move: $\mathbf{x}_i^{old} = \mathbf{x}_i^{new}$
Last move?
yes
Get local energy E_L
Last MC step?
no
yes
Collect samples
Is $\langle E \rangle_{min}$?
no
yes
End

Fig. 16.1 Optimization of the trial wave function $\Psi_{trial}(\alpha)$ and minimization of the energy with respect to the variational parameters.

which you need to program will be discussed during the lectures. Compare the results with those obtained using the hydrogen-like wave functions only.

3. 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

$$\psi_T(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}}) = \exp\left(-\alpha(r_1+r_2)\right)\exp\left(\frac{r_{12}}{2(1+\beta r_{12})}\right), \tag{16.47}$$

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

$$\langle E\rangle = \frac{\int d\mathbf{r_1}d\mathbf{r_2}\psi_T^*(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})\widehat{\mathbf{H}}(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})\psi_T(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})}{\int d\mathbf{r_1}d\mathbf{r_2}\psi_T^*(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})\psi_T(\mathbf{r_1},\mathbf{r_2},\mathbf{r_{12}})}. \tag{16.48}$$

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.

4. Repeat the last step but use now importance sampling. Study the dependence of the results as function of the time step δt.

5. Our final step is to replace the hydrogen-like orbits in Eq. (16.47) 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

$$\psi_T(\mathbf{r_1},\mathbf{r_2},\mathbf{r_3},\mathbf{r_4}) = Det\left(\phi_1(\mathbf{r_1}),\phi_2(\mathbf{r_2}),\phi_3(\mathbf{r_3}),\phi_4(\mathbf{r_4})\right)\prod_{i<j}^{4}\exp\left(\frac{r_{ij}}{2(1+\beta r_{ij})}\right), \tag{16.49}$$

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

$$\phi_{1s}(\mathbf{r_i}) = e^{-\alpha r_i}, \tag{16.50}$$

and

$$\phi_{2s}(\mathbf{r_i}) = (1-\alpha r_i/2)e^{-\alpha r_i/2}. \tag{16.51}$$

For neon , the trial wave function can take the form

$$\psi_T(\mathbf{r_1},\mathbf{r_2},\dots,\mathbf{r_{10}}) = Det\left(\phi_1(\mathbf{r_1}),\phi_2(\mathbf{r_2}),\dots,\phi_{10}(\mathbf{r_{10}})\right)\prod_{i<j}^{10}\exp\left(\frac{r_{ij}}{2(1+\beta r_{ij})}\right), \tag{16.52}$$

In this case you need to include the $2p$ wave function as well. It is given as

$$\phi_{2p}(\mathbf{r_i}) = \alpha \mathbf{r_i} e^{-\alpha r_i/2}. \tag{16.53}$$

Observe that $r_i = \sqrt{r_{i_x}^2 + r_{i_y}^2 + r_{i_z}^2}$.

6. 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.
7. 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).
8. Write a function which sets up the Slater determinant for beryllium and neon. 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. The calculations should include parallelization, blocking and importance sampling. Compare the results with the Hartree-Fock results. How important is the correlation part? Is there a difference compared with helium? Comment your results.

16.2. The aim of this project is to use the Variational Monte Carlo (VMC) method to evaluate the ground state energy, onebody densities, expectation values of the kinetic and potential energies and single-particle energies of quantum dots with $N = 2$, $N = 6$ and $N = 12$ electrons, so-called closed shell systems.

We consider a system of electrons confined in a pure two-dimensional isotropic harmonic oscillator potential, with an idealized total Hamiltonian given by

$$\widehat{\mathbf{H}} = \sum_{i=1}^{N} \left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2 \right) + \sum_{i<j} \frac{1}{r_{ij}}, \tag{16.54}$$

where natural units ($\hbar = c = e = m_e = 1$) are used and all energies are in so-called atomic units a.u. We will study systems of many electrons N as functions of the oscillator frequency ω using the above Hamiltonian. The Hamiltonian includes a standard harmonic oscillator part

$$\widehat{\mathbf{H}}_0 = \sum_{i=1}^{N} \left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2 \right),$$

and the repulsive interaction between two electrons given by

$$\widehat{\mathbf{H}}_1 = \sum_{i<j} \frac{1}{r_{ij}},$$

with the distance between electrons given by $r_{ij} = \sqrt{\mathbf{r}_1 - \mathbf{r}_2}$. We define the modulus of the positions of the electrons (for a given electron i) as $r_i = \sqrt{r_{i_x}^2 + r_{i_y}^2}$.

1a) In exercises 1a-1e we will deal only with a system of two electrons in a quantum dot with a frequency of $\hbar\omega = 1$. The reason for this is that we have exact closed form expressions for the ground state energy from Taut's work for selected values of ω, see M. Taut, Phys. Rev. A **48**, 3561 (1993). The energy is given by 3 a.u. (atomic units) when the interaction between the electrons is included. If only the harmonic oscillator part of the Hamiltonian, the so-called unperturbed part,

$$\widehat{\mathbf{H}}_0 = \sum_{i=1}^{N} \left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2 \right),$$

the energy is 2 a.u. The wave function for one electron in an oscillator potential in two dimensions is

$$\phi_{n_x,n_y}(x,y) = AH_{n_x}(\sqrt{\omega}x)H_{n_y}(\sqrt{\omega}y)\exp\left(-\omega(x^2+y^2)/2.\right.$$

The functions $H_{n_x}(\sqrt{\omega}x)$ are so-called Hermite polynomials, discussed in appendix while A is a normalization constant. For the lowest-lying state we have $n_x = n_y = 0$ and an energy $\varepsilon_{n_x,n_y} = \omega(n_x+n_y+1) = \omega$. Convince yourself that the lowest-lying energy for the two-electron system is simply 2ω.

The unperturbed wave function for the ground state of the two-electron system is given by

$$\Phi(\mathbf{r_1},\mathbf{r_2}) = C\exp\left(-\omega(r_1^2+r_2^2)/2\right),$$

with C being a normalization constant and $r_i = \sqrt{r_{i_x}^2+r_{i_y}^2}$. Note that the vector $\mathbf{r_i}$ refers to the x and y position for a given particle. What is the total spin of this wave function? Find arguments for why the ground state should have this specific total spin.

1b) We want to perform a Variational Monte Carlo calculation of the ground state of two electrons in a quantum dot well with different oscillator energies, assuming total spin $S=0$ using the Hamiltonian of Eq. (16.54). In our first attempt we will use a brute force Metropolis sampling with a trial wave function which has the following form

$$\psi_T(\mathbf{r_1},\mathbf{r_2}) = C\exp\left(-\alpha\omega(r_1^2+r_2^2)/2\right)\exp\left(\frac{ar_{12}}{(1+\beta r_{12})}\right), \tag{16.55}$$

where a is equal to one when the two electrons have anti-parallel spins and $1/3$ when the spins are parallel. Finally, α and β are our variational parameters.

Your task is to perform a Variational Monte Carlo calculation using the Metropolis algorithm to compute the integral

$$\langle E\rangle = \frac{\int d\mathbf{r_1}d\mathbf{r_2}\psi_T^*(\mathbf{r_1},\mathbf{r_2})\widehat{\mathbf{H}}(\mathbf{r_1},\mathbf{r_2})\psi_T(\mathbf{r_1},\mathbf{r_2})}{\int d\mathbf{r_1}d\mathbf{r_2}\psi_T^*(\mathbf{r_1},\mathbf{r_2})\psi_T(\mathbf{r_1},\mathbf{r_2})}. \tag{16.56}$$

You should parallelize your program. As an optional possibility, to program GPUs can be used instead of standard parallelization with MPI throughout the project.

