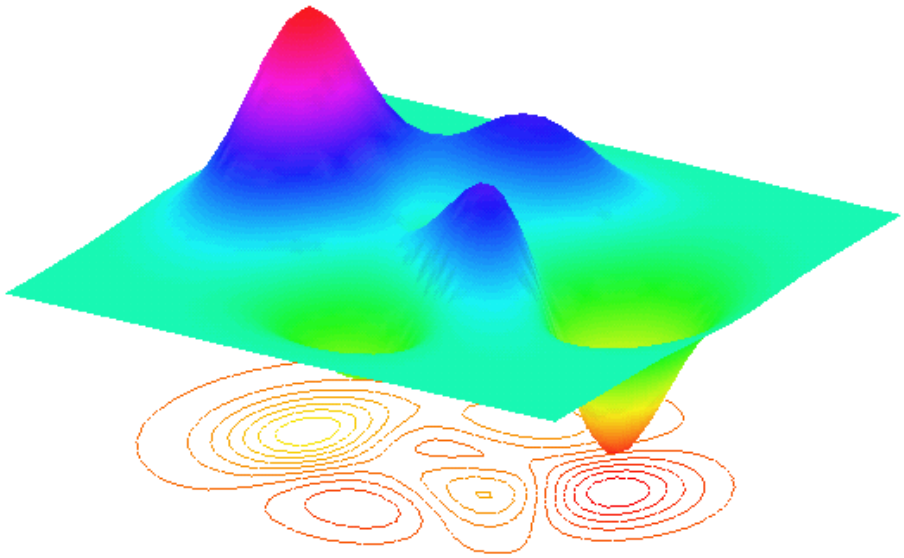
COMPUTATIONAL PHYSICS

M. Hjorth-Jensen



Department of Physics, University of Oslo, 2003

Preface

In 1999, when we started teaching 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 graduate students had a very rudimentary knowledge of computational techniques and methods. Four years later most students have had a fairly uniform introduction to computers, basic programming skills and use of numerical exercises in undergraduate courses. Practically every undergraduate student in physics has now made a Matlab or Maple simulation of e.g., the pendulum, with or without chaotic motion. These exercises 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 to complicated or expensive to carry out. Thus, computer simulations are nowadays an integral part of contemporary basic and applied research in the physical sciences. Computation is becoming as important as theory and experiment. We could even strengthen this statement by saying that computational physics, theoretical physics and experimental are all equally important in our daily research and studies of physical systems. Physics is nowadays the unity of theory, experiment and computation. The ability "to compute" is now part of the essential repertoire of research scientists. Several new fields 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. To be able to e.g., simulate quantal systems will be of great importance for future directions in fields like materials science and nanotechonology.

This ability combines knowledge from many different subjects, in our case essentially from the physical sciences, numerical analysis, computing languages and some knowledge of computers. These topics are, almost as a rule of thumb, taught in different, and we would like to add, disconnected courses. Only at the level of thesis work is the student confronted with the synthesis of all these subjects, and then in a bewildering and disparate manner, trying to e.g., understand old Fortran 77 codes inherited from his/her supervisor back in the good old ages, or even more archaic, programs. Hours may have elapsed in front of a screen which just says 'Underflow', or 'Bus error', etc etc, without fully understanding what goes on. Porting the program to another machine could even result in totally different results!

The first aim of this course is therefore to bridge the gap between undergraduate courses in the physical sciences and the applications of the aquired knowledge to a given project, be it either a thesis work or an industrial project. We expect you to have some basic knowledge in the physical sciences, especially within mathematics and physics through e.g., sophomore courses in basic calculus, linear algebraand general physics. Furthermore, having taken an introductory course on programming is something we recommend. As such, an optimal timing for taking this course, would be when you are close to embark on a thesis work, or if you've just started with a thesis. But obviously, you should feel free to choose your own timing.

We have several other aims as well in addition to prepare you for a thesis work, 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 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 1-2 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.

- The exam reflects this project-like philosophy. The exam itself is a project which lasts one month. You have to hand in a report on a specific problem, and your report forms the basis for an oral examination with a final grading.

Our overall goal is to encourage you to learn about science through experience and by asking questions. Our objective is always understanding, not the generation of numbers. The purpose of computing is further insight, not mere numbers! Moreover, and this is our personal bias, 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 of the problem.

Most of you are by now familiar, through various undergraduate courses in physics and mathematics, with interpreted languages such as Maple, Mathlab and 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 you need however a programme which is fairly fast when computational speed matters. In this case you would most likely write a high-performance computing programme in languages which are taylored for that. These are represented by programming languages like Fortran 90/95 and C/C++. However, to visualize the results you would find interpreted languages like e.g., Matlab or scripting languages like Python extremely suitable for your tasks. You will therefore end up writing e.g., a script in Matlab which calls a Fortran 90/95 ot C/C++ programme 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 90/95 or C/C++ programs and performs the visualization in Matlab as well.

Being multilingual is thus a feature which not only applies to modern society but to computing environments as well.

However, there is more to the picture than meets the eye. This course emphasizes the use of programming languages like Fortran 90/95 and C/C++ instead of interpreted ones like Matlab or Maple. Computational speed is not the only reason for this choice of programming languages. The main reason is that we feel 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 etc. Although we will at various stages recommend the use of library routines for say linear algebra[1], our belief is that one should understand what the given function does, at least to have a mere idea. From such a starting point we do further believe that it can be easier to develope more complicated programs, on your own. We do therefore devote 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'd 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

Needless to say, these lecture notes are upgraded continuously, from typos to new input. And we do always benifit from your comments, suggestions and ideas for making these notes better. It's through the scientific discourse and critics we advance.

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

Contents

Part I

Introduction to Computational Physics

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 few-body systems through Monte-Carlo methods. Several minor exercises of a more numerical character are scattered throughout the main text.

The topics we cover start with an introduction to C/C++ and Fortran 90/95 programming combining it with a discussion on numerical precision, a point we feel is often neglected in computational science. This chapter serves also as input to our discussion on numerical derivation in chapter 3. In that chapter we introduce several programming concepts such as dynamical memory allocation and call by reference and value. Several program examples are presented in this chapter. For those who choose to program in C/C++ we give also an introduction to the auxiliary library Blitz++, which contains several useful classes for numerical operations on vectors and matrices. The link to Blitz++, matrices and selected algorithms for linear algebra problems are dealt with in chapter 5. Chapters 6 and 7 deal with the solution of non-linear equations and the finding of roots of polynomials and numerical interpolation, extrapolation and data fitting.

Therafter we switch to numerical integration for integrals with few dimensions, typically less than 3, in chapter 8. The numerical integration chapter serves also to justify the introduction of Monte-Carlo methods discussed in chapters 9 and 10. 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

11 continues this discussion by extending to studies of phase transitions in statistical physics. Chapter 12 deals with Monte-Carlo studies of quantal systems, with an emphasis on variational Monte Carlo methods and diffusion Monte Carlo methods. In chapter 13 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 14-16 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 and modern languages such as Java, C/C++, Fortran 77/90/95, etc. Fortran[1] and C/C++ are examples of compiled high-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. [8, 4, **?**, **?**, 6, 9, 7, 10], ranging from introductory ones to more advanced ones. Most of these texts treat however in a rather cavalier way the mathematics behind the various numerical methods. We've also succumbed to this approach, mainly due to the following reasons: several of the methods discussed are rather involved, and would thus require at least a two-semester course for an introduction. In so doing, little time would be left for problems and computation. This course is a compromise between three disciplines, numerical methods, problems from the physical sciences and computation. To achieve such a synthesis, we will have to relax our presentation in order to avoid lengthy and gory mathematical expositions. You should also keep in mind that Computational Physics and Science in more general terms consist of the combination of several fields and crafts with the aim of finding solution strategies for complicated problems. However, where we do indulge in presenting more formalism, we have borrowed heavily from the text of Stoer and Bulirsch [**?**], a text we really recommend if you'd like to have more math to chew on.

1.1 Choice of programming language

As programming language we have ended up with preferring C/C++, but every chapter, except for the next, contains also in an appendix the corresponding Fortran 90/95 programs. Fortran (FORmula TRANslation) was introduced in 1957 and remains in many scientific computing environments the language of choice. The latest standard, Fortran 95 [**?**, 11, **?**], includes extensions that are familiar to users of C/C++. Some of the most important features of Fortran 90/95 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

[1]With Fortran we will consistently mean Fortran 90/95. There are no programming examples in Fortran 77 in this text.

reasons for choosing C/C++ as programming language for scientific and engineering problems. Here are some:

- C/C++ is now the dominating language in Unix and Windows environments. It is widely available and is the language of choice for system programmers.
- The C/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/C++ compiler.
- In the last years there has been an enormous effort towards developing numerical libraries for C/C++. Numerous tools (numerical libraries such as MPI[**?**]) are written in C/C++ and interfacing them requires knowledge of C/C++. Most C/C++ and Fortran 90/95 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 90/95, 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/C++ or F90/95, see however the article by Jung *et al.* for a discussion on numerical aspects of Java [**?**].
- Complex variables, one of Fortran 77 and 90/95 strongholds, can also be defined in the new ANSI C/C++ standard.
- C/C++ is a language which catches most of the errors as early as possible, typically at compilation time. Fortran 90/95 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 90/95. This means that it supports three fundamental ideas, namely objects, class hierarchies and polymorphism. Fortran 90/95 has, through the `MODULE` declaration the capability of defining classes, but lacks inheritance, although polymorphism is possible. Fortran 90/95 is then considered as an object-based programming language, to be contrasted with C/C++ which has the capability of relating classes to each other in a hierarchical way.

C/C++ is however 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 error is dynamic memory management.

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 debuging and trying to understand what the codes do. A clear program will also allow you to remember better what the program really does!
- The planning of the program should be from top down to bottom, trying to keep the flow as linear as possible. Avoid jumping back and forth in the program. First you need to arrange the major tasks to be achieved. Then try to break the major tasks into subtasks. These can be represented by functions or subprograms. They should accomplish limited tasks and as far as possible be independent of each other. That will allow you to use them in other programs as well.
- Try always to find some cases where an analytical solution exists or where simple test cases can be applied. If possible, devise different algorithms for solving the same problem. If you get the same answers, you may have coded things correctly or made the same error twice.

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

- Use always the standard ANSI version of the programming language. Avoid local dialects if you wish to port your code to other machines.
- Add always comments to describe what a program or subprogram does. Comment lines help you remember what you did e.g., one month ago.
- Declare all variables. Avoid totally the `IMPLICIT` statement in Fortran. The program will be more readable and help you find errors when compiling.

[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.

- 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. Also, the use of debuggers like **gdb** is something we highly recommend during the development of a program.

Chapter 2

Introduction to C/C++ and Fortran 90/95

2.1 Getting started

In all programming languages 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 e.g.,`INTEGER (KIND=2)` for Fortran 90/95[1] or `short int/int` in C/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 the following table lists the standard variable declarations of C/C++ and Fortran 90/95 (note well that there 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/C++ and Fortran 90/95 operate with several types of variables and the answers to these questions depend on how we have defined `int var`. The following list may help in clarifying the above points:

[1]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/C++ does.

type in C/C++ and Fortran 90/95	bits	range
char/CHARACTER	8	-128 to 127
unsigned char	8	0 to 255
signed char	8	-128 to 127
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	$3.4e^{-38}$ to $3.4e^{+38}$
double/REAL(8)	64	$1.7e^{-308}$ to $1.7e^{+308}$
long double	64	$1.7e^{-308}$ to $1.7e^{+308}$

Table 2.1: Examples of variable declarations for C/C++ and Fortran 90/95. We reserve capital letters for Fortran 90/95 declaration statements throughout this text, although Fortran 90/95 is not sensitive to upper or lowercase letters.

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/C++. In addition, both C++ and Fortran 90/95 allow for complex variables. In Fortran 90/95 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/C++ we would need to include a complex library through the statements

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

We will come back to these topics in later chapter.

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.

Fortran 90/95	C/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;
DOUBLE PRECISION :: (or REAL (8)) x, y	double x, y;
INTEGER :: x, y	int x,y;
CHARACTER :: name	char name;
DOUBLE PRECISION, 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.

programs/chap2/program1.cpp

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

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

The compiler must see a declaration of a function before you can call it (the compiler checks the argument and return types). The declaration of library functions appears in so-called header files that must be included in the program, e.g., **#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, int (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 **int** main (**int** argc , **char*** argv []) The integer argc is the no 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. Here we define floating points, see also below, through the keywords **float** for single precision real numbers and *double* for double precision. The function atof transforms a text (argv[1]) to a float. The sine function is declared in math.h, a library which is not automatically included and needs to be linked when computing an executable file.

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

In C++ this program can be written as

```
// A comment line begins like this in C++ programs
using namespace std;
#include <iostream>
int main (int argc, char* argv[])
{
//  convert the text argv[1] to double using atof:
  double r = atof(argv[1]);
  double s = sin(r);
  cout << "Hello, World! sin(" << r << ")=" << s << '\n';
// 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.

To run these programs, you need first to compile and link it 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 90/95 code is

programs/chap2/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 one can see statements like IMPLICIT REAL*8(a-h,o-z), meaning that all variables beginning with any of the above letters are by deafult 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.

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
```

2.1.1 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/C++ keywords are short int , int , long int , long long int . The byte-length is compiler dependent within some limits. The GNU C/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/C++ has a library function called sizeof (**type**) which returns the number of bytes for **type** .

An example of program declaration is

```
Fortran:    INTEGER (KIND=2) :: age_of_participant
C/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/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

$$417 = 4 \times 10^2 + 1 \times 10^1 + 7 \times 10^0, \tag{2.1}$$

which in binary representation becomes

$$417 = 1 \times a_n 2^n + a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_0 2^0, \tag{2.2}$$

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

$$a_n 2^n + a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_0 2^0. \tag{2.3}$$

In binary notation we have thus

$$(417)_{10} = (110100001)_2 =, \tag{2.4}$$

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 1	coefficient of 2^2 is 0
52/2=27	remainder 1	coefficient of 2^3 is 0
26/2=13	remainder 1	coefficient of 2^4 is 0
13/2= 6	remainder 1	coefficient of 2^5 is 1
6/2= 3	remainder 1	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

A simple program which performs these operations is listed below. Here we employ the modulus operation, which in C/C++ is given by the a%2 operator. In Fortran 90/95 the difference is that we call the function MOD(a,2).

programs/chap2/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/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. Another example, the number 3 is given in an 8 bits word as

$$3 = 0000011. \tag{2.5}$$

Note that for 417 we need 9 bits in order to represent the number wheras 3 needs only 2 significant bits.

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

$$2^{20} \times 2^{20} = 0. \tag{2.6}$$

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).

programs/chap2/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()
```

The corresponding Fortran 90/95 example is

programs/chap2/program2.f90

```
    PROGRAM integer_exp
    IMPLICIT NONE
    INTEGER (KIND=4) :: 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=int2**30
    WRITE(*,*) 'int2**30+int2**30', int1+int1
    int3=int1-1
    WRITE(*,*) 'int2**30+int2**30-1', int1+int3
    WRITE(*,*) 'int2**31-1', 2*int1-1

    END PROGRAM integer_exp
```

2.2 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 e.g., 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. You should always keep in mind that the machine can only represent a floating number to a given precision. Let us in the following example 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

$$f(x) = \frac{1 - cos(x)}{sin(x)}, \tag{2.7}$$

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

$$f(x) = \frac{sin(x)}{1 + cos(x)}. \tag{2.8}$$

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

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

and

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

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

$$f(x) = \frac{1 - 0.99998}{0.69999 \times 10^{-2}} = \frac{0.2 \times 10^{-4}}{0.69999 \times 10^{-2}} = 0.28572 \times 10^{-2}, \tag{2.9}$$

while the second expression results in

$$f(x) = \frac{0.69999 \times 10^{-2}}{1 + 0.99998} = \frac{0.69999 \times 10^{-2}}{1.99998} = 0.35000 \times 10^{-2}, \tag{2.10}$$

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.

2.2.1 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). In the decimal system we would write a number like 6.7894 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

$$6.7894 = 0.67894 \times 10^1,$$

and a real non-zero number could be generalized as

$$x = \pm r \times 10^{\rm n}, \tag{2.11}$$

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

$$x = \pm q \times 2^{\rm m}, \tag{2.12}$$

with a q a number in the range $1/2 \leq q < 1$.