Find the energy minimum and compute also the mean distance $r_{12} = \sqrt{\mathbf{r}_1 - \mathbf{r}_2}$ (with $r_i = \sqrt{r_{i_x}^2+r_{i_y}^2}$) between the two electrons for the optimal set of the variational parameters. A code for doing a VMC calculation for a two-electron system (the three-dimensional helium atom) can be found on the webpage of the course, see under programs.

You should also find a closed-form expression for the local energy. Compare the results of this calculation (in terms of CPU time) compared with a calculation which performs a brute force numerical derivation.

1c) Introduce now importance sampling and study the dependence of the results as a function of the time step δt. Compare the results with those obtained under 1a) and comment eventual differences. 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.

1d) With the optimal parameters for the ground state wave function, compute the onebody density. Discuss your results and compare the results with those obtained with a pure harmonic oscillator wave functions. Run a Monte Carlo calculations without the Jastrow factor as well and compute the same quantities. How important are the correlations induced by the Jastrow factor? Compute also the expectation value of the kinetic energy and potential energy using $\omega = 0.01$, $\omega = 0.28$ and $\omega = 1.0$. Comment your results.

1e) Repeat step 1c) by varying the energy using the conjugate gradient method to obtain the best possible set of parameters α and β. Discuss the results.

The previous exercises have prepared you for extending your calculational machinery to other systems. Here we will focus on quantum dots with $N=6$ and $N=12$ electrons. 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, value of n_x and n_y 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 harmonic oscillator like wave functions, the trial wave function for say an $N = 6$ electron quantum dot can be written as

$$\psi_T(\mathbf{r_1},\mathbf{r_2},\dots,\mathbf{r_6}) = Det\left(\phi_1(\mathbf{r_1}),\phi_2(\mathbf{r_2}),\dots,\phi_6(\mathbf{r_6})\right)\prod_{i<j}^{6}\exp\left(\frac{ar_{ij}}{(1+\beta r_{ij})}\right), \tag{16.57}$$

where Det is a Slater determinant and the single-particle wave functions are the harmonic oscillator wave functions for the $n_x = 0,1$ and $n_y = 0,1$ orbitals. For the $N = 12$ quantum dot, the trial wave function can take the form

$$\psi_T(\mathbf{r_1},\mathbf{r_2},\dots,\mathbf{r_{12}}) = Det\left(\phi_1(\mathbf{r_1}),\phi_2(\mathbf{r_2}),\dots,\phi_{12}(\mathbf{r_{12}})\right)\prod_{i<j}^{12}\exp\left(\frac{ar_{ij}}{2(1+\beta r_{ij})}\right), \tag{16.58}$$

In this case you need to include the $n_x = 2$ and $n_y = 2$ wave functions as well. Observe that $r_i = \sqrt{r_{i_x}^2 + r_{i_y}^2}$. Use the Hermite polynomials defined in the appendix.

(1f) Write a function which sets up the Slater determinant handle larger systems as well. Find the Hermite polynomials which are needed for $n_x = 0,1,2$ and obviously n_y as well. Compute the ground state energies of quantum dots for $N = 6$ and $N = 12$ electrons, following the same set up as in exercise 1e) for $\omega = 0.01$, $\omega = 0.28$ and $\omega = 1.0$. The calculations should include parallelization, blocking, importance sampling and energy minimization using the conjugate gradient approach. To test your Slater determinant code, you should reproduce the unperturbed single-particle energies when the electron-electron repulsion is switched off. Convince yourself that the unperturbed ground state energies for $N = 6$ is 10ω and for $N = 12$ we obtain 28ω. What is the expected total spin of the ground states?

1g) With the optimal parameters for the ground state wave function, compute again the onebody density. Discuss your results and compare the results with those obtained with a pure harmonic oscillator wave functions. Run a Monte Carlo calculations without the Jastrow factor as well and compute the same quantities. How important are the correlations induced by the Jastrow factor? Compute also the expectation value of the kinetic energy and potential energy using $\omega = 0.01$, $\omega = 0.28$ and $\omega = 1.0$. Comment your results.

Additional material on Hermite polynomials

The Hermite polynomials are the solutions of the following differential equation

$$\frac{d^2H(x)}{dx^2} - 2x\frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0. \tag{16.59}$$

The first few polynomials are

$$H_0(x) = 1,$$

$$H_1(x) = 2x,$$

$$H_2(x) = 4x^2 - 2,$$

$$H_3(x) = 8x^3 - 12x,$$

and

$$H_4(x) = 16x^4 - 48x^2 + 12.$$

They fulfil the orthogonality relation

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x)^2 dx = 2^n n! \sqrt{\pi},$$

and the recursion relation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x).$$

Chapter 17
Bose-Einstein condensation and Diffusion Monte Carlo

Abstract We discuss how to perform diffusion Monte Carlo calculations for systems of bosons

17.1 Diffusion Monte Carlo

The DMC method belongs to a larger class of methods often called *projector* Monte Carlo. As the name indicates, this is a general class of methods based on taking projections in Hilbert space.

Consider a system governed by a Hamiltonian $\widehat{H}$. Its stationary eigenfunctions are then given by

$$\widehat{H}\phi_i = \varepsilon_i\phi_i$$

DMC is a method of projecting out the ϕ_i from Ψ with the lowest energy. Most often, and in our case particularly, we are interested in the ground state. But just as with the variational principle used by VMC, if we let Ψ fulfill a certain mathematical symmetry, the projected ϕ_k will be the state of lowest energy with that given symmetry.

In contrast to VMC, which relies on the variational principle, DMC does not directly depend on any a priori choice of wave function that ultimately restricts the quality of the result. Thus, DMC can in principle produce the exact ground state of our system, at least within the statistical limits of the algorithm.

The procedure of projecting out the component state of Ψ with the lowest energy is based on operating on Ψ with the special evolution operator $\exp(-\widehat{H}t)$

$$e^{(-\widehat{H}t)}\Psi(x) = \sum_i c_i \exp(-\varepsilon_i t)\phi_i(x)$$

This is just the formal solution of the special equation:

$$-\frac{\partial}{\partial t}\Psi(x,t) = \widehat{H}\Psi(x,t) \tag{17.1}$$

which resembles the time dependent Schrödinger equation as if it were transformed to imaginary time $it \to t$. Recall that the formal solution of the time dependent Schrödinger equation is just

$$\Psi(x,t) = e^{-\frac{i}{\hbar}\widehat{H}t}\,\Psi(x) = \sum_i c_i e^{-\frac{i}{\hbar}\varepsilon_i t}\,\phi_i(x)$$

As we let $t \to \infty$, the exponential makes all the eigenstates with negative energy blow up while the ones with positive energy vanish. To control this effect we introduce a constant energy shift E_{T}, called a *trial energy*, to the potential term of $\widehat{H}$. This shift does, of course, not change any relevant physical properties of our system since it is generally independent of the choice of the zero point of the energy. The effect on the projection operation becomes

$$\Psi(x,t) = e^{-(\widehat{H}-E_{\mathrm{T}})t}\Psi(x) = \sum_i c_i e^{-(\varepsilon_i - E_{\mathrm{T}})t}\phi_i(x) \tag{17.2}$$

Consider the ideal situation of letting E_{T} equal exactly ε_0, resulting in

$$\Psi(x,t) = e^{-(\widehat{H}-E_{\mathrm{T}})t}\Psi(x) = c_0\phi_0 + \sum_{i>0} c_i\phi_i(x)e^{-(\varepsilon_i-\varepsilon_0)t}$$

Since $\varepsilon_i > \varepsilon_0$ for $i \neq 0$, all the remaining exponents become negative. In the limit $t \to \infty$, the contributions from excited states must obviously vanish, so that propagating Ψ according to Eq. (17.1) gives

$$\lim_{t\to\infty} e^{-(\widehat{H}-\varepsilon_0)t}\Psi(x) = c_0\phi_0$$

thus projecting out the ground state.