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

$$x = (-1)^s \times \text{mantissa} \times 2^{\text{exponent}}, \tag{2.13}$$

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

```
Fortran:    REAL (4) :: size_of_fossile
C/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.

If our number x can be exactly represented in the machine, we call x a machine number. Unfortunately, most numbers cannot 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. This means in turn that for real numbers, we may have to deal with essentially four types of problems[2] . Let us list them and discuss how to discover these problems and their eventual cures.

[2]There are others, like errors made by the programmer, or problems with compilers.

1. **Overflow :** When the positive exponent exceeds the max value, e.g., 308 for DOUBLE PRECISION (64 bits). Under such circumstances the program will terminate and some compilers may give you the warning 'OVERFLOW'.

2. **Underflow :** When the negative exponent becomes smaller than the min value, e.g., -308 for DOUBLE PRECISION. Normally, the variable is then set to zero and the program continues. Other compilers (or compiler options) may warn you with the 'UNDERFLOW' message and the program terminates.

3. **Roundoff errors** A floating point number like

$$x = 1.2345678911121314 68 = 0.12345678911121314 68 \times 10^1 \tag{2.14}$$

may be stored in the following way. The exponent is small and is stored in full precision. However, the mantissa is not stored fully. In double precision (64 bits), digits beyond the 15th are lost since the mantissa is normally stored in two words, one which is the most significant one representing 123456 and the least significant one containing 789111213. The digits beyond 3 are lost. Clearly, if we are summing alternating series with large numbers, subtractions between two large numbers may lead to roundoff errors, since not all relevant digits are kept. This leads eventually to the next problem, namely

4. **Loss of precision** Overflow and underflow are normally among the easiest problems to deal with. When one has to e.g., multiply two large numbers where one suspects that the outcome may be beyond the bonds imposed by the variable declaration, one could represent the numbers by logarithms, or rewrite the equations to be solved in terms of dimensionless variables. When dealing with problems in e.g., particle physics or nuclear physics where distance is measured in fm (10^{-15}m), it can be quite convenient to redefine the variables for distance in terms of a dimensionless variable of the order of unity. To give an example, suppose you work with single precision and wish to perform the addition $1 + 10^{-8}$. In this case, the information containing in 10^{-8} is simply lost in the addition. Typically, when performing 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.

   However, the loss of precision and significance due to the way numbers are represented in the computer and the way mathematical operations are performed, can at the end lead to totally wrong results.

Other cases which may cause problems are singularities of the type $0/0$ which may arise from functions like $sin(x)/x$ as $x \to 0$. Such problems may need the restructuring of the algorithm.

In order to illustrate the above problems, we consider in this section three possible algorithms for computing e^{-x}:

1. by simply coding

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

2. or to employ a recursion relation for

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

   using

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

3. or to first calculate

$$\exp\left(x\right) = \sum_{n=0}^{\infty}s_n$$

   and thereafter taking the inverse

$$\exp\left(-x\right) = \frac{1}{\exp\left(x\right)}$$

Below we have included a small program which calculates

$$\exp\left(-x\right) = \sum_{n=0}^{\infty}(-1)^n\frac{x^n}{n!}, \tag{2.15}$$

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$\{\ldots\}$ statement.

programs/chap2/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}. In Fortran 90/95 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.

[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 $\exp(-x)$.

Fortran 90/95 uses a do construct to have the computer execute the same statements more than once. Note also that Fortran 90/95 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/C++ program. One could altrenatively use the EXIT statement inside a do loop. Fortran 90/95 has also if statements as in C/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 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/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 90/95 allows such a control. Furthermore, it increases the readability of the program.

programs/chap2/program3.f90

```
PROGRAM exp_prog
```

```
      IMPLICIT NONE
      REAL (KIND=8) :: x, term, final_sum, &
                       factorial, truncation
      INTEGER :: n, loop_over_x
      truncation=1.0E-10
!  loop over x-values
      DO loop_over_x=0, 100, 10
         x=loop_over_x
!  initialize the EXP sum
         final_sum=1.0 ; sum_term=1.0 ; exponent=0
         DO WHILE ( ABS(sum_term) > truncation)
            n=n+1
            term = ((-1.)**n)*(x**n)/ factorial(n)
            final_sum=final_sum+term
         ENDDO
!  write the argument x, the exact value, the computed value and n
         WRITE(*,*) argument,EXP(-x), final_sum, n
      ENDDO

      END PROGRAM exp_prog

      DOUBLE PRECISION FUNCTION factorial(n)
      INTEGER (KIND=2), INTENT(IN) :: n
      INTEGER (KIND =2 ) :: loop

      factorial = 1.
      IF ( n > 1 ) THEN
         DO loop = 2, n
            factorial=factorial*loop
         ENDDO
      ENDIF

      END FUNCTION factorial
```

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

$$\exp(-x) = \sum_{n=0}^{\infty} s_n = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!}, \tag{2.16}$$

is to note that

$$s_n = -s_{n-1}\frac{x}{n}, \tag{2.17}$$

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

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

programs/chap2/program5.cpp

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

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

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

In this case, we do not get the overflow problem, as can be seen from the large number of terms. Our results do however not make much sense for larger 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 $n = 19$ and 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. m. 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

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 $\exp(-x)$.

right end of the number. Unless we are very careful we will find ourselves adding up series that finally consists entirely of roundoff errors! 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.

The Fortran 90/95 program is rather similar in structure to the C/C++ progra

programs/chap2/program4.f90

```
      PROGRAM improved
      IMPLICIT NONE
      REAL (KIND=8) :: x, term, final_sum, truncation_test
      INTEGER (KIND=4)} :: n, loop_over_x
      truncation_test=1.0E-10
!  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 ; sum_term=1.0 ; exponent=0
         DO WHILE ( ABS(sum_term) > truncation_test )
            n=n+1
            term = -term*x/FLOAT(n)
            final_sum=final_sum+term
         ENDDO
!  write the argument x, the exact value, the computed value and n
         WRITE(*,*) argument,EXP(-x), final_sum, n
      ENDDO
```

```
END PROGRAM improved
```

2.2.2 Further examples

Summing $1/n$

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

$$s_1 = \sum_{n=1}^{N} \frac{1}{n}, \tag{2.18}$$

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

$$s_2 = \sum_{n=N}^{1} \frac{1}{n}, \tag{2.19}$$

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

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

The standard algorithm for the standard deviation

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

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

$$\overline{x} = \frac{\sum_{i=1}^{N} x(i)}{N}, \tag{2.20}$$

while

$$\sigma = \sqrt{\frac{\sum_i x(i)^2 - \overline{x} \sum_i x(i)}{N-1}}. \tag{2.21}$$

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 $\overline{x}$. Using single precision results in a standard deviation of $\sigma = 40.05720139$ for the most used algorithm, while the exact answer is $\sigma = 36.80579758$, a number which also results from the above two-step 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[5]

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

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

programs/chap2/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.;
```

[5]Note that this number may be compiler and machine dependent.

```
    //  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 cruder 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 90/95 program is given below.

programs/chap2/program5.f90

```
      PROGRAM standard_deviation
      IMPLICIT NONE
      REAL*4 :: sum, sumsq2, xbar
      REAL*4 :: sigma1, sigma2
      REAL*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 text book 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 method 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
```

2.3 Loss of precision

2.3.1 Machine numbers

How can we understand the above mentioned problems? First let us note that a real number x has a machine representation $fl(x)$

$$fl(x) = x(1+\epsilon) \tag{2.22}$$

where $|\epsilon| \leq \epsilon_M$ and ϵ is given by the specified precision, 10^{-7} for single and 10^{-16} for double precision, respectively. ϵ_M is the given precision. Suppose that we are dealing with a 32-bit word and deal with single precision real number. This means that the precision is at the 6-7 decimal places. Thus, we cannot represent all decimal numbers with an exact binary representation in a computer. A typical example is 0.1, whereas 9.90625 has an exact binary representation even with single precision.

In case of a subtraction $a = b - c$, we have

$$fl(a) = fl(b) - fl(c) = a(1+\epsilon_a), \tag{2.23}$$

or

$$fl(a) = b(1+\epsilon_b) - c(1+\epsilon_c), \tag{2.24}$$

meaning that

$$fl(a)/a = 1 + \epsilon_b\frac{b}{a} - \epsilon_c\frac{c}{a}, \tag{2.25}$$

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

$$\epsilon_a \approx \frac{b}{a}(\epsilon_b - \epsilon_c), \tag{2.26}$$

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

$$|fl(a) - a|, \tag{2.27}$$

whereas the relative error is

$$\frac{|fl(a) - a|}{a} \leq \epsilon_a. \tag{2.28}$$

The above subraction is thus

$$\frac{|fl(a) - a|}{a} = \frac{|fl(b) - f(c) - (b - c)|}{a}, \tag{2.29}$$

yielding

$$\frac{|fl(a) - a|}{a} = \frac{|b\epsilon_b - c\epsilon_c|}{a}. \tag{2.30}$$

The relative error is 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. If we go back to the algorithm with the alternating sum for computing $\exp -x$ of program example 3, we do happen to know the final answer, but 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.

2.3.2 Floating-point error analysis

To understand roundoff errors in general, it is customary to regard it as a random process. One can represent these errors by a semi-empirical expression if the roundoff errors come in randomly up or down

$$\epsilon_{ro} \approx \sqrt{N}\epsilon_M, \tag{2.31}$$

where N is e.g., the number of terms in the summation over n for the exponential. Note well that this estimate can be wrong especially if the roundoff errors accumulate in one specific direction. One special case is that of subtraction of two almost equal numbers.

The total error will then be the sum of a roundoff error and an approximation error of the algorithm. The latter would correspond to the truncation test of examples 2 and 3. Let us assume that the approximation error goes like

$$\epsilon_{approx} = \frac{\alpha}{N^\beta}, \tag{2.32}$$

with the obvious limit $\epsilon_{approx} \to 0$ when $N \to \infty$. The total error reads then

$$\epsilon_{tot} = \frac{\alpha}{N^\beta} + \sqrt{N}\epsilon_M \tag{2.33}$$

We are now in a position where we can make an empirical error analysis. Let us assume that we have an exact answer with which we can compare the outcome of our algorithm with. We

label these results as Y_{exact} and Y_{alg} for the exact and algorithmic results, respectively. Suppose thereafter that our algorithm depends on the number of steps used in each iteration. An example of this is found in Examples 2 and 3 for $\exp(-x)$. The outcome of our algorithm is then a function of N, $Y_{alg} = Y_{alg}(N)$.

We assume now that the approximation error is the most important one for small values of N. This means that

$$Y_{alg}(N) = Y_{exact} + \frac{\alpha}{N^{\beta}}. \tag{2.34}$$

If we now double the number of steps and still have a result which does not vary too much from the exact one, we have

$$Y_{alg}(N) - Y_{alg}(2N) \approx \frac{\alpha}{N^{\beta}}. \tag{2.35}$$

If we plot $log_{10}(|Y_{alg}(N) - Y_{alg}(2N)|)$ versus $log_{10}(N)$, the part which is a straight line indicates the region in which our approximation for the error is valid. The slope of the line is then $-\beta$. When we increase N, we are most likely to enter into a region where roundoff errors start to dominate. If we then obtain a straight line again, the slope will most likely, if $\epsilon_{ro} \approx \sqrt{N}\epsilon_M$, be close to $1/2$.

In examples for $\exp(-x)$, we saw for $x = 20$ that Y_{exact} and Y_{alg} differ quite a lot. Even if we were to improve our truncation test, we will not be able to improve the agreement with the exact result. This means that we are essentially in the region where roundoff errors take place. A straight line will then give us the empirical behavior of the roundoff errors for the specific function under study.

2.4 Additional features of C/C++ and Fortran 90/95

2.4.1 Operators in C/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/C++ is represented by the operator % whereas in Fortran 90/95 we employ the intrinsic function MOD. Note also that the increment operator ++ and the decrement operator −− is not available in Fortran 90/95. In C/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;

C/C++ offers also interesting possibilities for combined operators. These are collected in the next table.

Finally, we show some special operators pertinent to C/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.

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 90/95.

Logical operators		
C/C++	Effect	Fortran 90/95
0	False value	.FALSE.
1	True value	.TRUE.
!x	Logical negation	.NOT.x
x&& y	Logical AND	x.AND.y
x||y	Logical inclusive	x.OR.y

Table 2.6: List of logical operators in C/C++ and Fortran 90/95.

Bitwise operations		
C/C++	Effect	Fortran 90/95
~i	Bitwise complement	NOT(j)
i&j	Bitwise and	IAND(i,j)
i^j	Bitwise exclusive or	IEOR(i,j)
i|j	Bitwise inclusive or	IOR(i,j)
i<<j	Bitwise shift left	ISHFT(i,j)
i>>n	Bitwise shift right	ISHFT(i,-j

Table 2.7: List of bitwise operations.

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

Table 2.8: C/C++ specific expressions.

2.4.2 Pointers and arrays in C/C++.

In addition to constants and variables C/C++ contain important types such as pointers and arrays (vectors and matrices). These are widely used in most C/C++ program. C/C++ allows also for pointer algebra, a feature not included in Fortran 90/95. Pointers and arrays are important elements in C/C++. To shed light on these types, consider the following setup

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

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

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

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

programs/chap2/program7.cpp

```
1   using namespace std;
2   main()
3     {
4       int var;
5       int *pointer;
6
```

```
 7        pointer = &var;
 8        var   = 421;
 9        printf("Address of the integer variable var : %p\n",&var);
10        printf("Value of var : %d\n", var);
11        printf("Value of the integer pointer variable: %p\n",pointer)
    ;
12        printf("Value which pointer is pointing at :  %d\n",*pointer)
    ;
13        printf("Address of the pointer variable : %p\n",&pointer);
14      }
```

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

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

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

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

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

programs/chap2/program8.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
```

2.4.3 Macros in C/C++

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

```
1.    #define ONE       1
2.    #define TWO       ONE + ONE
3.    #define THREE     ONE + TWO
4.
5.    main()
6.       {
```

```
7.          printf("ONE=%d, TWO=%d, THREE=%d",ONE,TWO,THREE);
8.      }
```

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

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

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

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

where we have defined the transferred variables to be of type **double**. The functions also return a **double** type. These functions could easily be generalized through the use of classes and templates, see chapter 5, 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.

2.4.4 Structures in C/C++ and TYPE in Fortran 90/95

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

However, C/C++ and Fortran 90/95 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/C++, or the correspondingly similar TYPE in Fortran 90/95. The latter data type will also be discussed in chapter 5 in connection with classes and object-based programming using Fortran 90/95.

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

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

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

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

are 10 possible single-particle orbits when we account for spin and orbital momentum projections. In this case we would thus need to allocate memory for arrays containing 10 elements.

The above structure is written in a generic way and it can be used to define other variables as well. For electrons we could write **struct** single_particle_descript electrons ; and is a new variable with the name `electrons` containing all the elements of `single_particle_descript`.

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

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

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

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

```
TYPE, PUBLIC :: single_particle_descript
    INTEGER :: total_orbits
    INTEGER, DIMENSION(:) , POINTER :: n, lorb , jang , spin , m_l
    CHARACTER (LEN=10) , DIMENSION(:) , POINTER :: orbit_status
    DOUBLE PRECISION, 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 will be given later.

Chapter 3

Numerical differentiation

3.1 Introduction

Numerical integration and differentiation are some of the most frequently needed methods in computational physics. Quite often we are confronted with the need of evaluating either f' or an integral $\int f(x)dx$. The aim of this chapter is to introduce some of these methods with a critical eye on numerical accuracy, following the discussion in the previous chapter. More refined methods such as Richardson's deferred extrapolation will also be discussed at the end of this chapter.

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

3.2 Numerical differentiation

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

$$\frac{df(x)}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \tag{3.1}$$

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

$$f(x+h) = f(x) + hf'(x) + \frac{h^2 f''(x)}{2} + \ldots \tag{3.2}$$

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

$$f_c'(x) \approx \frac{f(x+h) - f(x)}{h} \approx f'(x) + \frac{hf''(x)}{2} + \ldots \tag{3.3}$$

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 could represent the derivative with

$$f_2'(x) = \frac{f(x+h) - f(x)}{h} + O(h), \tag{3.4}$$

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

$$f_2'(x) = \frac{f(x) - f(x-h)}{h} + O(h). \tag{3.5}$$

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

$$f_2'(x) = 2bx + bh, \tag{3.6}$$

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

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

$$f(x = x_0 \pm h) = f(x_0) \pm hf' + \frac{h^2 f''}{2} \pm \frac{h^3 f'''}{6} + O(h^4), \tag{3.7}$$

which we rewrite as

$$f_{\pm h} = f_0 \pm hf' + \frac{h^2 f''}{2} \pm \frac{h^3 f'''}{6} + O(h^4). \tag{3.8}$$

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

$$f_3' = \frac{f_h - f_{-h}}{2h} - \frac{h^2 f'''}{6} + O(h^3), \tag{3.9}$$

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

$$f_h - 2f_0 + f_{-h} = h^2 f'' + O(h^4), \tag{3.10}$$

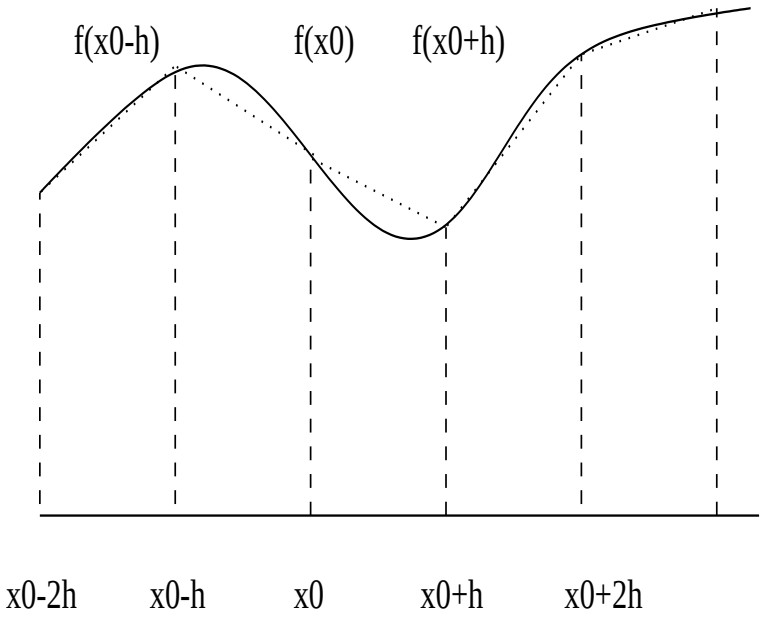


Figure 3.1: Demonstration of the subdivision of the x-axis into small steps h. See text for discussion.

we can also define higher derivatives like e.g.,

$$f'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} + O(h^2). \tag{3.11}$$

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

$$f_{\pm 2h} = f_0 \pm 2hf' + 2h^2 f'' \pm \frac{4h^3 f'''}{3} + O(h^4), \tag{3.12}$$

with a first derivative given by

$$f'_{5c} = \frac{f_{-2h} - 8f_{-h} + 8f_h - f_{2h}}{12h} + O(h^4), \tag{3.13}$$

with a dominating error of the order of h^4. This formula can be useful in case our function is represented by a fourth-order polynomial in x in the region $[-2h, 2h]$.

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

$$\frac{f_h - f_{-h}}{2h} = f'_0 + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j}, \tag{3.14}$$

and

$$\frac{f_h - 2f_0 + f_{-h}}{h^2} = f_0'' + 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!}h^{2j}, \tag{3.15}$$

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

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

$$a_{-h}f_{-h} + a_0 f_0 + a_h f_h = a_{-h}\sum_{j=0}^{\infty} \frac{f_0^{(j)}}{j!}(-h)^j + a_0 f_0 + a_h \sum_{j=0}^{\infty} \frac{f_0^{(j)}}{j!}(h)^j, \tag{3.16}$$

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. (3.16) can be rewritten as

$$\begin{aligned} a_{-h}f_{-h} + a_0 f_0 + a_h f_h &= \left[a_{-h} + a_0 + a_h\right] f_0 \\ + \left[a_h - a_{-h}\right] h f_0' + \left[a_{-h} + a_h\right] \frac{h^2 f_0''}{2} &+ \sum_{j=3}^{\infty} \frac{f_0^{(j)}}{j!}(h)^j \left[(-1)^j a_{-h} + a_h\right]. \end{aligned} \tag{3.17}$$

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

$$a_{-h} + a_0 + a_h = 0, \tag{3.18}$$

$$-a_{-h} + a_h = \frac{1}{h}, \tag{3.19}$$

and

$$a_{-h} + a_h = 0. \tag{3.20}$$

These equations have the solution

$$a_{-h} = -a_h = -\frac{1}{2h}, \tag{3.21}$$

and

$$a_0 = 0, \tag{3.22}$$

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

$$a_{-h} + a_0 + a_h = 0, \tag{3.23}$$

$$-a_{-h} + a_h = 0, \tag{3.24}$$

and

$$a_{-h} + a_h = \frac{2}{h^2}. \tag{3.25}$$

These equations have the solution

$$a_{-h} = -a_h = -\frac{1}{h^2}, \tag{3.26}$$

and

$$a_0 = -\frac{2}{h^2}, \tag{3.27}$$

yielding

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

3.2.1 The second derivative of e^x

As an example, let us calculate the second derivatives of $\exp(x)$ for various values of x. Furthermore, we will use this section to introduce three important C/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 write e.g.,

```
printf(''speed_of_sound = %lf\n'', speed_of_sound);
```

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

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

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

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

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

Consider now the following program

```
    //
    // This program module
    //  demonstrates memory allocation and data transfer in
    // between functions in  C++
    //

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

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

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

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

There are several features to be noted.

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

  The value of a: a. The address of a: &a
  The value of b: *b. The address of b: &b.

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

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

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

- Line 6: The main() function calls the function func() and the program counter transfers to the first statement in func(). With respect to data the following happens. The content of a (= 10) and the content of b (a memory address) are copied to a stack (new memory location) associated with the function func()
- Line 7: The variable x and y are local variables in func(). They have the values – x = 10, y = address of the first element in b in the main().
- Line 8: The local variable x stored in the stack memory is changed to 17. Nothing happens with the value a in main().
- Line 9: The value of y is an address and the symbol *y means the position in memory which has this address. The value in this location is now increased by 10. This means that the value of b[0] in the main program is equal to 10. Thus func() has modified a value in main().
- Line 10: This statement has the same effect as line 9 except that it modifies the element b[6] in main() by adding a value of 10 to what was there originally, namely 5.
- Line 11: The program counter returns to main(), the next expression after *func(a,b);*. All data on the stack associated with func() are destroyed.
- The value of a is transferred to func() and stored in a new memory location called x. Any modification of x in func() does not affect in any way the value of a in main(). This is called **transfer of data by value**. On the other hand the next argument in func() is an address which is transferred to func(). This address can be used to modify the corresponding value in main(). In the C language 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
  ....
}
```

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.

Initialisations and main program

In every program we have to define the functions employed. The style chosen here is to declare these functions at the beginning, followed thereafter by the main program and the detailed task performed by each function. Another possibility is to include these functions and their statements before the main program, viz., the main program appears at the very end. I find this programming style less readable however. A further option, specially in connection with larger projects, is to include these function definitions in a user defined header file.

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

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

int main()
{
        // declarations of variables
        int number_of_steps;
        double x, initial_step;
        double *h_step, *computed_derivative;
        //  read in input data from screen
        initialise (&initial_step, &x, &number_of_steps);
        //  allocate space in memory for the one-dimensional arrays
        //  h_step and computed_derivative
        h_step =  new double[number_of_steps];
```

```
        computed_derivative = new double[number_of_steps];
        //  compute the second derivative of exp(x)
        second_derivative( number_of_steps, x, initial_step, h_step,
                           computed_derivative);
        //  Then we print the results to file
        output(h_step, computed_derivative, x, number_of_steps );
        // free memory
        delete [] h_step;
        delete [] computed_derivative;
        return 0;
}   // end main program
```

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

These arrays are defined as pointers through the statement **double** *h_step , * computed_derivative ; A call in the main function to the function second_derivative looks then like this second_derivative (number_of_steps, x, 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_deriva); indicating that **double** *h_step , **double** *computed_derivative ; are pointers and that we transfer the address of the first elements. The other variables **int** number_of_steps, **double** x; are transferred by value and are not changed in the called function.

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

The function initialise

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

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

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

The function second_derivative

```
//  This function computes the second derivative

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

The loop over the number of steps serves to compute the second derivative for different values of h. In this function the step is halved for every iteration. The step values and the derivatives are stored in the arrays h_step and **double** computed_derivative.

The output function

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

```
//    function to write out the final results
void output(double *h_step, double *computed_derivative, double x,
            int number_of_steps )
{
     int i;
     FILE *output_file;
     output_file = fopen("out.dat", "w") ;
     for( i=0; i < number_of_steps; i++)
       {
```

```
          fprintf(output_file, "%12.5E %12.5E \n",
                  log10(h_step[i]),
                  log10(fabs(computed_derivative[i]-exp(x))/exp(x)));
        }
     fclose (output_file);
}  // end of function output
```

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

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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int col:
4
5 int main(int argc, char *argv[])
6 {
7     FILE *in, *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(0);
13    in = fopen( argv[1], "r");}    // returns pointer to the in_file
14    if( inn == NULL )  {          // can't find in_file
15       printf("Can't find the input file %s\n", argv[1]);
16       exit(0);
17    }
18    out = fopen( argv[2], "w");     // returns a pointer to the
   out_file
19    if( ut == NULL )  {          // can't find out_file
20       printf("Can't find the output file %s\n", argv[2]);
21       exit(0);
22    }
    ... program statements

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

This program has several interesting features.

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

The above represents a standard procedure in C for reading file names. C++ has its own class for such operations. We will come back to such features later.

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

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 $\exp(x)$. A comparison is made with the exact value. The step size is also listed.

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.

3.2.2 Error analysis

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

$$\epsilon = log_{10}\left(\left|\frac{f''_{\rm computed} - f''_{\rm exact}}{f''_{\rm exact}}\right|\right), \tag{3.28}$$

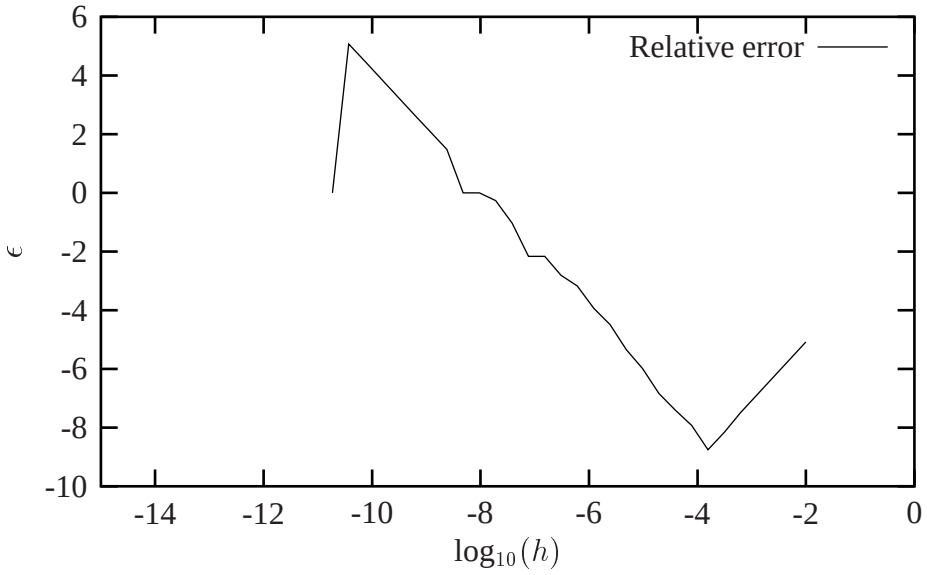


Figure 3.2: Log-log plot of the relative error of the second derivative of e^x as function of decreasing step lengths h. The second derivative was computed for $x = 10$ in the program discussed above. See text for further details

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

$$\epsilon_{\mathrm{tot}} = \epsilon_{\mathrm{approx}} + \epsilon_{\mathrm{ro}}. \tag{3.29}$$

For the computed second derivative, Eq. (3.15), we have

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

and the truncation or approximation error goes like

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

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

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

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

We have then

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

Our total error becomes

$$|\epsilon_{\rm tot}| \leq \frac{2\epsilon_M}{h^2} + \frac{f_0^{(4)}}{12}h^2. \tag{3.30}$$

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

$$h = \left(\frac{24\epsilon_M}{f_0^{(4)}} \right)^{1/4}. \tag{3.31}$$

With double precision and $x = 10$ we obtain

$$h \approx 10^{-4}. \tag{3.32}$$

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

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

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

as

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

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

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

We can repeat this for $-10 < log_{10}(h) < -4$ and extract a slope ≈ -2. This agrees again with our simple expression in Eq. (3.30).

3.2.3 How to make figures with Gnuplot

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

h	$e^h + e^{-h}$	$e^h + e^{-h} - 2$
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^{-5}	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.

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.

In order to check if gnuplot is present type

```
which gnuplot
```

If gnuplot is available, simply write

```
gnuplot
```

to start the program. You will then see the following prompt

```
gnuplot>
```

and type help for a list of various commands and help options. Suppose you wish to plot data points stored in the file **mydata.dat**. This file contains two columns of data points, where the first column refers to the argument x while the second one refers to a computed function value $f(x)$.

If we wish to plot these sets of points with gnuplot we just to need to write

```
gnuplot>plot 'mydata.dat' using 1:2 w l
```

or

```
gnuplot>plot 'mydata.dat' w l
```

since gnuplot assigns as default the first column as the x-axis. The abbreviations **w l** stand for 'with lines'. If you prefer to plot the data points only, write

```
gnuplot>plot 'mydata.dat' w p
```

For more plotting options, how to make axis labels etc, type help and choose **plot** as topic.

Gnuplot will typically display a graph on the screen. If we wish to save this graph as a postscript file, we can proceed as follows

```
gnuplot>set terminal postscript
gnuplot>set output 'mydata.ps'
gnuplot>plot 'mydata.dat' w l
```

and you will be the owner of a postscript file called **mydata.ps**, which you can display with **ghostview** through the call

```
gv mydata.ps
```

The other alternative is to generate a figure file for the document handling program **LaTeX**. The advantage here is that the text of your figure now has the same fonts as the remaining **LaTeX** document. Fig. 3.2 was generated following the steps below. You need to edit a file which ends with **.gnu**. The file used to generate Fig. 3.2 is called **derivative.gnu** and contains the following statements, which are a mix of **LaTeX** and **Gnuplot** statements. It generates a file **derivative.tex** which can be included in a **LaTeX** document.

```
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
```

To generate the file **derivative.tex**, you need to call **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}
```

3.3 Richardson's deferred extrapolation method

Here we will show how one can use the polynomial representation discussed above in order to improve calculational results. We will again study the evaluation of the first and second derivatives of $\exp(x)$ at a given point $x = \xi$. In Eqs. (3.14) and (3.15) 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

$$D(h) = D(0) + a_1 h^2 + a_2 h^4 + a_3 h^6 + \ldots, \tag{3.33}$$

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

$$D(h) = D(0) + a_1 h^2 + a_2 h^4 + a_3 h^6 + \ldots, \tag{3.34}$$

and

$$D(h/2) = D(0) + \frac{a_1 h^2}{4} + \frac{a_2 h^4}{16} + \frac{a_3 h^6}{64} + \ldots, \tag{3.35}$$

and we can eliminate the term with a_1 by combining

$$D(h/2) + \frac{D(h/2) - D(h)}{3} = D(0) - \frac{a_2 h^4}{4} - \frac{5 a_3 h^6}{16}. \tag{3.36}$$

We see that this approximation to $D(0)$ is better than the two previous ones since the error now goes like $O(h^4)$. As an example, let us evaluate the first derivative of a function f using a step with lengths h and $h/2$. We have then

$$\frac{f_h - f_{-h}}{2h} = f_0' + O(h^2), \tag{3.37}$$

$$\frac{f_{h/2} - f_{-h/2}}{h} = f_0' + O(h^2/4), \tag{3.38}$$

which can be combined, using Eq. (3.36) to yield

$$\frac{-f_h + 8 f_{h/2} - 8 f_{-h/2} + f_{-h}}{6h} = f_0' - \frac{h^4}{480} f^{(5)}. \tag{3.39}$$

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

$$\begin{array}{cccc} D_0^{(0)} & & & \\ D_0^{(1)} & D_1^{(0)} & & \\ D_0^{(2)} & D_1^{(1)} & D_2^{(0)} & \\ D_0^{(3)} & D_1^{(2)} & D_2^{(1)} & D_3^{(0)} \\ \cdots & \cdots & \cdots & \cdots \end{array}, \tag{3.40}$$

where the elements in the first column represent the given approximations

$$D_0^{(k)} = D(h/2^k). \tag{3.41}$$

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

$$D_m^{(k)} = D_{m-1}^{(k)} + \frac{D_{m-1}^{(k+1)} - D_{m-1}^{(k)}}{4^m - 1} \tag{3.42}$$

with $m > 0$. I.e., it is a linear combination of the element to the left of it and the element right over the latter.

In Table 3.1 we presented the results for various step sizes for the second derivative of $\exp(x)$ using $f_0'' = \frac{f_h - 2f_0 + f_{-h}}{h^2}$. The results were compared with the exact ones for various x values. Note well that as the step is decreased we get closer to the exact value. However, if it is further increased, we run into problems of loss of precision. This is clearly seen for $h = 0000001$. This means that even though we could let the computer run with smaller and smaller values of the step, there is a limit for how small the step can be made before we loose precision. Consider now the results in Table 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. (3.42).

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.

Chapter 4

Classes, templates and modules

in preparation (not finished as of 11/26/03)

4.1 Introduction

C++' strength over C and F77 is the possibility to define new data types, tailored to some problem.