Without even considering how to do the time propagation in practice, the formulas already indicate to us that approaching the problem first with VMC can help produce an initial Ψ close to ϕ_0 and, more importantly, a trial energy E_{T} being an upper bound to the true ground state energy ε_0. Hopefully E_{T} is close enough to ε_0 to be smaller than the first excited energy. Inserting such a E_{T} into Eq. (17.2) while letting $t \to \infty$ will make all the excited states collapse while the ground state blows up and dominates because of the positive exponent in the coefficient.

From this consideration we also see that the evolution operator is not unitary. The norm of Ψ is not necessarily conserved with time. Depending on the value of E_{T} it may grow without limit or go to zero. Only in the case of $E_{\mathrm{T}} = \varepsilon_0$ the state tends to a constant. This proposes a method of determining the ground state energy by adjusting E_{T} dynamically during time propagation so that the state stays constant.

Writing out the imaginary time Schrödinger equation, Eq. (17.1), including the energy offset E_{T}, we get

$$\begin{aligned}\frac{\partial}{\partial t}\Psi(x,t) &= -\widehat{K}\Psi(x,t) - (\widehat{V}(x) - E_{\mathrm{T}})\Psi(x,t)\\ &= \frac{\hbar^2}{2m}\nabla^2\Psi(x,t) - (\widehat{V}(x) - E_{\mathrm{T}})\Psi(x,t)\end{aligned} \tag{17.3}$$

where $\widehat{K}$ and $\widehat{V}$ is the kinetic and potential energy operator, respectively. We see that this is of the form of an extended diffusion equation. We can therefore consider the wave function Ψ as a probability distribution evolving according to this equation. This greatly contrasts the usual quantum mechanical interpretation of $|\Psi|^2$ being the actual probability density function (PDF). We will see that the approach poses some potentially serious problems when the wave function Ψ we seek has nodes and is partially negative and possibly complex. But to illustrate the main mechanisms of the method we will at the moment focus on the simple cases of bosonic ground states which usually are strictly positive and real.

In simple terms, the job consists of representing the initial state Ψ by a collection of walkers, much the same way as in VMC, and letting them evolve in time as a controlled diffusion process governed by Eq. (17.3). Interpreting the rhs. terms of Eq. (17.3), we see that the first term is a standard diffusion term with a diffusion constant of

$$D \equiv \frac{\hbar^2}{2m} \tag{17.4}$$

The second term is called a *branching term*. When positive, it induces a growth of the number of walkers and a decay if it is negative.

Before we expand on how to carry out DMC in practice, it may be helpful to consider an analytically exact approach by focusing on the Green's function corresponding to the time evolution of the wave function. The approach is called *Green's function Monte Carlo* (GFMC) and shares its main ideas with the DMC method. Using Dirac's bracket notation, we start over by substituting the initial wave function $\Psi(x)$ with its corresponding state ket $|\Psi\rangle$ and let the special evolution operator work on $|\Psi\rangle$

$$|\Psi\rangle_t = e^{-(\widehat{H}-E_{\mathrm{T}})t}|\Psi\rangle,$$

Now we switch to position representation:

$$\Psi(x,t) = \langle x|\Psi\rangle_t = \langle x|e^{-(\widehat{H}-E_{\mathrm{T}})t}|\Psi\rangle = \int \langle x|e^{-(\widehat{H}-E_{\mathrm{T}})t}|x'\rangle\langle x'|\Psi\rangle dx'$$

where we have just inserted a completeness relation, $1 = \int |x'\rangle\langle x'|dx'$. Defining the Green's function for this case

$$G(x,x',t) \equiv \langle x|e^{-(\widehat{H}-E_{\mathrm{T}})t}|x'\rangle = \langle x|e^{-(\widehat{K}+\widehat{V}-E_{\mathrm{T}})t}|x'\rangle$$

we get:

$$\Psi(x,t) = \int G(x,x',t)\Psi(x')dx' \tag{17.5}$$

Calculating the Green's function G would be greatly simplified if we could split it into separate factors for each of the terms $\widehat{K}$ and $(\widehat{V}-E_{\mathrm{T}})$ of the Hamiltonian $\widehat{H}$, as with the following factorization

$$e^{\widehat{A}+\widehat{B}} = e^{\widehat{A}}e^{\widehat{B}}$$

However, such a factorization requires that the operators $\widehat{A}$ and $\widehat{B}$ commute, $[A,B]=0$. Unfortunately, this is not the case for the terms $\widehat{K}$ and $\widehat{V}$ of the Hamiltonian. But, by expanding the exponential of each part, using a descendent of the so called Baker-Campbell-Hausdorff formula

$$e^{\widehat{A}}e^{\widehat{B}} = e^{\widehat{A}+\widehat{B}}e^{\frac{1}{2}[\widehat{A},\widehat{B}]}$$

we get that:

$$e^{-(\widehat{H}-E_{\mathrm{T}})t} = e^{-(\widehat{K}+\widehat{V}-E_{\mathrm{T}})t} = e^{-\widehat{K}t}e^{-(\widehat{V}-E_{\mathrm{T}})t} + \mathscr{O}(t^2)$$

It is also possible to make approximations to higher orders of t, but we will here keep to the simple first order case. As $t \to 0$, the error term disappears and the simple factorized form becomes exact. Thus, an approximation of the form

$$e^{-(\widehat{H}-E_{\mathrm{T}})t} \approx e^{-\widehat{K}t}e^{-(\widehat{V}-E_{\mathrm{T}})t}$$

is only good for small t and is readily called a *short time approximation*. The Green's function can by this be approximated as follows

$$\begin{aligned}
G(x,x',t) &= \langle x|e^{-(\widehat{H}-E_{\mathrm{T}})t}|x'\rangle \\
&= \langle x|e^{-\widehat{K}t}\,e^{-(\widehat{V}-E_{\mathrm{T}})t}|x'\rangle + \mathscr{O}(t^2) \\
&= \int \langle x|e^{-\widehat{K}t}|x''\rangle\,\langle x''|e^{-(\widehat{V}-E_{\mathrm{T}})t}|x'\rangle\,dx'' + \mathscr{O}(t^2) \\
&= \int G_K(x,x'',t)\,G_V(x'',x',t)\,dx'' + \mathscr{O}(t^2)
\end{aligned}$$