- A user-defined data type contains data (variables) and functions operating on the data
- Example: a point in 2D
  - data: x and y coordinates of the point
  - functions: print, distance to another point, ...
- 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.

4.2 A first encounter, the vector class

- Class MyVector: a vector
- Data: plain C array
- Functions: subscripting, change length, assignment to another vector, inner product with another vector, ...
- This examples demonstrates many aspects of C++ programming
- Create vectors of a specified length: MyVector v(n);
- Create a vector with zero length: MyVector v;
- Redimension a vector to length n: v.redim(n);
- Create a vector as a copy of another vector w: MyVector v(w);
- Extract the length of the vector: const int n = v.size();
- Extract an entry: double e = v(i);
- Assign a number to an entry: v(j) = e;
- Set two vectors equal to each other: w = v;
- Take the inner product of two vectors: double a = w.inner(v);
- or alternatively a = inner(w,v);
- Write a vector to the screen: v.print(cout);

```
class MyVector
{
private:
  double* A;                        // vector entries (C-array)
  int     length;
  void    allocate (int n);         // allocate memory, length=n
  void    deallocate();             // free memory
public:
  MyVector ();                      // MyVector v;
  MyVector (int n);                 // MyVector v(n);
  MyVector (const MyVector& w);     // MyVector v(w);
 ~MyVector ();                      // clean up dynamic memory

  bool redim (int n);                        // v.redim(m);
  MyVector& operator= (const MyVector& w);// 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 MyVector& w) const; // a = v.inner(w);
  int size () const { return length; }    // n = v.size();
};
\end{listing}
Constructors tell how we declare a variable of type MyVector and how
   this variable is initialized
\begin{lstlisting}
      MyVector v;   // declare a vector of length 0

      // this actually means calling the function

      MyVector::MyVector ()
      { A = NULL; length = 0; }
```

```
      MyVector v(n);  // declare a vector of length n

      // means calling the function

      MyVector::MyVector (int n)
      { allocate(n); }

      void MyVector::allocate (int n)
      {
        length = n;
        A = new double[n];  // create n doubles in memory
      }
```

A MyVector 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:

```
    MyVector::~MyVector ()
    {
      deallocate();
    }

    // free dynamic memory:
    void MyVector::deallocate ()
    {
      delete [] A;
    }
```

```
// v and w are MyVector objects
v = w;
// means calling
MyVector& MyVector::operator= (const MyVector& 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];
  }
  return *this;
}
// return of *this, i.e. a MyVector&, allows nested
u = v = u_vec = v_vec;
```

```
v.redim(n);  // make a v of length n

bool MyVector::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
  }
}
```

```
MyVector v(w);  // take a copy of w

MyVector::MyVector (const MyVector& w)
{
  allocate (w.size());  // "this" object gets w's length
  *this = w;            // call operator=
}
```

```
// a and v are MyVector objects; want to set

a(j) = v(i+1);
```

```
// the meaning of a(j) is defined by

inline double& MyVector::operator() (int i)
{
  return A[i-1];
  // base index is 1 (not 0 as in C/C++)
}
```

- Inline functions: function body is copied to calling code, no overhead of function call!
- Note: inline is just a hint to the compiler; there is no guarantee that the compiler really inlines the function
- Why return a double reference?

```
double& MyVector::operator() (int i) { return A[i-1]; }
// returns a reference (``pointer'') directly to A[i-1]
// such that the calling code can change A[i-1]
```

```
// given MyVector a(n), b(n), c(n);
for (int i = 1; i <= n; i++)
  c(i) = a(i)*b(i);

// compiler inlining translates this to:
for (int i = 1; i <= n; i++)
  c.A[i-1] = a.A[i-1]*b.A[i-1];
// or perhaps
for (int i = 0; i < n; i++)
  c.A[i] = a.A[i]*b.A[i];

// more optimizations by a smart compiler:
double* ap = &a.A[0]; // start of a
double* bp = &b.A[0]; // start of b
double* cp = &c.A[0]; // start of c
for (int i = 0; i < n; i++)
  cp[i] = ap[i]*bp[i];          // pure C!
```

Inlining and the programmer's complete control with the definition of subscripting allow

```
void MyVector::print (std::ostream& o) const
{
  int i;
  for (i = 1; i <= length; i++)
    o << "(" << i << ")=" << (*this)(i) << '\n';
}
```

```
double a = v.inner(w);
double MyVector::inner (const MyVector& 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;
}
```

```
// MyVector v
cout << v;

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

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

```
// have some MyVector u, v, w; double a;
u = v + a*w;
// global function operator+
MyVector operator+(const MyVector& a, const MyVector& b)
{
```

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

4.3 Classes and templates in C++

- Class MyVector is a vector of doubles
- What about a vector of floats or ints?
- Copy and edit code...?
- No, this can be done automatically by use of macros or templates!!

Templates are the native C++ constructs for parameterizing parts of classes

```
template<typename Type>
class MyVector
{
  Type* A;
  int length;
public:
  ...
  Type& operator() (int i) { return A[i-1]; }
  ...
};
```

Declarations in user code:

```
MyVector<double> a(10);
MyVector<int> counters;
```

Much simpler to use than macros for parameterization.

- It is easy to use class MyVector
- Lots of details visible in C and Fortran 77 codes are hidden inside the class
- It is not easy to write class MyVector
- Thus: rely on ready-made classes in C++ libraries unless you really want to write develop your own code and you know what are doing
- C++ programming is effective when you build your own high-level classes out of well-tested lower-level classes

4.4 Using Blitz++ with vectors and matrices

4.5 Building new classes

4.6 MODULE and TYPE declarations in Fortran 90/95

4.7 Object orienting in Fortran 90/95

4.8 An example of use of classes in C++ and Modules in Fortran 90/95

Chapter 5

Linear algebra

5.1 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. This chapter serves also the purpose of introducing important programming details such as handling memory allocation for matrices, introducing the concept of classes and templates and the auxiliary library Blitz++ [**?**].

The matrices we will deal with are primarily symmetric or hermitian. Thus, before we proceed with the more detailed description of algorithms, a brief summary of matrix properties may be appropriate.

For the sake of simplicity, let us look at a (4×4) matrix $\mathbf{A}$ and a corresponding identity matrix $\mathbf{I}$

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \qquad \mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.1}$$

The inverse of a matrix is defined by

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

Other essential features are given in table 5.1.

Finally, an important property of hermitian and symmetric matrices is that they have real eigenvalues.

5.2 Programming details

In the following discussion, matrices are always two-dimensional arrays while vectors are one-dimensional arrays. 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

Table 5.1: 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	$\sum_k a_{ik}a^*_{jk} = \sum_k a^*_{ki}a_{kj} = \delta_{ij}$

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.

5.2.1 Declaration of fixed-sized vectors and matrices

Table 5.2 presents a small program which treats essential features of vector and matrix handling where the dimensions are declared in the program code.

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

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

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

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 vect[] and matr[][] to sub_1().

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

The next definition is **int** matr[][5]. This is a pointer to a vector of five elements and the

Table 5.2: Matrix handling program where arrays are defined at compilation time

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

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

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

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

compiler must be told that each vector element contains five integers. Here an alternative version could be int (∗matr)[5] which clearly specifies that matr is a pointer to a vector of five integers.

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

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

5.2.2 Runtime declarations of vectors and matrices

As mentioned in the previous subsection a fixed size declaration of vectors and matrices before compilation is in many cases bad. You may not know beforehand the actually needed sizes of vectors and matrices. In large projects where memory is a limited factor it could be important to reduce memory requirement for matrices which are not used any more. In C an C++ it is possible and common to postpone size declarations of arrays untill you really know what you need and also release memory reservations when it is not needed any more. The details are shown in Table 5.3.

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

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

Next we reserve memory for the vector in **line d**. The library function malloc reserves memory to store row integers and return the address to the reserved region in memory. This address is stored in vec. Note, none of the integers in vec[] have been assigned any specific values.

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 Table 5.3 is the call to the user-defined functions **matrix** and **free_matrix**. These functions are defined in the library file **lib.cpp**. The code is given below.

```
/*
```

Table 5.3: Matrix handling program with dynamic array allocation.

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

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

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

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

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

```
   * 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()
```

```
    /*
     * The function
     *      void free_matrix()
     * releases the memory reserved by the function matrix()
     *for the two-dimensional matrix[][]
     * Input data:
     *  void far **matr - pointer to the matrix
```

```
     */

void free_matrix(void **matr)
{
  delete [] (char *) matr[0];
}  // End: function free_matrix()
```

5.2.3 Fortran features of matrix handling

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

First of all, when we think of an $N \times N$ matrix in Fortran and C/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, ie., by column-major, while C/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/C++ start with 0 for the first element. That is $A(1,1)$ in Fortran is equivalent to $A[0][0]$ in C/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/C++ and old Fortran 77 (can obviously also be done in Fortran 90/95). We have $N \times N$ matrices A, B and C and we wish to evaluate $A = B + C$. In C/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 77 we would have

```
  DO 10 j=1, N
    DO 20 i=1, N
      a(i,j)=b(i,j)+c(i,j)
```

```
20    CONTINUE
10 CONTINUE
```

Interchanging the order of i and j can lead to a considerable enhancement in process time. Fortran 90 writes the above statements in a much simpler way

```
a=b+c
```

However, the addition still involves N^2 operations. Operations like matrix multiplication or taking the invers involve N^3. Matrix multiplication $A = BC$ could then take the following form in C/C++

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

while Fortran 90 has an intrisic function called MATMUL, so that the above three loops are coded in one single statement

```
a=MATMUL(b,c)
```

Fortran 90 contains several array manipulation statements, such as dot product of vectors, the transpose of a matrix etc etc.

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 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 you to use a dynamic memory allocation and deallocation of arrays when they are no longer needed. In Fortran 90/95 one uses the intrisic functions **ALLOCATE** and **DEALLOCATE**, while C++ employs the function **new**.

Fortran 90 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

```
DOUBLE PRECISION, 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 90 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 90, consider the following simple example where we set up a 4×4 unity matrix.

```
        ......
        IMPLICIT NONE
!       The definition of the matrix, using dynamic allocation
        DOUBLE PRECISION, 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