For the kinetic energy part, G_K, we get

$$\begin{aligned} G_K(x,x'',t) &= \langle x|e^{-\widehat{K}t}|x''\rangle \\ &= \frac{1}{(2\pi)^{3N}}\int \langle x|k\rangle e^{-Dk^2t}\langle k|x''\rangle\,dk \\ &= \frac{1}{(2\pi)^{3N}}\int e^{-ikx}e^{-Dk^2t}e^{ikx''}\,dk \\ &= \frac{1}{(4\pi Dt)^{3N/2}}\,e^{-(x''-x)^2/4Dt} \end{aligned}$$

The constant D is still the same diffusion constant in Eq. (17.4). The part related to the potential, G_V, is simpler, since it contains a local operator only dependent on x

$$G_V(x'',x',t) = e^{-(V(x')-E_{\mathrm{T}})t}\,\delta(x'-x'')$$

Putting these two parts together and carrying out the integral over x'' is simplified by the delta function of G_V. Ignoring the normalization constant of G_K we get

$$G(x,x',t) = e^{-(x'-x)^2/4Dt}\,e^{(E_{\mathrm{T}}-V(x'))t} + \mathscr{O}(t^2) \tag{17.6}$$

At this point, the GFMC method would pursue the problem by explicitly evaluating the Green's function integral of Eq. (17.5) with a suitable approximation. We, on the other hand, are now ready to interpret the diffusion process controlled by Eq. (17.3) in terms of the short time approximated Green's function. Our wave function $\Psi(x)$ is represented by a set of random walkers, so the integral over $G(x,x',t)\Psi(x')$ expresses the probability of a walker ending up at x given the initial configuration of walkers $\Psi(x')$. Calculating the integral corresponds to one iteration of the DMC algorithm (to be described in detail later on). Operationally this has a different effect for each of the two exponential factors of G.

The first factor expresses the probability for a walker to move from position x to x'. Since this clearly is just a Gaussian of x around x', we can simply generate new positions as a simple diffusion process

$$x = x' + \chi$$

where χ is a Gaussian pseudo-random number with mean equal zero and variance equal $2Dt$. The second factor, the branching term, is only dependent on x' and can be interpreted as the rate of growth of random walkers at each position x'. The short time approximation permits us to conduct the two processes, diffusion and branching separately, as long as the time step t is kept small.

Even though this approach, after a large enough number of iterations, should yield a distribution of walkers corresponding to the exact ground state of $\widehat{H}$, the method is not efficient. In particular, a serious problem arises when our system is governed by an unbounded potential, like a Coulomb potential typical for interactions between electrons or a parametrized nucleon-nucleon central potential with a hard central core. The branching exponential may diverge giving a huge production of new random walkers that may be difficult to handle numerically. Also the fluctuations become very large making statistical estimates of physical quantities inaccurate.

Another problem is that there is actually no simple way of calculating a mean energy estimate from the set of walkers alone. The mean energy is not only the most essential result of the algorithm, we also need it to be able to adjust the trial energy E_{T} dynamically throughout the DMC calculation.

For these reasons, and others that will become apparent, we introduce so called importance sampling by biasing the combined branching-diffusion process with a trial wave function Ψ_{T} which hopefully imitates the exact solution well. The technical contents of this will be made clear shortly.

17.1.1 Importance Sampling

Let us introduce a time independent trial wave function $\Psi_{\mathrm{T}}(x)$. We define the new quantity

$$f(x,t) \equiv \Psi_{\mathrm{T}}(x)\,\Psi(x,t)$$

Now by inserting $\Psi = f/\Psi_{\mathrm{T}}$ into Eq. (17.3) we get a slightly more complicated equation in terms of $f(x,t)$

$$\frac{\partial f(x,t)}{\partial t} = D\nabla^2 f(x,t) - D\nabla(F(x)f(x,t)) - (E_{\mathrm{L}}(x) - E_{\mathrm{T}})f(x,t) \tag{17.7}$$

with the constant D defined as in Eq. (17.4). Notice that the rhs. consists of three terms. By the same line of thought as before we now recognize the first term as the familiar diffusion term, acting on f instead of Ψ. The last term, similar in form to the potential term, is our new branching term, also acting on f. The quantity $E_{\mathrm{L}}(x)$ is just the local energy with respect to Ψ_{T}, defined in the same manner as in the variational Monte Carlo procedure

$$E_{\mathrm{L}} \equiv \frac{1}{\Psi_{\mathrm{T}}}\widehat{H}\Psi_{\mathrm{T}}$$

The vector quantity $F(x)$ in the unfamiliar middle term is the drift velocity as we know it from the Fokker-Planck formalism we used to improve the Metropolis algorithm

$$F(x) \equiv \frac{2}{\Psi_{\mathrm{T}}}\nabla\Psi_{\mathrm{T}} \tag{17.8}$$

Actually, the two first terms on the rhs. of Eq. (17.7) can equivalently be expressed as the familiar Fokker-Planck drift-diffusion on the rhs. of Eq. (14.26).

Eq. (17.7) motivates us to let the set of random walkers represent f instead of Ψ. A typical first order short time approximation of the corresponding Green's function is

$$\begin{aligned} G(x,x',t) &= \frac{1}{(4\pi Dt)^{3N/2}}\, e^{-(x-x'-DtF(x'))^2/4Dt}\; e^{-((E_{\mathrm{L}}(x)+E_{\mathrm{L}}(x'))/2-E_{\mathrm{T}})t} \\ &\quad + \mathscr{O}(t^2) \end{aligned} \tag{17.9}$$

We see that the Green's function above consists of two factors. The first one is similar to the Gaussian in Eq. (17.6). But now we have in addition a drift term displacing the mean of the Gaussian by $DtF(x')$. Notice that this factor of the Green's function is practically identical to the transition proposition rule introduced by the Fokker-Planck formalism to improve the Metropolis algorithm (see Eq. (14.25)). We can therefore use the same drift-diffusion formalism for the first factor in the above Green's function. Recall that if a walker is initially at position x', the new position x is calculated as follows (see Eq. (14.24))

$$x = x' + \chi + DF(x')t \tag{17.10}$$

where χ is again a Gaussian pseudo-random number with mean equal zero and variance equal $2Dt$ while $DF(x')t$ gives a drift in the direction that Ψ_{T} increases. The size of the time step t biased the final outcome of the Fokker-Planck algorithm of the drifted diffusion. A desirable diffusion was reached only as $t \to 0$ for each iteration. Also in the present application to DMC, this bias must be taken into account. In the Metropolis algorithm, the rejection mechanism took care of it. So we may use the same approach here. After calculating a new position with Eq. (17.10), we accept it according to the acceptance matrix

$$A(x,x',t) = \min\left[1, \frac{G_K(x',x)}{G_K(x,x')}\frac{|\Psi_{\rm T}(x)|^2}{|\Psi_{\rm T}(x')|^2}\right] \tag{17.11}$$

where G_K is the kinetic part of the Green's function, the part related to the diffusion process. If we do not wish to do such a Metropolis test, we may alternatively conduct separate calculations for a set of different time steps t and extrapolate the results to $t = 0$.