```
n=SIZE(unity,DIM=1)
```

will give the size of the rows, while using DIM=2 gives that of the columns.

5.3 LU decomposition of a matrix

In this section we describe how one can decompose a matrix A in terms of a matrix B with elements only below the diagonal (and thereby the naming lower) and a matrix C which contains both the diagonal and matrix elements above the diagonal (leading to the labelling upper). Consider again the matrix $\mathbf{A}$ given in eq. (5.1). The LU decomposition method means that we can rewrite this matrix as the product of two matrices $\mathbf{B}$ and $\mathbf{C}$ where

$$\mathbf{A} = \mathbf{BC} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ b_{21} & 1 & 0 & 0 \\ b_{31} & b_{32} & 1 & 0 \\ b_{41} & b_{42} & b_{43} & 1 \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ 0 & c_{22} & c_{23} & c_{24} \\ 0 & 0 & c_{33} & c_{34} \\ 0 & 0 & 0 & c_{44} \end{pmatrix} \tag{5.2}$$

The algorithm for obtaining B and C is actually quite simple. We start always with the first column. In our simple (4×4) case we have equations for the first column

$$\begin{aligned} a_{11} &= c_{11} \\ a_{21} &= b_{21}c_{11} \\ a_{31} &= b_{31}c_{11} \\ a_{41} &= b_{41}c_{11}, \end{aligned} \tag{5.3}$$

which determine the elements c_{11}, b_{21}, b_{31} and b_{41} in **B** and **C**. Writing out the equations for the second column we get

$$\begin{aligned} a_{12} &= c_{12} \\ a_{22} &= b_{21}c_{12} + c_{22} \\ a_{32} &= b_{31}c_{12} + b_{32}c_{22} \\ a_{42} &= b_{41}c_{12} + b_{42}c_{22}. \end{aligned} \tag{5.4}$$

Here the unknowns are c_{12}, c_{22}, b_{32} and b_{42} which all can be evaluated by means of the results from the first column and the elements of **A**. Note an important feature. When going from the first to the second column we do not need any further information from the matrix elements a_{i1}. This is a general property throughout the whole algorithm. Thus the memory locations for the matrix **A** can be used to store the calculated matrix elements of **B** and **C**. This saves memory.

We can generalize this procedure into three equations

$$\begin{aligned} i < j: \quad & b_{i1}c_{1j} + b_{i2}c_{2j} + \cdots + b_{ii}c_{ij} = a_{ij} \\ i = j: \quad & b_{i1}c_{1j} + b_{i2}c_{2j} + \cdots + b_{ii}c_{jj} = a_{ij} \\ i > j: \quad & b_{i1}c_{1j} + b_{i2}c_{2j} + \cdots + c_{ij}c_{jj} = a_{ij} \end{aligned} \tag{5.5}$$

which gives the following algorithm:
Calculate the elements in **B** and **C** columnwise starting with column one. For each column (j):

- Compute the first element c_{1j} by
$$c_{1j} = a_{1j}. \tag{5.6}$$

- Next, Calculate all elements $c_{ij}, i = 2, \ldots, j-1$
$$c_{ij} = a_{ij} - \sum_{k=1}^{i-1} b_{ik} c_{kj}. \tag{5.7}$$

- Then calculate the diagonal element c_{jj}
$$c_{jj} = a_{jj} - \sum_{k=1}^{j-1} b_{jk} c_{kj}. \tag{5.8}$$

- Finally, calculate the elements $b_{ij}, i > j$
$$b_{ij} = \frac{1}{c_{jj}} \left(a_{ij} - \sum_{k=1}^{i-1} b_{ik} c_{kj} \right), \tag{5.9}$$

The algorithm is known as Crout's algoithm. A crucial point is obviously the case where c_{jj} is close or equals to zero which can lead to significant loss of precision. The solution is pivoting (interchanging rows) around the largest element in a column j. Then we are actually decomposing a rowwise permutation of the original matrix $\mathbf{A}$. The key point to notice is that eqs. (5.8, 5.9) are equal except for the case that we divide by c_{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 c_{jj} as the one which happens to fall along the diagonal in the first instance. Rather, we could promote one of the undivided b_{ij}'s in the column $i = j + 1, \ldots N$ to become the diagonal of C. The partial pivoting in Crout's method means then that we choose the largest value for c_{jj} (the pivot element) and then do the divisions by that element. Then we need to keep track of all permutations performed.

The programs which performs the above described LU decomposition

C: void ludcmp(double $**$a, int n, int $*$indx, double $*$d)
Fortran: CALL lu_decompose(a, n, indx, d)

are listed in the program libraries: *lib.c*, *f90lib.f*.

5.4 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

$$\mathbf{Ax} \equiv \mathbf{BCx} = \mathbf{w}. \tag{5.10}$$

This equation can be calculated in two steps

$$\mathbf{By} = \mathbf{w}; \qquad \mathbf{Cx} = \mathbf{y}. \tag{5.11}$$

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

$$\mathbf{BCx} = \mathbf{w}, \tag{5.12}$$

and since the determinat of $\mathbf{B}$ is equal to 1 (by construction since the diagonals of $\mathbf{B}$ equal 1) we can use the inverse of $\mathbf{B}$ to obtain

$$\mathbf{Cx} = \mathbf{B^{-1}w} = \mathbf{y}, \tag{5.13}$$

which yields the intermediate step

$$\mathbf{B^{-1}w} = \mathbf{y} \tag{5.14}$$

and multiplying with $\mathbf{B}$ on both sides we reobtain Eq. (5.11). As soon as we have $\mathbf{y}$ we can obtain $\mathbf{x}$ through $\mathbf{Cx} = \mathbf{y}$.

For our four-dimentional example this takes the form

$$\begin{aligned} y_1 &= w_1 \\ b_{21}y_1 + y_2 &= w_2 \\ b_{31}y_1 + b_{32}y_2 + y_3 &= w_3 \\ b_{41}y_1 + b_{42}y_2 + b_{43}y_3 + y_4 &= w_4. \end{aligned} \tag{5.15}$$

and

$$\begin{aligned} c_{11}x_1 + c_{12}x_2 + c_{13}x_3 + c_{14}x_4 &= y_1 \\ c_{22}x_2 + c_{23}x_3 + c_{24}x_4 &= y_2 \\ c_{33}x_3 + c_{34}x_4 &= y_3 \\ c_{44}x_4 &= y_4 \end{aligned} \tag{5.16}$$

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: void 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.

The codes are listed in the program libraries: *lib.c, f90lib.f.*

5.5 Inverse of a matrix and the determinant

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

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

where the sum runs over all permutations p of the indices $1, 2, \ldots, n$, altogether $n!$ terms. Also to calculate the inverse of **A** is a formidable task. Here we have to calculate *the complementary cofactor* a^{ij} of each element $a_i j$ which is the $(n-1)$determinant obtained by striking out the row i and column j in which the element a_{ij} appears. The inverse of **A** is the constructed as the transpose a matrix with the elements $(-)^{i+j}a^{ij}$. This involves a calculation of n^2 determinants using the formula above. Thus a simplified method is highly needed.

With the LU decomposed matrix **A** in eq. (5.2) it is rather easy to find the determinant

$$det\{\mathbf{A}\} = det\{\mathbf{B}\} \times det\{\mathbf{C}\} = det\{\mathbf{C}\}, \tag{5.17}$$

since the diagonal elements of **B** equal 1. Thus the determinant can be written

$$det\{\mathbf{A}\} = \prod_{k=1}^{N} c_{kk}. \tag{5.18}$$

The inverse is slightly more difficult to obtain from the LU decomposition. It is formally defined as

$$\mathbf{A}^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}. \tag{5.19}$$

We use this form since the computation of the inverse goes through the inverse of the matrices **B** and **C**. The reason is that the inverse of a lower (upper) triangular matrix is also a lower (upper) triangular matrix. If we call $\mathbf{D}$ for the inverse of **B**, we can determine the matrix elements of **D** through the equation

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ b_{21} & 1 & 0 & 0 \\ b_{31} & b_{32} & 1 & 0 \\ b_{41} & b_{42} & b_{43} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ d_{21} & 1 & 0 & 0 \\ d_{31} & d_{32} & 1 & 0 \\ d_{41} & d_{42} & d_{43} & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{5.20}$$

which gives the following general algorithm

$$d_{ij} = -b_{ij} - \sum_{k=j+1}^{i-1} b_{ik} d_{kj}, \tag{5.21}$$

which is valid for $i > j$. The diagonal is 1 and the upper matrix elements are zero. We solve this equation column by column (increasing order of j). In a similar way we can define an equation which gives us the inverse of the matrix **C**, labelled **E** in the equation below. This contains only non-zero matrix elements in the upper part of the matrix (plus the diagonal ones)

$$\begin{pmatrix} e_{11} & e_{12} & e_{13} & e_{14} \\ 0 & e_{22} & e_{23} & e_{24} \\ 0 & 0 & e_{33} & e_{34} \\ 0 & 0 & 0 & e_{44} \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ 0 & c_{22} & c_{23} & c_{24} \\ 0 & 0 & c_{33} & c_{34} \\ 0 & 0 & 0 & c_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{5.22}$$

with the following general equation

$$e_{ij} = -\frac{1}{c_{jj}} \sum_{k=1}^{j-1} e_{ik} c_{kj}. \tag{5.23}$$

for $i \leq j$.

A calculation of the inverse of a matrix could then be implemented in the following way:

- Set up the matrix to be inverted.
- Call the LU decomposition function.
- Check whether the determinant is zero or not.
- Then solve column by column eqs. (5.21, 5.23).

5.6 Project: Matrix operations

The aim of this exercise is to get familiar with various matrix operations, from dynamic memory allocation to the usage of programs in the library package of the course. For Fortran users memory handling and most matrix and vector operations are included in the ANSI standard of Fortran 90/95. For C++ user however, there are three possible options

1. Make your own functions for dynamic memory allocation of a vector and a matrix. Use then the library package lib.cpp with its header file lib.hpp for obtaining LU-decomposed matrices, solve linear equations etc.

2. Use the library package lib.cpp with its header file lib.hpp which includes a function `matrix` for dynamic memory allocation. This program package includes all the other functions discussed during the lectures for solving systems of linear equations, obatining the determinant, getting the inverse etc.

3. Finally, we provide on the web-page of the course a library package which uses Blitz++'s classes for array handling. You could then, since Blitz++ is installed on all machines at the lab, use these classes for handling arrays.

Your program, whether it is written in C++ or Fortran 90/95, should include dynamic memory handling of matrices and vectors.

(a) Consider the linear system of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= w_3. \end{aligned}$$

This can be written in matrix form as

$$\mathbf{A}\mathbf{x} = \mathbf{w}.$$

Use the included programs to solve the system of equations

$$\begin{aligned} -x_1 + x_2 - 4x_3 &= 0 \\ 2x_1 + 2x_2 &= 1 \\ 3x_1 + 3x_2 + 2x_3 &= \tfrac{1}{2}. \end{aligned}$$

Use first standard Gaussian elimination and compute the result analytically. Compare thereafter your analytical results with the numerical ones obatined using the programs in the program library.

(b) Consider now the 4×4 linear system of equations

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

with

$$\begin{aligned} x_1 + 2x_3 + x_4 &= 2 \\ 4x_1 - 9x_2 + 2x_3 + x_4 &= 14 \\ 8x_1 + 16x_2 + 6x_3 + 5x_4 &= -3 \\ 2x_1 + 3x_2 + 2x_3 + x_4 &= 0. \end{aligned}$$

Use again standard Gaussian elimination and compute the result analytically. Compare thereafter your analytical results with the numerical ones obtained using the programs in the program library.

(c) If the matrix A is real, symmetric and positive definite, then it has a unique factorization (called Cholesky factorization)

$$A = LU = LL^T$$

where L^T is the upper matrix, implying that

$$L^T_{ij} = L_{ji}.$$

. The algorithm for the Cholesky decomposition is a special case of the general LU-decomposition algorithm. The algorithm of this decomposition is as follows

- Calculate the diagonal element L_{ii} by setting up a loop for $i = 0$ to $i = n - 1$ (C++ indexing of matrices and vectors)

$$L_{ii} = \left(A_{ii} - \sum_{k=0}^{i-1} L^2_{ik} \right)^{1/2}. \tag{5.24}$$

- within the loop over i, introduce a new loop which goes from $j = i + 1$ to $n - 1$ and calculate

$$L_{ji} = \frac{1}{L_{ii}} \left(A_{ij} - \sum_{k=0}^{i-1} L_{ik} l_{jk} \right) . \tag{5.25}$$

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

$$\mathbf{A} = \begin{pmatrix} 6 & 3 & 2 \\ 3 & 2 & 1 \\ 2 & 1 & 1 \end{pmatrix} \tag{5.26}$$

Are the matrices in exercises a) and b) positive definite? If so, employ your function for Cholesky decomposition and compare your results with those from LU-decomposition.

Chapter 6

Non-linear equations and roots of polynomials

6.1 Introduction

In Physics we often encounter the problem of determining the root of a function $f(x)$. Especially, we may need to solve non-linear equations of one variable. Such equations are usually divided into two classes, algebraic equations involving roots of polynomials and transcendental equations. When there is only one independent variable, the problem is one-dimensional, namely to find the root or roots of a function. Except in linear problems, root finding invariably proceeds by iteration, and this is equally true in one or in many dimensions. This means that we cannot solve exactly the equations at hand. Rather, we start with some approximate trial solution. The chosen algorithm will in turn improve the solution until some predetermined convergence criterion is satisfied. The algoritms we discuss below attempt to implement this strategy. We will deal mainly with one-dimensional problems. The methods

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

$$-\frac{\hbar^2}{2m}\frac{d^2u}{dr x^2}+V(x)u(x)=Eu(x), \tag{6.1}$$

and our potential is defined as

$$V(r)=\begin{cases} -V_0 & 0\leq x<a \\ 0 & x>a \end{cases} \tag{6.2}$$

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)

$$\frac{d^2u(x)}{dx^2}+\frac{2m}{\hbar^2}\left(V_0+E\right)u(x)=0 \quad x<a, \tag{6.3}$$

and

$$\frac{d^2u(x)}{dx^2}+\frac{2m}{\hbar^2}Eu(x)=0 \quad x>a. \tag{6.4}$$

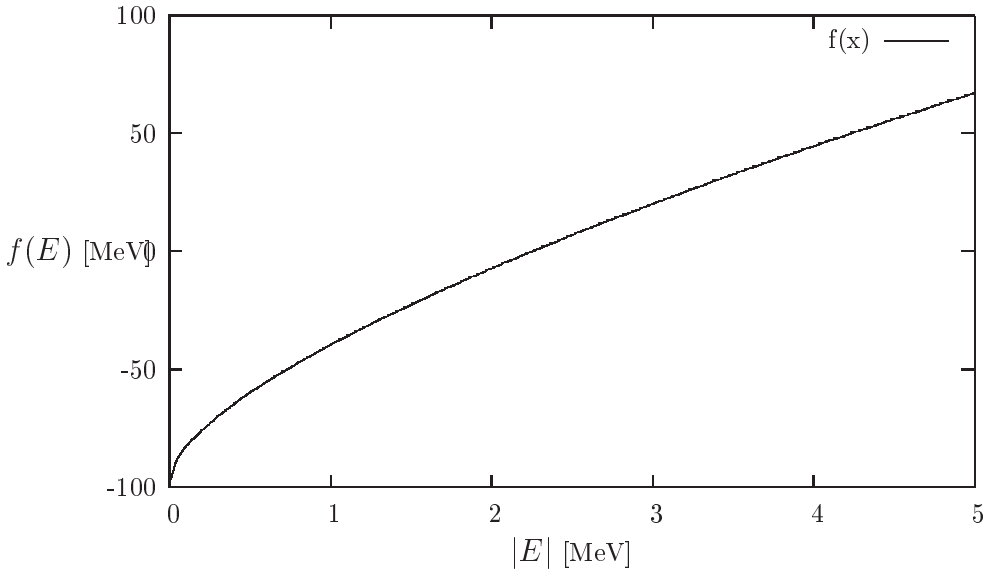


Figure 6.1: Plot of $f(E)$ Eq. (6.8) as function of energy |E| in MeV. $f(E)$ has dimension MeV. Note well that the energy E is for bound states.

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

$$u(r) = Asin(\sqrt{2m(V_0 - |E|)}r/\hbar) \qquad r < a, \tag{6.5}$$

and

$$u(r) = B\exp(-\sqrt{2m|E|}r/\hbar) \qquad r > a, \tag{6.6}$$

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

$$\sqrt{2m(V_0 - |E|)}cot(\sqrt{2ma^2(V_0 - |E|)}/\hbar) = -\sqrt{2m|E|}. \tag{6.7}$$

This equation is an example of the kind of equations which could be solved by some of the methods discussed below. The algorithms we discuss are the bisection method, the secant, false position and Brent's methods and Newton-Raphson's method. Moreover, we will also discuss how to find roots of polynomials in section 6.6.

In order to find the solution for Eq. (6.7), a simple procedure is to define a function

$$f(E) = \sqrt{2m(V_0 - |E|)}cot(\sqrt{2ma^2(V_0 - |E|)}/\hbar)\sqrt{2m|E|}. \tag{6.8}$$

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. 6.1 for $V_0 = 20$ MeV, $a = 2$ fm and $m = 938$ MeV. Fig. 6.1 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.

6.2 Iteration methods

To solve an equation of the type $f(x) = 0$ means mathematically to find all numbers s[1] so that $f(s) = 0$. In all actual calculations we are always limited by a given precision when doing numerics. Through an iterative search of the solution, the hope is that we can approach, within a given tolerance ϵ, a value x_0 which is a solution to $f(s) = 0$ if

$$|x_0 - s| < \epsilon, \tag{6.9}$$

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

$$\left|\frac{x_0 - s}{s}\right| < \epsilon, \tag{6.10}$$

and $|f(x_0)| < \epsilon$ or a combination of these. However, it is not given that the iterative process will converge and we would like to have some conditions on f which ensures a solution. This condition is provided by the so-called Lipschitz criterion. If the function f, defined on the interval $[a, b]$ satisfies for all x_1 and x_2 in the chosen interval the following condition

$$|f(x_1) - f(x_2)| \leq k\,|x_1 - x_2|\,, \tag{6.11}$$

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

$$f(x_1) - f(x_2) = f'(\xi)(x_1 - x_2), \tag{6.12}$$

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

$$|f(x_1) - f(x_2)| \leq |f'(\xi)|\,|x_1 - x_2|\,. \tag{6.13}$$

The derivative can be used as the constant k. We can now formulate the sufficient conditions for the convergence of the iterative search for solutions to $f(s) = 0$.

1. We assume that f is defined in the interval $[a, b]$.
2. f satisfies the Lipschitz condition with $k < 1$.

With these conditions, the equation $f(x) = 0$ has only one solution in the interval $[a, b]$ and it coverges after n iterations towards the solution s irrespective of choice for x_0 in the interval $[a, b]$. If we let x_n be the value of x after n iterations, we have the condition

$$|s - x_n| \leq= \frac{k}{1-k}\,|x_1 - x_2|\,. \tag{6.14}$$

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

1.
$$|x_n - s| < \epsilon, \tag{6.15}$$

and

2.
$$|f(s)| < \delta, \tag{6.16}$$

3. and a maximum number of iterations N_{maxiter} in actual calculations.

6.3 Bisection method

This is an extremely simple method to code. The philosophy can best be explained by choosing a region in e.g., Fig. 6.1 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

$$c = \frac{a+b}{2}, \tag{6.17}$$

and calculate $f(c)$. If $f(a)f(c) < 0$, the solution lies in the region $[a, c] = [a, (a+b)/2]$. Change then $b \leftarrow c$ and calculate a new value for c. If $f(a)f(c) > 0$, the new interval is in $[c, b] = [(a+b)/2, b]$. Now you need to change $a \leftarrow c$ and evaluate then a new value for c. We can continue to halve the interval till we have reached a value for c which fulfils $f(c) = 0$ to a given numerical precision. The algorithm can be simply expressed in the following program