The second factor of Eq. (17.9) plays the same role as the branching term of Eq. (17.6). But the potential V is replaced by an expression dependent on the local energy,

$$\frac{E_{\rm L}(x) + E_{\rm L}(x')}{2} - E_{\rm T}$$

The new branching term gives a greatly reduced branching effect compared to the one in Eq. (17.6). Particularly, in the limit of $\Psi_{\rm T} = \phi_0$ and $E_{\rm T} = \varepsilon_0$ (exactly equal the ground state of $\widehat{H}$), the local energy is constant, giving equal branching everywhere, or in effect, no branching at all. Thus we can in general expect the number of walkers to fluctuate less and we certainly avoid uncontrollable growth. In addition, the branching favors the areas of the configuration space that give the lowest local energy, i.e. the local energy closest to the true ground state energy. Furthermore, the drifted diffusion pushes the walkers towards the desirable areas. Thus the whole DMC process is conducted more efficiently.

Finally, the introduction of the trial wave function $\Psi_{\rm T}$ makes it possible to evaluate an estimate of the mean energy given the distribution of the walkers. Instead of calculating the typical mean energy

$$\frac{\int \Psi^* \widehat{H} \Psi dx}{\int \Psi^* \Psi dx}$$

we calculate the so called *mixed estimator*

$$\langle E \rangle_{\rm mixed} = \frac{\int \Psi_{\rm T} \widehat{H} \Psi dx}{\int \Psi_{\rm T} \Psi dx} \tag{17.12}$$

As the DMC method approaches the exact result $\Psi = \phi_0$, the mixed estimator becomes

$$\langle E \rangle_{\rm mixed} = \frac{\int \Psi_{\rm T}\, \varepsilon_0\, \Psi dx}{\int \Psi_{\rm T} \Psi dx} = \frac{\varepsilon_0 \int \Psi_{\rm T} \Psi dx}{\int \Psi_{\rm T} \Psi dx} = \varepsilon_0$$

so that the estimate indeed becomes correct. Because of the hermiticity of $\widehat{H}$ we can rewrite the mixed estimator of Eq. (17.12) as follows

$$\langle E \rangle_{\rm mixed} = \frac{\int \Psi \widehat{H} \Psi_{\rm T} dx}{\int \Psi_{\rm T} \Psi dx} = \frac{\int \Psi_{\rm T} \Psi \frac{1}{\Psi_{\rm T}} \widehat{H} \Psi_{\rm T}\, dx}{\int \Psi_{\rm T} \Psi dx} = \frac{\int E_{\rm L} f(x) dx}{\int f(x) dx} \tag{17.13}$$

Since the walkers represent f we just need to average the local energy $E_{\rm L}$ over the set of walkers.

This energy estimator allows us to calculate relatively easily the mean energy on the distribution of walkers so that we can update the trial energy $E_{\rm T}$ dynamically as the algorithm proceeds. As the trial energy gets better and better, the algorithm will hopefully stabilize on the exact result within the limits of statistical fluctuations imposed by the local energy and our choice of $\Psi_{\rm T}$.

As we know, a good choice of $\Psi_{\rm T}$ reduces the fluctuations of the local energy $E_{\rm L}$. Now we see that this also makes the estimation of the mean energy in Eq. (17.13) more efficient since the same number of points (walkers) gives a smaller variance, thus pinpointing the energy more exactly.

Importance sampling makes it also to some extent easier to deal with wave functions that are not positive definite, like fermionic states whose wave function have nodes. What happens is that the nodes of the function f are tied by the nodes of the trial wave function Ψ_T. Therefore it is necessary for Ψ_T to have a node configuration that best reproduces the physical properties of the exact wave function. Operationally, the node surfaces of Ψ_T become impenetrable walls to the walkers in the sense that the drift velocity F in the vicinity of such a surface increases in a direction away from it pushing any walker away, preventing it from crossing the nodal surfaces of Ψ_T. The approach is called a *fixed node approximation*.

From all these deliberations on importance sampling we should by now understand the importance of the trial wave function being as close to the exact solution as possible. We therefore rely heavily on simpler methods like VMC that do not necessarily solve the problem exactly, but are easier to handle computationally and are much less sensitive to initial conditions.

17.2 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 [130–132] 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. [133].

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. [130]). 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 $85 < a_{Rb} < 140 a_0$ where $a_0 = 0.5292$ Å is the Bohr radius. The definite value $a_{Rb} = 100 a_0$ 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 [133]. 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 [134], 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 [135, 136] for the condensate describes the whole gas well. Effects of atoms excited above the condensate have been incorporated within the Popov approximation [137].

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 [135, 136]. 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. [133] 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 [138].

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 [139] of the energy density

$$\frac{E}{V} = \frac{2\pi n^2 a\hbar^2}{m}\left[1+\frac{128}{15}\left(\frac{na^3}{\pi}\right)^{1/2}+\cdots\right], \tag{17.14}$$

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 [140] and Hyper-Netted-Chain techniques [141].

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. (17.14)

$$E_{\mathrm{GP}}[\Psi] = \int d\mathbf{r}\left[\frac{\hbar^2}{2m}\mid\nabla\Psi(\mathbf{r})\mid^2 + V_{\mathrm{trap}}(\mathbf{r})\mid\Psi\mid^2 + \frac{2\pi\hbar^2 a}{m}\mid\Psi\mid^4\right], \tag{17.15}$$

where

$$V_{\mathrm{trap}}(\mathbf{r}) = \frac{1}{2}m(\omega_\perp^2 x^2 + \omega_\perp^2 y^2 + \omega_z^2 z^2) \tag{17.16}$$

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 $E_{\mathrm{GP}}[\Psi]$ with respect to Ψ^* one finds the corresponding Euler-Lagrange equation, known as the Gross-Pitaevskii (GP) equation

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V_{\mathrm{trap}}(\mathbf{r}) + \frac{4\pi\hbar^2 a}{m}\mid\Psi\mid^2\right]\Psi = \mu\Psi, \tag{17.17}$$

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. (17.15) the next term of the low density expansion of Eq. (17.14). The functional variation gives then rise to the so-called modified GP equation (MGP) [142]

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V_{\mathrm{trap}}(\mathbf{r}) + \frac{4\pi\hbar^2 a}{m}\mid\Psi\mid^2\left(1+\frac{32a^{3/2}}{3\pi^{1/2}}\mid\Psi\mid\right)\right]\Psi = \mu\Psi. \tag{17.18}$$

The MGP corrections have been estimated in Ref. [142] 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 [138]. Fully microscopic calculations using a hard-spheres interaction have also been performed in the framework of Variational and Diffusion Monte Carlo methods [143–146].

17.3 Exercises

17.1. 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. [94] 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

$$V_{ext}(\mathbf{r}) = \begin{cases} \frac{1}{2}m\omega_{ho}^2 r^2 & (S) \\ \frac{1}{2}m[\omega_{ho}^2(x^2+y^2)+\omega_z^2 z^2] & (E) \end{cases} \tag{17.19}$$

where (S) stands for symmetric and

$$H = \sum_i^N \left(\frac{-\hbar^2}{2m}\bigtriangledown_i^2 + V_{ext}(\mathbf{r}_i)\right) + \sum_{i<j}^N V_{int}(\mathbf{r}_i,\mathbf{r}_j), \tag{17.20}$$

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 (17.19) 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

$$V_{int}(|\mathbf{r}_i-\mathbf{r}_j|) = \begin{cases} \infty & |\mathbf{r}_i-\mathbf{r}_j| \le a \\ 0 & |\mathbf{r}_i-\mathbf{r}_j| > a \end{cases} \tag{17.21}$$

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

$$\Psi_T(\mathbf{R}) = \Psi_T(\mathbf{r}_1,\mathbf{r}_2,\dots\mathbf{r}_N,\alpha,\beta) = \prod_i g(\alpha,\beta,\mathbf{r}_i)\prod_{i<j} f(a,|\mathbf{r}_i-\mathbf{r}_j|), \tag{17.22}$$