```
        ......
        fa = f(a);
        fb = f(b);
//     check if your interval is correct, if not return to main
        if  ( fa*fb > 0) {
           cout << ``\n Error, root not in interval'' << endl;
           return;
        }
        for (j=1; j <= iter_max; j++) {
           c=(a+b)/2;
           fc=f(c)
//    if this test is satisfied, we have the root c
           if  ((abs(a-b) < epsilon ) || fc < delta ); return to main
           if ( fa*fc < 0){
              b=c  ;  fb=fc;
           }
           else{
              a=c ; fa=fc;
           }
        }
        ......
```

Note that one needs to define the values of δ, ϵ and `iter_max` when calling this function.

The bisection method is an almost foolproof method, although it may converge slowly towards the solution due to the fact that it halves the intervals. After n divisions by 2 we have a possible solution in the interval with length

$$\frac{1}{2^n}\left|b-a\right|, \tag{6.18}$$

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. (6.14) results in

$$\left|s-x_n\right| \leq = \frac{1}{2^{n+1}}\left|b-a\right|, \tag{6.19}$$

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

$$\frac{\left|s-x_n\right|}{\left|s\right|} \leq 10^{-12}. \tag{6.20}$$

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

$$\frac{\left|s-x_n\right|}{50} \leq 10^{-12}, \tag{6.21}$$

and with Eq. (6.19) we obtain

$$\frac{13}{2^{n+1}50} \leq 10^{-12}, \tag{6.22}$$

meaning $n \geq 37$.

6.4 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. (6.8) 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

$$f(s) = 0 = f(x) + (s-x)f'(x) + \frac{(s-x)^2}{2}f''(x) + \dots . \tag{6.23}$$

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

$$f(x) + (s-x)f'(x) \approx 0, \tag{6.24}$$

yielding

$$s \approx x - \frac{f(x)}{f'(x)}. \tag{6.25}$$

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

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{6.26}$$

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. 6.2

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

$$e_{n+1} = x_{n+1} - s = x_n - \frac{f(x_n)}{f'(x_n)} - s, \tag{6.27}$$

and using Eq. (6.23) we have

$$e_{n+1} = e_n + \frac{-e_n f'(x_n) + e_n^2/2f''(\xi)}{f'(x_n)} = \frac{e_n^2/2f''(\xi)}{f'(x_n)}. \tag{6.28}$$

This gives

$$\frac{|e_{n+1}|}{|e_n|^2} = \frac{1}{2}\frac{|f''(\xi)|}{|f'(x_n)|^2} = \frac{1}{2}\frac{|f''(s)|}{|f'(s)|^2} \tag{6.29}$$

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

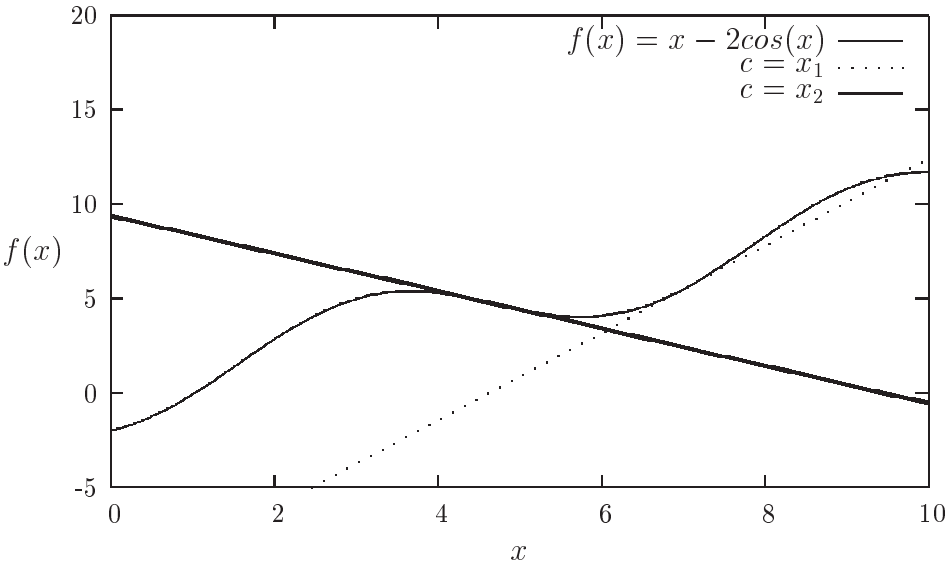


Figure 6.2: Example of a case where Newton-Raphson's method does not converge. For the function $f(x) = x - 2cos(x)$, we see that if we start at $x = 7$, the first iteration gives us that the first point where we cross the $x-$axis is given by x_1. However, using x_1 as a starting point for the next 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 $f(x) = 0$.

with respect to loss of numerical precision. Recall that the numerical evaluation of derivatives involves differences between function values at different x_n.

We can rewrite the last equation as

$$|e_{n+1}| = C|e_n|^2, \tag{6.30}$$

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

$$f(x) = x^2 - R.$$

Teh variable x is a root if $f(x) = 0$. Newton-Raphson's method yields then the following iterative approach to the root

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{R}{x_n}\right), \tag{6.31}$$

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 sustems of several non-linear equations and variables. Consider the case with two equations

$$\begin{array}{ll} f_1(x_1, x_2) & = 0 \\ f_2(x_1, x_2) & = 0 \end{array}, \tag{6.32}$$

which we Taylor expand to obtain

$$\begin{array}{lll} 0 = f_1(x_1 + h_1, x_2 + h_2) = & f_1(x_1, x_2) + h_1\partial f_1/\partial x_1 + h_2\partial f_1/\partial x_2 + \ldots \\ 0 = f_2(x_1 + h_1, x_2 + h_2) = & f_2(x_1, x_2) + h_1\partial f_2/\partial x_1 + h_2\partial f_2/\partial x_2 + \ldots \end{array}. \tag{6.33}$$

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

$$\hat{\mathbf{J}} = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{pmatrix}, \tag{6.34}$$

we can rephrase Newton's method as

$$\begin{pmatrix} x_1^{n+1} \\ x_2^{n+1} \end{pmatrix} = \begin{pmatrix} x_1^n \\ x_2^n \end{pmatrix} + \begin{pmatrix} h_1^n \\ h_2^n \end{pmatrix}, \tag{6.35}$$

where we have defined

$$\begin{pmatrix} h_1^n \\ h_2^n \end{pmatrix} = -\hat{\mathbf{J}}^{-1} \begin{pmatrix} f_1(x_1^n, x_2^n) \\ f_2(x_1^n, x_2^n) \end{pmatrix}. \tag{6.36}$$

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.

6.5 The secant method and other methods

For functions that are smooth near a root, the methods known respectively as false position (or regula falsi) and secant method generally converge faster than bisection but slower than Newton-Raphson. In both of these methods the function is assumed to be approximately linear in the

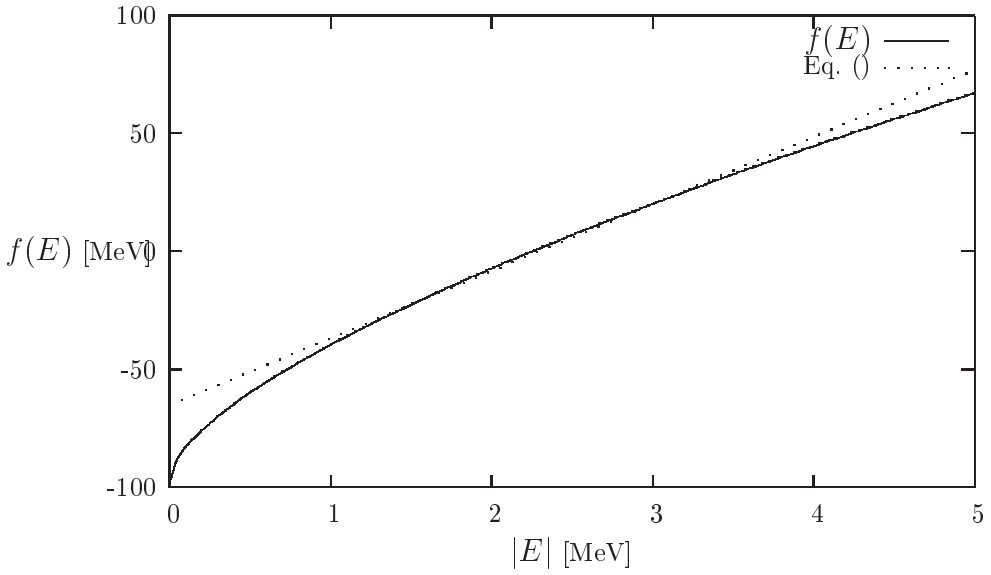


Figure 6.3: Plot of $f(E)$ Eq. (6.8) as function of energy |E|. The point c is determined by where the straight line from $(a, f(a))$ to $(b, f(b))$ crosses the $x - axis$.

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

$$x_{n+1} = x_n - f(x_n)\left(\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}\right), \tag{6.37}$$

which we rewrite to

$$x_{n+1} = \frac{f(x_n)x_{n-1} - f(x_{n-1}x_n)}{f(x_n) - f(x_{n-1})}. \tag{6.38}$$

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. 6.3.

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

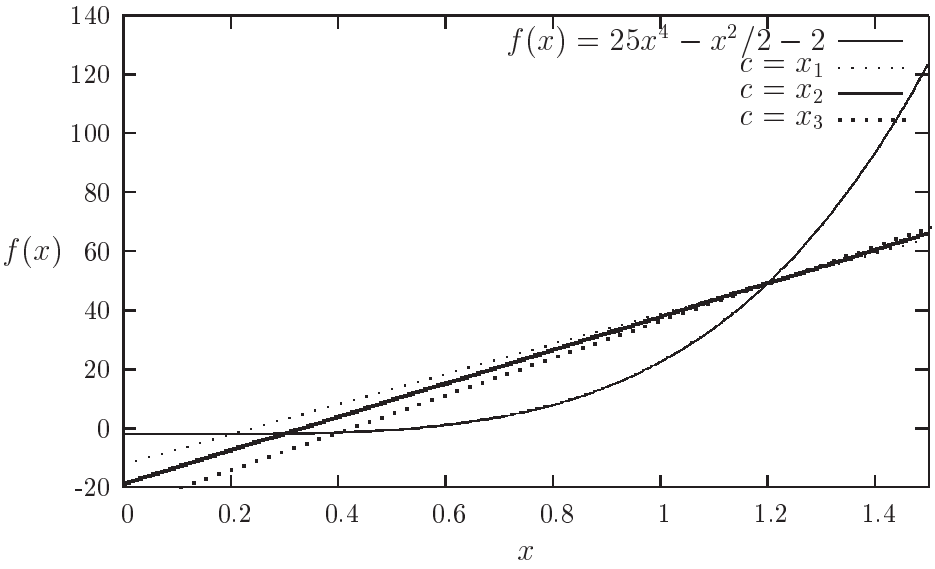


Figure 6.4: Plot of $f(x) = 25x^4 - x^2/2 - 2$. The various straight lines correspond to the determination of the point c after each iteration. c is determined by where the straight line from $(a, f(a))$ to $(b, f(b))$ crosses the $x - axis$. Here we have chosen three values for c, x_1, x_2 and x_3 which refer to the first, second and third iterations respectively.

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

$$c = \frac{f(b)a - f(a)b}{f(b) - f(a)}. \tag{6.39}$$

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

$$|e_{n+1}| \approx A|e_n|^{\alpha}, \tag{6.40}$$

with $\alpha \approx 1.62$. The convergence is better than linear, but not as good as Newton-Raphson's method which converges quadratically.

While the secant method formally converges faster than bisection, one finds in practice pathological functions for which bisection converges more rapidly. These can be choppy, discontinuous functions, or even smooth functions if the second derivative changes sharply near the root. Bisection always halves the interval, while the secant method can sometimes spend many cycles slowly pulling distant bounds closer to a root. We illustrate the weakness of this method in Fig. 6.4 where we show the results of the first three iterations, i.e., the first point is $c = x_1$, the next iteration gives $c = x_2$ while the third iterations ends with $c = x_3$. We may risk that one of the endpoints is kept fixed while the other one only slowly converges to the desired solution.

The search for the solution s proceeds in much of the same fashion as for the bisection method, namely after each iteration one of the previous boundary points is discarded in favor of the latest estimate of the root. A variation of the secant method is the so-called false position

method (regula falsi from Latin) where the interval [a,b] is chosen so that $f(a)f(b) < 0$, else there is no solution. This is rather similar to the bisection method. Another possibility is to determine the starting point for the iterative search using three points $(a, f(a))$, $(b, f(b))$ and $(c, f(c))$. One can use Lagrange's interpolation formula for a polynomial, see the discussion in next chapter. This procedure leads to Brent's method. You will find a function in the program library which computes the zeros according to the latter method as well.

6.5.1 Calling the various functions

In the program library you will find the following functions

```
rtbis(double (*func)(double), double x1, double x2, double xacc)
rtsec(double (*func)(double), double x1, double x2, double xacc)
rtnewt(void (*funcd)(double, double *, double *), double x1,
       double x2, double xacc)
zbrent(double (*func)(double), double x1, double x2, double xacc)
```

In all of these functions 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 these function, not in any case is the test $f(s) < \delta$ implemented. Rather, the test is done through $f(s) = 0$, which not necessarily is a good option.

Note also that these functions transfer a pointer to the name of the given function through e.g., *double (*func)(double)*. For Newton-Raphson's method we need a function which returns both the function and its derivative at a point x. This is then done by transferring `void (*funcd)(double, double`

6.6 Roots of polynomials

in preparation

6.6.1 Polynomials division

in preparation

6.6.2 Root finding by Newton-Raphson's method

in preparation

6.6.3 Root finding by deflation

in preparation

6.6.4 Bairstow's method

Chapter 7

Numerical interpolation, extrapolation and fitting of data

7.1 Introduction

Numerical interpolation and extrapolation is perhaps one of the most used tools in numerical applications to physics. The often encountered situation is that of a function f at a set of points $x_1 \dots x_n$ where an analytic form is missing. The function f may represent some data points from experiment or the result of a lengthy large-scale computation of some physical quantity that cannot be cast into a simple analytical form.

We may then need to evaluate the function f at some point x within the data set $x_1 \dots x_n$, but where x differs from the tabulated values. In this case we are dealing with interpolation. If x is outside we are left with the more troublesome problem of numerical extrapolation. Below we will concentrate on two methods for interpolation and extrapolation, namely polynomial interpolation and extrapolation and the qubic spline interpolation approach.

7.2 Interpolation and extrapolation

7.2.1 Polynomial interpolation and extrapolation

Let us assume that we have a set of $N+1$ points $y_0 = f(x_0), y_1 = f(x_1), \dots, y_N = f(x_N)$ where none of the x_i values are equal. We wish to determine a polynomial of degree n so that

$$P_N(x_i) = f(x_i) = y_i, \qquad i = 0, 1, \dots, N \tag{7.1}$$

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

$$P_N(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_N(x - x_0)\dots(x - x_{N-1}), \tag{7.2}$$

then Eq. (7.1) results in a triangular system of equations

$$\begin{array}{llll} a_0 & = f(x_0) & & \\ a_0+ & a_1(x_1-x_0) & = f(x_1) & \\ a_0+ & a_1(x_2-x_0)+ & a_2(x_2-x_0)(x_2-x_1) & = f(x_2) \\ \cdots & \cdots & \cdots & \cdots \end{array} . \tag{7.3}$$

The coefficients $a_0, \ldots, a_N$ are then determined in a recursive way, starting with $a_0, a_1, \ldots$.

The classic of interpolation formulae was created by Lagrange and is given by

$$P_N(x) = \sum_{i=0}^{N} \prod_{k \neq i} \frac{x - x_k}{x_i - x_k} y_i. \tag{7.4}$$

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

$$P_1(x) = \frac{x - x_0}{x_1 - x_0} y_1 + \frac{x - x_1}{x_0 - x_1} y_0, \tag{7.5}$$

and with three points (a parabolic approximation) we have

$$P_3(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} y_2 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} y_1 + \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} y_0 \tag{7.6}$$

and so forth. It is easy to see from the above equations that when $x = x_i$ we have that $f(x) = f(x_i)$ It is also possible to show that the approximation error (or rest term) is given by the second term on the right hand side of

$$f(x) = P_N(x) + \frac{\omega_{N+1}(x) f^{(N+1)}(\xi)}{(N+1)!}. \tag{7.7}$$

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

$$\omega_{N+1}(x) = a_N(x - x_0) \ldots (x - x_N), \tag{7.8}$$

and $\xi = \xi(x)$ is a point in the smallest interval containing all interpolation points x_j and x. The algorithm we provide however (the code POLINT in the program library) is based on divided differences. The recipe is quite simple. If we take $x = x_0$ in Eq. (7.2), we then have obviously that $a_0 = f(x_0) = y_0$. Moving a_0 over to the left-hand side and dividing by $x - x_0$ we have

$$\frac{f(x) - f(x_0)}{x - x_0} = a_1 + a_2(x - x_1) + \cdots + a_N(x - x_1)(x - x_2) \ldots (x - x_{N-1}), \tag{7.9}$$

where we hereafter omit the rest term

$$\frac{f^{(N+1)}(\xi)}{(N+1)!}(x - x_1)(x - x_2) \ldots (x - x_N). \tag{7.10}$$

The quantity

$$f_{0x} = \frac{f(x) - f(x_0)}{x - x_0}, \tag{7.11}$$

is a divided difference of first order. If we then take $x = x_1$, we have that $a_1 = f_{01}$. Moving a_1 to the left again and dividing by $x - x_1$ we obtain

$$\frac{f_{0x} - f_{01}}{x - x_1} = a_2 + \cdots + a_N(x - x_2)\ldots(x - x_{N-1}). \tag{7.12}$$

and the quantity

$$f_{01x} = \frac{f_{0x} - f_{01}}{x - x_1}, \tag{7.13}$$

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

$$a_1 = f_{01}, \tag{7.14}$$

is determined from f_{0x} by setting $x = x_1$. We can continue along this line and define the divided difference of order $k + 1$ as

$$f_{01\ldots kx} = \frac{f_{01\ldots(k-1)x} - f_{01\ldots(k-1)k}}{x - x_k}, \tag{7.15}$$

meaning that the corresponding coefficient a_k is given by

$$a_k = f_{01\ldots(k-1)k}. \tag{7.16}$$

With these definitions we see that Eq. (7.7) can be rewritten as

$$f(x) = a_0 + \sum_{k=1} N f_{01\ldots k}(x - x_0)\ldots(x - x_{k-1}) + \frac{\omega_{N+1}(x) f^{(N+1)}(\xi)}{(N+1)!}. \tag{7.17}$$

If we replace $x_0, x_1, \ldots, x_k$ in Eq. (7.15) with $x_{i+1}, x_{i+2}, \ldots, x_k$, 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

$$f_{x_i x_{i+1}\ldots x_k} = \frac{f_{x_{i+1}\ldots x_k} - f_{x_i x_{i+1}\ldots x_{k-1}}}{x_k - x_i}. \tag{7.18}$$

Assuming that we have a table with function values $(x_j, f(x_j) = y_j)$ and need to construct the coefficients for the polynomial $P_N(x)$. We can then view the last equation by constructing the following table for the case where $N = 3$.

$$\begin{array}{llllll}
x_0 & y_0 & & & & \\
 & & f_{x_0x_1} & & & \\
x_1 & y_1 & & f_{x_0x_1x_2} & & \\
 & & f_{x_1x_2} & & f_{x_0x_1x_2x_3} & . \\
x_2 & y_2 & & f_{x_1x_2x_3} & & \\
 & & f_{x2x_3} & & & \\
x_3 & y_3 & & & &
\end{array} \tag{7.19}$$

The coefficients we are searching for will then be the elements along the main diagonal. We can understand this algorithm by considering the following. First we construct the unique polynomial of order zero which passes through the point x_0, y_0. This is just a_0 discussed above. Therafter we construct the unique polynomial of order one which passes through both $x_0 y_0$ and $x_1 y_1$. This corresponds to the coefficient a_1 and the tabulated value $f_{x_0 x_1}$ 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 $f_{x_1 x_2}$ and $f_{x_2 x_3}$. Furthermore, a coefficient like $a_2 = f_{x_0 x_1 x_2}$ spans now three points, and adding together $f_{x_0 x_1}$ we obtain a polynomial which represents three points, a parabola. In this fashion we can continue till we have all coefficients. The function POLINT included in the library is based on an extension of this algorithm, knowns as Neville's algorithm. It is based on equidistant interpolation points. The error provided by the call to the function POLINT is based on the truncation error in Eq. (7.7).

Exercise 6.1
Use the function $f(x) = x^3$ to generate function values at four points $x_0 = 0$, $x_1 = 1$, $x_2 = 5$ and $x_3 = 6$. Use the above described method to show that the interpolating polynomial becomes $P_3(x) = x + 6x(x-1) + x(x-1)(x-5)$. Compare the exact answer with the polynomial P_3 and estimate the rest term.

7.3 Qubic spline interpolation

Qubic spline interpolation is among one of the mostly used methods for interpolating between data points where the arguments are organized as ascending series. In the library program we supply such a function, based on the so-called qubic spline method to be described below.

A spline function consists of polynomial pieces defined on subintervals. The different subintervals are connected via various continuity relations.

Assume we have at our disposal $n+1$ points $x_0, x_1, \ldots x_n$ 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

$$s(x) = \begin{cases} s_0(x) = a_0 x + b_0 & x \in [x_0, x_1) \\ s_1(x) = a_1 x + b_1 & x \in [x_1, x_2) \\ \ldots & \ldots \\ s_{n-1}(x) = a_{n-1} x + b_{n-1} & x \in [x_{n-1}, x_n] \end{cases} \tag{7.20}$$

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 $x_0, x_1, \ldots x_n$ and their corresponding function values.

The most commonly used spline function is the one with $k = 3$, the so-called qubic spline function. Assume that we have in adddition to the $n+1$ knots a series of functions values $y_0 = f(x_0), y_1 = f(x_1), \ldots y_n = f(x_n)$. By definition, the polynomials s_{i-1} and s_i are thence supposed to interpolate the same point i, i.e.,

$$s_{i-1}(x_i) = y_i = s_i(x_i), \tag{7.21}$$

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

$$s_i(x) = a_{i0} + a_{i1}x + a_{i2}x^2 + a_{i2}x^3, \tag{7.22}$$

yielding $4n$ coefficients to determine. Every subinterval provides in addition the $2n$ conditions

$$y_i = s(x_i), \tag{7.23}$$

and

$$s(x_{i+1}) = y_{i+1}, \tag{7.24}$$

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

$$s'_{i-1}(x_i) = s'_i(x_i), \tag{7.25}$$

yields $n-1$ conditions. Similarly,

$$s''_{i-1}(x_i) = s''_i(x_i), \tag{7.26}$$

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

$$s''_i(x_i) = f_i, \tag{7.27}$$

and

$$s''_i(x_{i+1}) = f_{i+1}, \tag{7.28}$$

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

$$s''_i(x) = \frac{f_i}{x_{i+1}-x_i}(x_{i+1}-x) + \frac{f_{i+1}}{x_{i+1}-x_i}(x-x_i), \tag{7.29}$$

and integrating twice one obtains

$$s_i(x) = \frac{f_i}{6(x_{i+1}-x_i)}(x_{i+1}-x)^3 + \frac{f_{i+1}}{6(x_{i+1}-x_i)}(x-x_i)^3 + c(x-x_i) + d(x_{i+1}-x). \tag{7.30}$$

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

$$s_i(x) = \frac{f_i}{6(x_{i+1}-x_i)}(x_{i+1}-x)^3 + \frac{f_{i+1}}{6(x_{i+1}-x_i)}(x-x_i)^3 + (\frac{y_{i+1}}{x_{i+1}-x_i} - \frac{f_{i+1}(x_{i+1}-x_i)}{6})(x-x_i) + (\frac{y_i}{x_{i+1}-x_i} - \frac{f_i(x_{i+1}-x_i)}{6})(x_{i+1}-x). \quad (7.31)$$

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

$$s'_{i-1}(x_i) = s'_i(x_i), \quad (7.32)$$

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

$$h_{i-1}f_{i-1} + 2(h_i + h_{i-1})f_i + h_i f_{i+1} = \frac{6}{h_i}(y_{i+1} - y_i) - \frac{6}{h_{i-1}}(y_i - y_{i-1}), \quad (7.33)$$

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

$$\begin{bmatrix} u_1 & h_1 & 0 & \dots & & & & \\ h_1 & u_2 & h_2 & 0 & \dots & & & \\ 0 & h_2 & u_3 & h_3 & 0 & \dots & & \\ \dots & & \dots & \dots & \dots & \dots & \dots & \\ & \dots & & & 0 & h_{n-3} & u_{n-2} & h_{n-2} \\ & & & & & 0 & h_{n-2} & u_{n-1} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \dots \\ f_{n-2} \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \dots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}. \quad (7.34)$$

Note that this is a set of tridiagonal equations and can be solved through only $O(n)$ operations. The functions supplied in the program library are $spline$ and $splint$. In order to use qubic spline interpolation you need first to call