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.,

$$g(\alpha,\beta,\mathbf{r}_i) = \exp[-\alpha(x_i^2+y_i^2+\beta z_i^2)]. \tag{17.23}$$

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

$$f(a,|\mathbf{r}_i-\mathbf{r}_j|) = \begin{cases} 0 & |\mathbf{r}_i-\mathbf{r}_j| \le a \\ (1-\frac{a}{|\mathbf{r}_i-\mathbf{r}_j|}) & |\mathbf{r}_i-\mathbf{r}_j| > a. \end{cases} \tag{17.24}$$

a) Find analytic expressions for the local energy

$$E_L(\mathbf{R}) = \frac{1}{\Psi_T(\mathbf{R})}H\Psi_T(\mathbf{R}), \tag{17.25}$$

for the above trial wave function of Eq. (17.22). Compute also the analytic expression for the drift force to be used in importance sampling

$$F = \frac{2\nabla\Psi_T}{\Psi_T}. \tag{17.26}$$

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. (17.22). 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. [143, 147] $a/a_{ho} = 0.0043$. 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

$$H = \sum_{i=1}^N \frac{1}{2}\left(-\nabla_i^2 + x_i^2 + y_i^2 + \gamma^2 z_i^2\right) + \sum_{i<j}V_{int}(|\mathbf{r}_i-\mathbf{r}_j|). \tag{17.27}$$

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. (17.22). 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. [143, 147].

References

1. J. Dongarra, F. Sullivan, Computing in Science and Engineering **2**, 22 (2000)
2. B. Cipra, SIAM News **33**, 1 (2000)
3. J. Thijssen, *Computational Physics* (Cambridge, 1999)
4. S. Koonin, D. Meredith, *Computational Physics* (Addison Wesley, 1990)
5. J. Gibbs, *Computational Physics* (World Scientific, 1994)
6. B. Giordano, H. Nakanishi, *Computational Physics* (Preston, 2005)
7. R. Landau, M. Paez, *Computational Physics* (Wiley, 1997)
8. R. Guardiola, E. Higon, J. Ros, *Metodes Numèrics per a la Física* (Universitat de Valencia, 1997)
9. E. Schmid, G. Spitz, W. Lösch, *Theoretische Physik mit dem Personal Computer* (Springer Verlag, 1987)
10. H. Gould, J. Tobochnik, *An Introduction to Computer Simulation Methods: Applications to Physical Systems* (Addison-Wesley, 1996)
11. B. Smith, J. Adams, W. Brainerd, J. Wagener, *Fortran 95 Handbook* (MIT press, 1997)
12. M. Metcalf, J. Reid, *The F90 Programming Language* (Oxford University Press, 1996)
13. A. Marshall, *Fortran 90 Programming* (University of Liverpool, 1995)
14. J. Reid, The new features of fortran 2003. Tech. rep., ISO working group on Fortran (2007). URL `http://www.iso.org/iso/home.htm`
15. W. Gropp, E. Lusk, A. Skjellum, *Using MPI* (The MIT Press, 1999)
16. M. Snir, S. Otto, S. Huss-Ledermann, D. Walker, J. Dongarra, *MPI, the Complete Reference, Vols I and II* (The MIT Press, 1998)
17. G.E. Karniadakis, R.M. Kirby II, *Parallel scientific computing in C++ and MPI* (Cambridge, 2005)
18. B. Flowers, *An Introduction to Numerical Methods in C++* (Oxford University Press, 2000)
19. J. Barton, L. Nackman, *Scientific and Engineering C++* (Addison Wesley, 1994)
20. B. Stoustrup, *The C++ Programming Language* (Pearson, 1997)
21. H. Langtangen, *Python Scripting for Computational Science* (Springer, 2006)
22. H. Langtangen, *Introduction to Computer Programming; A Python-based approach for Computational Science* (Springer Verlag, 2009)
23. D. Kincaid, W. Cheney, *Numerical Analysis* (Brooks/Gole Publishing Company, 1996)
24. R. Kress, *Numerical Analysis* (Springer, 1998)
25. J. Stoer, R. Bulirsch, *Introduction to Numerical Analysis* (Springer Verlag, 1983)
26. *LAPACK – Linear Algebra PACKage* (`http://www.netlib.org/lapack/`)
27. C.L. Lawson, R.J. Hanson, D. Kincaid, F.T. Krogh, ACM Trans. Math. Soft. **5**, 308 (1979)
28. G. Golub, C. Van Loan, *Matrix Computations* (John Hopkins University Press, 1996)
29. B. Datta, *Numerical Linear Algebra and Applications* (Brooks/Cole Publishing Company, 1995)
30. L. Trefethen, D. Bau III, *Numerical Linear Algebra* (SIAM Publications, 1997)
31. J. Demmel, *Numerical Linear Algebra* (SIAM Publications, 1996)
32. T. Veldhuizen, *Blitz++ User's Guide* (`http://www.oonumerics.org/blitz/`, 2003)
33. K. Reek, *Pointers on C* (Addison Wesley, 1998)
34. J. 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. Press, B. Flannery, S. Teukolsky, W. Vetterling, *Numerical Recipes in C++, The art of scientific Computing* (Cambridge University Press, 1999)
37. L.H. Thomas, Proc. Camb. Phil. Soc. **23**, 542 (1927)
38. E. Fermi, Rend. Accad. Naz. Lincei **6**, 602 (1927)
39. E.H. Lieb, Rev. Mod. Phys. **53**(4), 603 (1981). DOI 10.1103/RevModPhys.53.603
40. H.R. Pagels, *Cosmic Code: Quantum Physics as the Law of Nature* (Simon and Schuster, 1982)
41. K. Eriksson, D. Estep, P. Hansbo, C. Johnson, *Computational Differential Equations* (Cambridge University Press, 1996)
42. J. Butcher, *Numerical Methods for Ordinary Differential equations* (Wiley, 2008)
43. E. Hairer, S. NÃÿrsett, G. Wanner, *Solving Ordinary Differential Equations I* (Springer Verlag, Berlin, 1987)
44. D.C. Rapaport, *The Art of Molecular Dynamics Simulation* (Cambridge University Press, 2007)
45. D. Marx, J. Hutter, *Ab Initio Molecular Dynamics* (Cambridge University Press, 2010)
46. S.L. Shapiro, S.A. Teukolsky, *Black holes, white dwarfs, and neutron stars: the physics of compact objects* (Wiley, 1983)
47. H. Heiselberg, M. Hjorth-Jensen, Phys. Rep. **328**, 237 (2000). DOI doi:10.1016/S0370-1573(99)00110-6
48. N.K. Glendenning, *Compact Stars* (Springer, 2000)
49. H. Bethe, M. Johnson, Nucl. Phys. A **230**, 1 (1974)
50. H. Langtangen, *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming* (Springer, 1999)