```
spline(double x[], double y[], int n, double yp1, double yp2, double
    y2[])
```

This function takes as input $x[0,..,n-1]$ and $y[0,..,n-1]$ containing a tabulation $y_i = f(x_i)$ with $x_0 < x_1 < .. < x_{n-1}$ together with the first derivatives of $f(x)$ at x_0 and x_{n-1}, respectively. Then the function returns $y2[0,..,n-1]$ which contanin the second derivatives of $f(x_i)$ at each point x_i. n is the number of points. This function provides the qubic spline interpolation for all subintervals and is called only once. Thereafter, if you wish to make various interpolations, you need to call the function

```
splint(double x[], double y[], double y2a[], int n, double x, double *
    y)
```

which takes as input the tabulated values $x[0,..,n-1]$ and $y[0,..,n-1]$ and the output y2a[0,..,n - 1] from $spline$. It returns the value y corresponding to the point x.

Chapter 8

Numerical integration

8.1 Introduction

In this chapter we discuss some of the classic formulae such as the trapezoidal rule and Simpson's rule for equally spaced abscissas and formulae based on Gaussian quadrature. The latter are more suitable for the case where the abscissas are not equally spaced. The emphasis is on methods for evaluating one-dimensional integrals. In chapter 9 we show how Monte Carlo methods can be used to compute multi-dimensional integrals. We end this chapter with a discussion on singular integrals and the construction of a class for integration methods.

The integral

$$I = \int_a^b f(x)dx \tag{8.1}$$

has a very simple meaning. If we consider Fig. 8.1 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.

8.2 Equal step methods

In considering equal step methods, our basic tool is the Taylor expansion of the function $f(x)$ around a point x and a set of surrounding neighbouring points. The algorithm is rather simple, and the number of approximations unlimited!

- Choose a step size

$$h = \frac{b-a}{N}$$

  where N is the number of steps and a and b the lower and upper limits of integration.

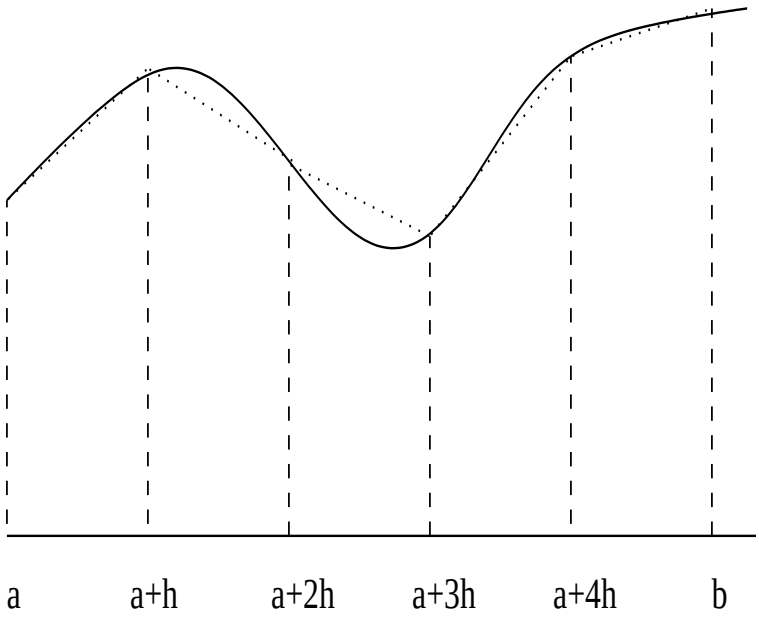


Figure 8.1: Area enscribed by the function $f(x)$ starting from $x = a$ to $x = b$. It is subdivided in several smaller areas whose evaluation is to be approximated by the techniques discussed in the text.

- Choose then to stop the Taylor expansion of the function $f(x)$ at a certain derivative. You should also choose how many points around x are to be included in the evaluation of the derivatives.
- With these approximations to $f(x)$ perform the integration.

Such a small measure may seemingly allow for the derivation of various integrals. To see this, let us briefly recall the discussion in the previous section and especially Fig. 3.1. First, we can rewrite the desired integral as

$$\int_a^b f(x)dx = \int_a^{a+2h} f(x)dx + \int_{a+2h}^{a+4h} f(x)dx + \ldots \int_{b-2h}^{b} f(x)dx. \tag{8.2}$$

The strategy then is to find a reliable Taylor expansion for $f(x)$ in the smaller sub intervals. Consider e.g., evaluating

$$\int_{-h}^{+h} f(x)dx \tag{8.3}$$

where we will Taylor expand $f(x)$ around a point x_0, see Fig. 3.1. The general form for the Taylor expansion around x_0 goes like

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

Let us now suppose that we split the integral in Eq. (8.3) in two parts, one from $-h$ to x_0 and the other from x_0 to h. Next we assume that we can use the two-point formula for the derivative, that is we can approximate $f(x)$ in these two regions by a straight line, as indicated in the figure. This means that every small element under the function $f(x)$ looks like a trapezoid, and as you may expect, the pertinent numerical approach to the integral bears the predictable name 'trapezoidal rule'. It means also that we are trying to approximate our function $f(x)$ with a first order polynomial, that is $f(x) = a + bx$. The constant b is the slope given by first derivative

$$f'(x_0 \pm h) = \frac{\mp f(x_0 \pm h) \pm f(x_0)}{h} + O(h),$$

and if we stop the Taylor expansion at that point our function becomes,

$$f(x) = f_0 + \frac{f_h - f_0}{h}x + O(x^2), \tag{8.4}$$

for $x = x_0$ to $x = x_0 + h$ and

$$f(x) = f_0 + \frac{f_0 - f_{-h}}{h}x + O(x^2), \tag{8.5}$$

for $x = x_0 - h$ to $x = x_0$. The error goes like $O(x^2)$. If we then evaluate the integral we obtain

$$\int_{-h}^{+h} f(x)dx = \frac{h}{2}\left(f_h + 2f_0 + f_{-h}\right) + O(h^3), \tag{8.6}$$

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. (8.1),

$$I = \int_a^b f(x)dx = h\left(f(a)/2 + f(a+h) + f(a+2h) + \cdots + f(b-h) + f_b/2\right), \tag{8.7}$$

with a global error which goes like $O(h^2)$. It can easily be implemented numerically through the following simple algorithm

- Choose the number of mesh points and fix the step.
- calculate $f(a)$ and $f(b)$ and multiply with $h/2$
- Perform a loop over $n = 1$ to $n - 1$ ($f(a)$ and $f(b)$ are known) and sum up the terms $f(a+h) + f(a+2h) + f(a+3h) + \cdots + f(b-h)$. Each step in the loop corresponds to a given value $a + nh$.
- Multiply the final result by h and add $hf(a)/2$ and $hf(b)/2$.

A simple function which implements this algorithm is as follows

```
double trapezoidal_rule(double a, double b, int n, double (*func)(
    double))
{
        double trapez_sum;
        double fa, fb, x, step;
        int    j;
        step=(b-a)/((double) n);
        fa=(*func)(a)/2.;
        fb=(*func)(b)/2.;
        trapez_sum=0.;
        for (j=1; j <= n-1; j++){
            x=j*step+a;
            trapez_sum+=(*func)(x);
        }
        trapez_sum=(trapez_sum+fb+fa)*step;
        return trapez_sum;
}  // end trapezoidal_rule
```

The function returns a new value for the specific integral through the variable **trapez_sum**. There is one new feature to note here, namely the transfer of a user defined function called **func** in the definition

```
  void trapezoidal_rule(double a, double b, int n, double *trapez_sum,
                        double (*func)(double) )
```

What happens here is that we are transferring a pointer to the name of a user defined function, which has as input a double precision variable and returns a double precision number. The function **trapezoidal_rule** is called as