51. J. Ortega, W. Poole, *An Introduction to Numerical Methods for Differential Equations* (Pitman, 1981)
52. A. Tveito, R. Winther, *Introduction to Partial Differential Equations* (Springer, 2002)
53. G. Arfken, *Mathematical Methods for Physicists* (Academic Press, 1985)
54. L. Evans, *Partial differential equations*, *Graduate Studies in Mathematics*, vol. 19 (American Mathematical Society, 2002)
55. L. Ramdas Ram-Mohan, *Finite Element and Boundary Element Applications in Quantm Mechanics* (Oxford University Press, 2002)
56. G. Evans, J. Blackledge, P. Yardley, *Numerical methods for partial differential equations* (Springer, 1999)
57. M. Plischke, B. Bergersen, *Equilibrium Statistical Physics* (Prentice-Hall, 1989)
58. P. Glasserman, *Monte Carlo Methods in Financial Engineering* (Springer Verlag, 2004)
59. P. Jackel, *Monte Carlo Methods in Finance* (John Wiley and Sons, LTD, 2002)
60. J. Voit, *The Statistical Mechanics of Financial Markets* (Springer Verlag, 2005)
61. J.L. McCauley, *Dynamics of Markets, Econophysics and Finance* (Cambridge University Press, 2004)
62. D. Sornette, *Why Stock Markets Crash* (Princeton University Press, 2002)
63. C. Robert, G. Casella, *Monte Carlo Statistical Methods* (Springer Verlag, 2004)
64. J. Johnson, *Probability and Statistics for Computer Science* (Wiley-Interscience, 2003)
65. G. Fishman, *Monte Carlo, Concepts, Algorithms and Applications* (Springer, 1996)
66. J.A. Rice, *Mathematical Statistics and Data Analysis* (Duxbury Advanced, 1994)
67. H. Flyvbjerg, H.G. Petersen, The Journal of Chemical Physics **91**(1), 461 (1989). DOI 10.1063/1.457480. URL `http://link.aip.org/link/?JCP/91/461/1`
68. H. Marzaglia, B. Zaman, Computers in Physics **8**, 117 (1994)
69. I. Karatsas, S. Shreve, *Brownian Motion and Stochastic Calculus* (Springer, 1988)
70. M. Chaichian, A. Demichev, *Path Integrals in Physics, Volume 1* (Institute of Physics Publishing, 2001)
71. L. Onsager, Phys. Rev. **31**, 405 (1931)
72. M. Patriarca, A. Chakraborti, K. Kaski, Physica A **340**, 334 (2004). DOI doi:10.1016/j.physa.2004.04.024
73. L. Onsager, Phys. Rev. **65**(3-4), 117 (1944). DOI 10.1103/PhysRev.65.117
74. N. Ashcroft, N. Mermin, *Solid State Physics* (Holt-Saunders, 1976)
75. J. Cardy, *Scaling and Renormalization in Statistical Physics* (Cambridge University Press, 1996)
76. K.G. Wilson, Rev. Mod. Phys. **47**, 773 (1975). DOI 10.1103/RevModPhys.47.773
77. K.G. Wilson, Rev. Mod. Phys. **55**, 583 (1983). DOI 10.1103/RevModPhys.55.583
78. H.E. Stanley, Rev. Mod. Phys. **71**(2), S358 (1999). DOI 10.1103/RevModPhys.71.S358
79. M. Newman, G. Barkema, *Monte Carlo Methods in Statistical Physics* (Clarendon Press, 1999)
80. D. Landau, K. Binder, *A Guide to Monte Carlo Simulations in Statistical Physics* (Cambridge University Press, 2000)
81. R. Shankar, Rev. Mod. Phys. **66**(1), 129 (1994). DOI 10.1103/RevModPhys.66.129
82. H. Stanley, *Introduction to Phase Transitions and Critical Phenomena* (Oxford University Press, 1971)
83. N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, The Journal of Chemical Physics **21**(6), 1087 (1953). URL `http://link.aip.org/link/?JCP/21/1087/1`
84. A.M. Ferrenberg, R.H. Swendsen, Phys. Rev. Lett. **61**, 2635 (1988). DOI 10.1103/PhysRevLett.61.2635
85. A.M. Ferrenberg, R.H. Swendsen, Phys. Rev. Lett. **63**, 1195 (1989). DOI 10.1103/PhysRevLett.63.1195
86. M.P. Nightingale, H.W.J. Blöte, Phys. Rev. Lett. **76**(24), 4548 (1996). DOI 10.1103/PhysRevLett.76.4548
87. U. Wolff, Phys. Rev. Lett. **62**(4), 361 (1989). DOI 10.1103/PhysRevLett.62.361
88. R.H. Swendsen, J.S. Wang, Phys. Rev. Lett. **58**, 86 (1987). DOI 10.1103/PhysRevLett.58.86
89. J.L. Monroe, Phys. Rev. E **66**(6), 066129 (2002). DOI 10.1103/PhysRevE.66.066129
90. M.S.S. Challa, D.P. Landau, K. Binder, Phys. Rev. B **34**(3), 1841 (1986). DOI 10.1103/PhysRevB.34.1841
91. F.Y. Wu, Rev. Mod. Phys. **54**(1), 235 (1982). DOI 10.1103/RevModPhys.54.235
92. K. Binder, Reports on Progress in Physics **50**(7), 783 (1987). URL `http://stacks.iop.org/0034-4885/50/783`
93. R. Liboff, *Introductory Quantum Mechanics* (Addison Wesley, 2003)
94. B.L. Hammond, W.A. Lester, P. Reynolds, *Monte Carlo Methods in ab Initio Quantum Chemistry* (World Scientific, 1994)
95. H. Kümmel, K.H. Luhrmann, J.G. Zabolitzky, Phys. Rep. **36**, 1 (1978, and references therein)
96. R.J. Bartlett, M. Musiał, Reviews of Modern Physics **79**(1), 291 (2007). DOI 10.1103/RevModPhys.79.291. URL `http://link.aps.org/abstract/RMP/v79/p291`
97. T. Helgaker, P. Jørgensen, J. Olsen, *Molecular Electronic Structure Theory. Energy and Wave Functions* (Wiley, Chichester, 2000)
98. D.J. Dean, M. Hjorth-Jensen, Physical Review C **69**, 054320 (2004). URL `http://link.aps.org/abstract/PRC/v69/e054320`
99. K. Kowalski, D.J. Dean, M. Hjorth-Jensen, T. Papenbrock, P. Piecuch, Phys. Rev. Lett. **92**, 132501 (2004). DOI 10.1103/PhysRevLett.92.132501. URL `http://link.aps.org/doi/10.1103/PhysRevLett.92.132501`

100. B.S. Pudliner, V.R. Pandharipande, J. Carlson, S.C. Pieper, R.B. Wiringa, Phys. Rev. C **56**, 1720 (1997). DOI 10.1103/PhysRevC.56.1720
101. S. Koonin, D. Dean, K. Langanke, Phys. Rep. **278**, 1 (1997). DOI doi:10.1016/S0370-1573(96)00017-8
102. D.M. Ceperley, Rev. Mod. Phys. **67**, 279 (1995). URL `http://link.aps.org/abstract/RMP/v67/p279`
103. P.J. Ellis, E. Osnes, Rev. Mod. Phys. **49**, 777 (1977). URL `http://link.aps.org/abstract/RMP/v49/p777`
104. I. Lindgren, J. Morrison, *Atomic Many-Body Theory* (Springer, Berlin, 1985)
105. M. Hjorth-Jensen, T.T.S. Kuo, E. Osnes, Phys. Rep. **261**, 125 (1995). DOI doi:10.1016/0370-1573(95)00012-6
106. W.H. Dickhoff, D.V. Neck, *Many-Body Theory exposed!* (World Scientific, 2005)
107. J.P. Blaizot, G. Ripka, *Quantum theory of finite systems* (MIT press, Cambridge, USA, 1986)
108. S.R. White, Phys. Rev. Lett. **69**, 2863 (1992). DOI 10.1103/PhysRevLett.69.2863. URL `http://link.aps.org/abstract/PRL/v69/p2863`
109. U. Schollwock, Rev. Mod. Phys. **77**, 259 (2005). URL `http://link.aps.org/abstract/RMP/v77/p259`
110. R.O. Jones, O. Gunnarsson, Rev. Mod. Phys. **61**(3), 689 (1989). DOI 10.1103/RevModPhys.61.689
111. R.J. Bartlett, V.F. Lotrich, I.V. Schweigert, J. Chem. Phys. **123**, 062205 (2005). URL `http://link.aip.org/link/?JCP/123/062205/1`
112. K. Peirs, D.V. Neck, M. Waroquier, Physical Review A (Atomic, Molecular, and Optical Physics) **67**, 012505 (2003). URL `http://link.aps.org/abstract/PRA/v67/e012505`
113. D.V. Neck, S. Verdonck, G. Bonny, P.W. Ayers, M. Waroquier, Physical Review A (Atomic, Molecular, and Optical Physics) **74**, 042501 (2006). URL `http://link.aps.org/abstract/PRA/v74/e042501`
114. R.R. Whitehead, A. Watt, B.J. Cole, I. Morrison, Adv.Nucl.Phys. **9**, 123 (1977)
115. E. Caurier, G. Martinez-Pinedo, F. Nowacki, A. Poves, A.P. Zuker, Rev. Mod. Phys. **77**, 427 (2005). URL `http://link.aps.org/abstract/RMP/v77/p427`
116. M. Horoi, B.A. Brown, T. Otsuka, M. Honma, T. Mizusaki, Phys. Rev. C **73**, 061305 (2006). URL `http://link.aps.org/abstract/PRC/v73/e061305`
117. D.J. Dean, M. Hjorth-Jensen, *Computational Quantum Mechanics for Nuclear Physics* (Chapman & Hall/CRC, 2009)
118. B.H. Bransden, C.J. Joachain, *Physics of Atoms and Molecules* (Longman, 1983)
119. J.W. Moskowitz, M.H. Kalos, Int. J. Quantum Chem. **20**, 1107 (1981)
120. J.W. Moskowitz, K.E. Schmidt, M.A. Lee, M.H. Kalos, The Journal of Chemical Physics **76**(2), 1064 (1982). DOI 10.1063/1.443098. URL `http://link.aip.org/link/?JCP/76/1064/1`
121. P.W. Atkins, R.S. Freidman, *Molecular Quantum Mechanics* (Oxford Univerity Press, 2003)
122. E. Clementi, D.L. Raimondi, J. Chem. Phys. **38**, 2686 (1963)
123. P. Hohenberg, W. Kohn, Phys. Rev. **136**(3B), B864 (1964). DOI 10.1103/PhysRev.136.B864
124. W. Kohn, L.J. Sham, Phys. Rev. **140**(4A), A1133 (1965). DOI 10.1103/PhysRev.140.A1133
125. J.P. Perdew, A. Zunger, Phys. Rev. B **23**(10), 5048 (1981). DOI 10.1103/PhysRevB.23.5048
126. J.P. Perdew, Y. Wang, Phys. Rev. B **45**(23), 13244 (1992). DOI 10.1103/PhysRevB.45.13244
127. J.P. Perdew, J.A. Chevary, S.H. Vosko, K.A. Jackson, M.R. Pederson, D.J. Singh, C. Fiolhais, Phys. Rev. B **46**(11), 6671 (1992). DOI 10.1103/PhysRevB.46.6671
128. A.J. Williamson, S.D. Kenny, G. Rajagopal, A.J. James, R.J. Needs, L.M. Fraser, W.M.C. Foulkes, P. Maccullum, Phys. Rev. B **53**(15), 9640 (1996). DOI 10.1103/PhysRevB.53.9640
129. D. Ceperley, G.V. Chester, M.H. Kalos, Phys. Rev. B **16**(7), 3081 (1977). DOI 10.1103/PhysRevB.16.3081
130. M.H. Anderson, J.R. Ensher, M.R. Matthews, C.E. Wieman, E.A. Cornell, Science **269**, 198 (1995)
131. K.B. Davis, M.O. Mewes, M.R. Andrews, N.J. van Druten, D.S. Durfee, D.M. Kurn, W. Ketterle, Phys. Rev. Lett. **75**(22), 3969 (1995). DOI 10.1103/PhysRevLett.75.3969
132. C.C. Bradley, C.A. Sackett, J.J. Tollett, R.G. Hulet, Phys. Rev. Lett. **75**(9), 1687 (1995). DOI 10.1103/PhysRevLett.75.1687
133. F. Dalfovo, S. Giorgini, L.P. Pitaevskii, S. Stringari, Rev. Mod. Phys. **71**, 463 (1999). DOI 10.1103/RevModPhys.71.463
134. N. Bogolyubov, Dolk. Akad. Nauk SSSR **119**, 244 (1959)
135. E. Gross, Nuovo Cimento **20**, 454 (1961)
136. L. Pitaevskii, Zh.EKSP.Teor.Fiz. **40**, 646 (1961)
137. D.A.W. Hutchinson, E. Zaremba, A. Griffin, Phys. Rev. Lett. **78**(10), 1842 (1997). DOI 10.1103/PhysRevLett.78.1842
138. S.L. Cornish, N.R. Claussen, J.L. Roberts, E.A. Cornell, C.E. Wieman, Phys. Rev. Lett. **85**(9), 1795 (2000). DOI 10.1103/PhysRevLett.85.1795
139. A.L. Fetter, J.D. Walecka, *Quantum Theory of Many-Particle Systems* (McGraw-Hill, 1971)
140. S. Giorgini, J. Boronat, J. Casulleras, Phys. Rev. A **60**(6), 5129 (1999). DOI 10.1103/PhysRevA.60.5129
141. F. Mazzanti, A. Polls, A. Fabrocini, Phys. Rev. A **67**(6), 063615 (2003). DOI 10.1103/PhysRevA.67.063615
142. A. Fabrocini, A. Polls, Phys. Rev. A **60**(3), 2319 (1999). DOI 10.1103/PhysRevA.60.2319
143. J.L. DuBois, H.R. Glyde, Phys. Rev. A **63**(2), 023602 (2001). DOI 10.1103/PhysRevA.63.023602

144. A.R. Sakhel, J.L. DuBois, H.R. Glyde, Phys. Rev. A **66**(6), 063610 (2002). DOI 10.1103/PhysRevA.66.063610

145. J.L. DuBois, H.R. Glyde, Phys. Rev. A **68**(3), 033602 (2003). DOI 10.1103/PhysRevA.68.033602

146. D. Blume, C.H. Greene, Phys. Rev. A **63**(6), 063601 (2001). DOI 10.1103/PhysRevA.63.063601

147. J.K. Nilsen, J. Mur-Petit, M. Guilleumas, M. Hjorth-Jensen, A. Polls, Physical Review A (Atomic, Molecular, and Optical Physics) **71**, 053610 (2005). URL `http://link.aps.org/abstract/PRA/v71/e053610`