```
  trapezoidal_rule(a, b, n, &trapez_sum, &myfunction )
```

in the calling function. We note that **a**, **b** and **n** are called by value, while **trapez_sum** and the user defined function **my_function** are called by reference.

Instead of using the above linear two-point approximation for f, we could use the three-point formula for the derivatives. This means that we will choose formulae based on function values which lie symmetrically around the point where we preform the Taylor expansion. It means also that we are approximating our function with a second-order polynomial $f(x) = a + bx + cx^2$. The first and second derivatives are given by

$$\frac{f_h - f_{-h}}{2h} = f_0' + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j}, \tag{8.8}$$

and

$$\frac{f_h - 2f_0 + f_{-h}}{h^2} = f_0'' + 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j}, \tag{8.9}$$

and we note that in both cases the error goes like $O(h^{2j})$. With the latter two expressions we can now approximate the function f as

$$f(x) = f_0 + \frac{f_h - f_{-h}}{2h}x + \frac{f_h - 2f_0 + f_{-h}}{h^2}x^2 + O(x^3). \tag{8.10}$$

Inserting this formula in the integral of Eq. (8.3) we obtain

$$\int_{-h}^{+h} f(x)dx = \frac{h}{3}\left(f_h + 4f_0 + f_{-h}\right) + O(h^5), \tag{8.11}$$

which is Simpson's rule. Note that the improved accuracy in the evaluation of the derivatives gives a better error approximation, $O(h^5)$ vs. $O(h^3)$. But this is just the *local error approximation*. Using Simpson's rule we can easily compute the integral of Eq. (8.1) to be

$$I = \int_a^b f(x)dx = \frac{h}{3}\left(f(a) + 4f(a+h) + 2f(a+2h) + \cdots + 4f(b-h) + f_b\right), \tag{8.12}$$

with a global error which goes like $O(h^4)$. It can easily be implemented numerically through the following simple algorithm

- Choose the number of mesh points and fix the step.
- calculate $f(a)$ and $f(b)$
- 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 $\frac{h}{3}$.

A critical evaluation of these methods will be given after the discussion on Guassian quadrature.

8.3 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,

$$I = \int_1^b x^{-2} f(x) dx, \tag{8.13}$$

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

$$I = \int_{b^{-1}}^1 f(t^{-1}) dt, \tag{8.14}$$

which has a small integration range and hopefully the number of mesh points needed is not that large.

However there are cases where no trick may help, and where the time expenditure in evaluating an integral is of importance. For such cases, we would like to recommend methods based on Gaussian quadrature. Here one can catch at least two birds with a stone, namely, increased precision and fewer (less time) mesh points. But it is important that the integrand varies smoothly over the interval, else we have to revert to splitting the interval into many small subintervals and the gain achieved may be lost. The mathematical details behind the theory for Gaussian quadrature formulae is quite terse. If you however are interested in the derivation, we advice you to consult the text of Stoer and Bulirsch [3], see especially section 3.6. Here we limit ourselves to merely delineate the philosophy and show examples of practical applications.

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

$$I = \int_a^b f(x) dx \approx \sum_{i=1}^N \omega_i f(x_i), \tag{8.15}$$

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. (8.12) would give

$$\omega : \{h/3, 4h/3, 2h/3, 4h/3, \ldots, 4h/3, h/3\}, \tag{8.16}$$

for the weights, while the trapezoidal rule resulted in

$$\omega : \{h/2, h, h, \ldots, h, h/2\}. \tag{8.17}$$

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

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

be determined. The points will not be equally spaced[2]. The theory behind GQ is to obtain an arbitrary weight ω through the use of so-called orthogonal polynomials. These polynomials are orthogonal in some interval say e.g., [-1,1]. Our points x_i are chosen in some optimal sense subject only to the constraint that they should lie in this interval. Together with the weights we have then $2N$ (N the number of points) parameters at our disposal.

Even though the integrand is not smooth, we could render it smooth by extracting from it the weight function of an orthogonal polynomial, i.e., we are rewriting

$$I = \int_a^b f(x)dx = \int_a^b W(x)g(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i), \tag{8.18}$$

where g is smooth and W is the weight function, which is to be associated with a given orthogonal polynomial.

The weight function W is non-negative in the integration interval $x \in [a, b]$ such that for any $n \geq 0$ $\int_a^b |x|^n W(x)dx$ is integrable. The naming weight function arises from the fact that it may be used to give more emphasis to one part of the interval than another. In physics there are several important orthogonal polynomials which arise from the solution of differential equations. These are Legendre, Hermite, Laguerre and Chebyshev polynomials. They have the following weight functions

Weight function	Interval	Polynomial
$W(x) = 1$	$x \in [a, b]$	Legendre
$W(x) = e^{-x^2}$	$-\infty \leq x \leq \infty$	Hermite
$W(x) = e^{-x^2}$	$0 \leq x \leq \infty$	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

  $$f(x) \approx P_{2N-1}(x),$$

[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.

and with only N mesh points these methods promise that

$$\int f(x)dx \approx \int P_{2N-1}(x)dx = \sum_{i=0}^{N-1} P_{2N-1}(x_i)\omega_i, \tag{8.19}$$

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.

8.3.1 Orthogonal polynomials, Legendre

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

$$C(1-x^2)P - m_l^2 P + (1-x^2)\frac{d}{dx}\left((1-x^2)\frac{dP}{dx}\right) = 0. \tag{8.20}$$

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 e.g., the solution of the angular dependence of Schrödinger's equation with spherically symmetric potentials such as the Coulomb potential.

The corresponding polynomials P are

$$L_k(x) = \frac{1}{2^k k!}\frac{d^k}{dx^k}(x^2-1)^k \qquad k = 0,1,2,\ldots, \tag{8.21}$$

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

$$\int_{-1}^{1} L_i(x)L_j(x)dx = \frac{2}{2i+1}\delta_{ij}, \tag{8.22}$$

and the recursion relation

$$(j+1)L_{j+1}(x) + jL_{j-1}(x) - (2j+1)xL_j(x) = 0. \tag{8.23}$$

It is common to choose the normalization condition

$$L_N(1) = 1. \tag{8.24}$$

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

$$L_0(x) = c, \tag{8.25}$$

with c a constant. Using the normalization equation $L_0(1) = 1$ we get that

$$L_0(x) = 1. \tag{8.26}$$

For $L_1(x)$ we have the general expression

$$L_1(x) = a + bx, \tag{8.27}$$

and using the orthorgonality relation

$$\int_{-1}^{1} L_0(x)L_1(x)dx = 0, \tag{8.28}$$

we obtain $a = 0$ and with the condition $L_1(1) = 1$, we obtain $b = 1$, yielding

$$L_1(x) = x. \tag{8.29}$$

We can proceed in a similar fashion in order to determine the coefficients of L_2

$$L_2(x) = a + bx + cx^2, \tag{8.30}$$

using the orthorgonality relations

$$\int_{-1}^{1} L_0(x)L_2(x)dx = 0, \tag{8.31}$$

and

$$\int_{-1}^{1} L_1(x)L_2(x)dx = 0, \tag{8.32}$$

and the condition $L_2(1) = 1$ we would get

$$L_2(x) = \frac{1}{2}\left(3x^2 - 1\right). \tag{8.33}$$

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. (8.23), resulting in

$$2L_2(x) = 3xL_1(x) - L_0, \tag{8.34}$$

which leads to Eq. (8.33).

The orthogonality relation above is important in our discussion of 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

$$Q_{N-1}(x) = \sum_{k=0}^{N-1} \alpha_k L_k(x), \tag{8.35}$$

where α_k's are constants.

Using the orthogonality relation of Eq. (8.22) we see that

$$\int_{-1}^{1} L_N(x) Q_{N-1}(x) dx = \sum_{k=0}^{N-1} \int_{-1}^{1} L_N(x) \alpha_k L_k(x) dx = 0. \tag{8.36}$$

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

$$L_0(x) = 1, \tag{8.37}$$

$$L_1(x) = x, \tag{8.38}$$

$$L_2(x) = (3x^2 - 1)/2, \tag{8.39}$$

$$L_3(x) = (5x^3 - 3x)/2, \tag{8.40}$$

and

$$L_4(x) = (35x^4 - 30x^2 + 3)/8. \tag{8.41}$$

The following simple function implements the above recursion relation of Eq. (8.23). 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;
         } // 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 $L_{j-1}(x)$.

8.3.2 Mesh points and weights with orthogonal polynomials

To understand how the weights and the mesh points are generated, we define first a polynomial of degree $2N-1$ (since we have $2N$ variables at hand, the mesh points and weights for N points). This polynomial can be represented through polynomial division by

$$P_{2N-1}(x) = L_N(x)P_{N-1}(x) + Q_{N-1}(x), \tag{8.42}$$

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. (8.36) to rewrite the above integral as

$$\int_{-1}^{1} P_{2N-1}(x)dx = \int_{-1}^{1} (L_N(x)P_{N-1}(x) + Q_{N-1}(x))dx = \int_{-1}^{1} Q_{N-1}(x)dx, \tag{8.43}$$

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_i where L_N is zero, we have

$$P_{2N-1}(x_i) = Q_{N-1}(x_i) \qquad i = 1, 2, \ldots, N, \tag{8.44}$$

and we see that through these N points we can fully define $Q_{N-1}(x)$ and thereby the integral. We develope then $Q_{N-1}(x)$ in terms of Legendre polynomials, as done in Eq. (8.35),

$$Q_{N-1}(x) = \sum_{i=0}^{N-1} \alpha_i L_i(x). \tag{8.45}$$

Using the orthogonality property of the Legendre polynomials we have

$$\int_{-1}^{1} Q_{N-1}(x)dx = \sum_{i=0}^{N-1} \alpha_i \int_{-1}^{1} L_0(x)L_i(x)dx = 2\alpha_0, \tag{8.46}$$

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. (8.45) as

$$Q_{N-1}(x_k) = \sum_{i=0}^{N-1} \alpha_i L_i(x_k) = \sum_{i=0}^{N-1} \alpha_i L_{ik} \qquad k = 1, 2, \ldots, N. \tag{8.47}$$

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. We can then invert the latter equation and have

$$\sum_{i=0}^{N-1} (L^{-1})_{ki} Q_{N-1}(x_i) = \alpha_k, \tag{8.48}$$

and since

$$\int_{-1}^{1} P_{2N-1}(x)dx = \int_{-1}^{1} Q_{N-1}(x)dx = 2\alpha_0 = 2\sum_{i=0}^{N-1}(L^{-1})_{0i}P_{2N-1}(x_i), \tag{8.49}$$

we see that if we identify the weights with $2(L^{-1})_{0i}$, where the points x_i are the zeros of L, we have an integration formula of the type

$$\int_{-1}^{1} P_{2N-1}(x)dx = \sum_{i=0}^{N-1}\omega_i P_{2N-1}(x_i) \tag{8.50}$$

and if our function $f(x)$ can be approximated by a polynomial P of degree $2N-1$, we have finally that

$$\int_{-1}^{1} f(x)dx \approx \int_{-1}^{1} P_{2N-1}(x)dx = \sum_{i=0}^{N-1}\omega_i P_{2N-1}(x_i). \tag{8.51}$$

In summary, the mesh points x_i are defined by the zeros of L while the weights are given by $2(L^{-1})_{0i}$.

8.3.3 Application to the case $N = 2$

Let us visualize the above formal results for the case $N = 2$. This means that we can approximate a function $f(x)$ with a polynomial $P_3(x)$ of order $2N - 1 = 3$.

The mesh points are the zeros of $L_2(x) = 1/2(3x^2 - 1)$. These points are $x_0 = -1/\sqrt{3}$ and $x_1 = 1/\sqrt{3}$.

Specializing Eq. (8.47)

$$Q_{N-1}(x_k) = \sum_{i=0}^{N-1}\alpha_i L_i(x_k) \qquad k = 1, 2, \ldots, N.$$

to $N = 2$ yields

$$Q_1(x_0) = \alpha_0 - \alpha_1\frac{1}{\sqrt{3}}, \tag{8.52}$$

and

$$Q_1(x_1) = \alpha_0 + \alpha_1\frac{1}{\sqrt{3}}, \tag{8.53}$$

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. (8.47) is then

$$L_{ik} = \begin{pmatrix} 1 & -\frac{1}{\sqrt{3}} \\ 1 & \frac{1}{\sqrt{3}} \end{pmatrix}, \tag{8.54}$$

with an inverse given by

$$(L_{ik})^{-1} = \frac{\sqrt{3}}{2}\begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ -1 & 1 \end{pmatrix}. \tag{8.55}$$

The weights are given by the matrix elements $2(L_{0k})^{-1}$. We have thence $\omega_0 = 1$ and $\omega_1 = 1$.

Summarizing, for Legendre polynomials with $N = 2$ we have weights

$$\omega : \{1, 1\}, \tag{8.56}$$

and mesh points

$$x : \left\{-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right\}. \tag{8.57}$$

If we wish to integrate

$$\int_{-1}^{1} f(x)dx,$$

with $f(x) = x^2$, we approximate

$$I = \int_{-1}^{1} x^2 dx \approx \sum_{i=0}^{N-1} \omega_i x_i^2. \tag{8.58}$$

The exact answer is $2/3$. Using $N = 2$ with the above two weights and mesh points we get

$$I = \int_{-1}^{1} x^2 dx = \sum_{i=0}^{1} \omega_i x_i^2 = \frac{1}{3} + \frac{1}{3} = \frac{2}{3}, \tag{8.59}$$

the exact answer!

If we were to emply the trapezoidal rule we would get

$$I = \int_{-1}^{1} x^2 dx = \frac{b-a}{2}\left((a)^2 + (b)^2\right)/2 = \frac{1-(-1)}{2}\left((-1)^2 + (1)^2\right)/2 = 1! \tag{8.60}$$

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?

8.3.4 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

$$t = -1 + 2\frac{x-a}{b-a}, \tag{8.61}$$

rewrite the integral for an interval [a,b]

$$\int_a^b f(t)dt = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{(b-a)x}{2} + \frac{b-a}{2}\right)dx. \tag{8.62}$$

If we have an integral on the form

$$\int_0^\infty f(t)dt, \tag{8.63}$$

we can choose new mesh points and weights by using the mapping

$$\tilde{x}_i = tan\left\{\frac{\pi}{4}(1+x_i)\right\}, \tag{8.64}$$

and

$$\tilde{\omega}_i = \frac{\pi}{4}\frac{\omega_i}{cos^2\left(\frac{\pi}{4}(1+x_i)\right)}, \tag{8.65}$$

where x_i and ω_i are the original mesh points and weights in the interval $[-1,1]$, while $\tilde{x}_i$ and $\tilde{\omega}_i$ are the new mesh points and weights for the interval $[0,\infty]$.

To see that this is correct by inserting the the value of $x_i = -1$ (the lower end of the interval $[-1,1]$) into the expression for $\tilde{x}_i$. That gives $\tilde{x}_i = 0$, the lower end of the interval $[0,\infty]$. For $x_i = 1$, we obtain $\tilde{x}_i = \infty$. To check that the new weights are correct, recall that the weights should correspond to the derivative of the mesh points. Try to convince yourself that the above expression fulfils this condition.

8.3.5 Other orthogonal polynomials

Laguerre polynomials

If we are able to rewrite our integral of Eq. (8.18) 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

$$I = \int_0^\infty f(x)dx = \int_0^\infty x^\alpha e^{-x} g(x)dx. \tag{8.66}$$

These polynomials arise from the solution of the differential equation

$$\left(\frac{d^2}{dx^2} - \frac{d}{dx} + \frac{\lambda}{x} - \frac{l(l+1)}{x^2}\right)\mathcal{L}(x) = 0, \tag{8.67}$$

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

$$\mathcal{L}_0(x) = 1, \tag{8.68}$$

$$\mathcal{L}_1(x) = 1 - x, \tag{8.69}$$

$$\mathcal{L}_2(x) = 2 - 4x + x^2, \tag{8.70}$$

$$\mathcal{L}_3(x) = 6 - 18x + 9x^2 - x^3, \tag{8.71}$$

and

$$\mathcal{L}_4(x) = x^4 - 16x^3 + 72x^2 - 96x + 24. \tag{8.72}$$

They fulfil the orthorgonality relation

$$\int_{-\infty}^{\infty} e^{-x}\mathcal{L}_n(x)^2 dx = 1, \tag{8.73}$$

and the recursion relation

$$(n+1)\mathcal{L}_{n+1}(x) = (2n+1-x)\mathcal{L}_n(x) - n\mathcal{L}_{n-1}(x). \tag{8.74}$$

Hermite polynomials

In a similar way, for an integral which goes like

$$I = \int_{-\infty}^{\infty} f(x)dx = \int_{-\infty}^{\infty} e^{-x^2} g(x)dx. \tag{8.75}$$

we could use the Hermite polynomials in order to extract weights and mesh points. The Hermite polynomials are the solutions of the following differential equation

$$\frac{d^2H(x)}{dx^2} - 2x\frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0. \tag{8.76}$$

A typical example is again the solution of Schrödinger's equation, but this time with a harmonic oscillator potential. The first few polynomials are

$$H_0(x) = 1, \tag{8.77}$$

$$H_1(x) = 2x, \tag{8.78}$$

$$H_2(x) = 4x^2 - 2, \tag{8.79}$$

$$H_3(x) = 8x^3 - 12, \tag{8.80}$$

and

$$H_4(x) = 16x^4 - 48x^2 + 12. \tag{8.81}$$

They fulfil the orthorgonality relation

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x)^2 dx = 2^n n!\sqrt{\pi}, \tag{8.82}$$

and the recursion relation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x). \tag{8.83}$$

Table 8.1: 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

8.3.6 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 e.g., $x \in [0, \infty]$, the Gauss-Legendre method places more points at the beginning of the integration interval. If your integrand varies slowly for large values of x, then this method may be appropriate.

Let us here compare three methods for integrating, namely the trapezoidal rule, Simpson's method and the Gauss-Legendre approach. We choose two functions to integrate, namely

$$\int_1^{100} \frac{\exp(-x)}{x} dx,$$

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.

```
#include <iostream>
#include "lib.h"
using namespace std;
//      Here we define various functions called by the main program
//      this function defines the function to integrate
double int_function(double x);
```

```
//    Main function begins here
int main()
{
     int n;
     double a, b;
     cout << "Read in the number of integration points" << endl;
     cin >> n;
     cout << "Read in integration limits" << endl;
     cin >> a >> b;
//    reserve space in memory for vectors containing the mesh points
//    weights and function values for the use of the gauss-legendre
//    method
     double *x = new double [n];
     double *w = new double [n];
//    set up the mesh points and weights
     gauleg(a, b,x,w, n);
//    evaluate the integral with the Gauss-Legendre method
     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
```

In Table 8.2 we show the results for the first integral using various mesh points, while Table 8.3 displays the corresponding results obtained with the second integral. We note here that, since the area over where we integrate is rather large and the integrand goes slowly to zero for large values of x, both the trapezoidal rule and Simpson's method need quite many points in order to approach the Gauss-Legendre method. This integrand demonstrates clearly the strength of the Gauss-Legendre method (and other GQ methods as well), viz., few points are needed in order to

Table 8.2: 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

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 8.3: 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

8.4 Treatment of singular Integrals

So-called principal value (PV) integrals are often employed in physics, from Green's functions for scattering to dispersion relations. Dispersion relations are often related to measurable quantities and provide important consistency checks in atomic, nuclear and particle physics. A PV integral is defined as

$$I(x) = P\int_a^b dt \frac{f(t)}{t-x} = \lim_{\epsilon\to 0^+}\left[\int_a^{x-\epsilon} dt\frac{f(t)}{t-x} + \int_{x+\epsilon}^b dt\frac{f(t)}{t-x}\right], \tag{8.84}$$

and arises in applications of Cauchy's residue theorem when the pole x lies on the real axis within the interval of integration $[a,b]$.

An important assumption is that the function $f(t)$ is continuous on the interval of integration.

In case $f(t)$ is an analytic expression or it has an analytic continuation in the complex plane, it may be possible to obtain an expression on closed form for the above integral.

However, the situation which we are often confronted with is that $f(t)$ is only known at some points t_i with corresponding values $f(t_i)$. In order to obtain $I(x)$ we need to resort to a numerical evaluation.

To evaluate such an integral, let us first rewrite it as

$$P\int_a^b dt\frac{f(t)}{t-x} = \int_a^{x-\Delta} dt\frac{f(t)}{t-x} + \int_{x+\Delta}^b dt\frac{f(t)}{t-x} + P\int_{x-\Delta}^{x+\Delta} dt\frac{f(t)}{t-x}, \tag{8.85}$$

where we have isolated the principal value part in the last integral.

Defining a new variable $u = t - x$, we can rewrite the principal value integral as

$$I_\Delta(x) = P\int_{-\Delta}^{+\Delta} du\frac{f(u+x)}{u}. \tag{8.86}$$

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

$$I_\Delta(x) \approx \sum_{n=0}^{N_{max}} f^{(2n+1)}(x)\frac{\Delta^{2n+1}}{(2n+1)(2n+1)!}. \tag{8.87}$$

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. (8.86). Let us first introduce a new variable $s = u/\Delta$ and rewrite Eq. (8.86) as

$$I_\Delta(x) = P\int_{-1}^{+1} ds\frac{f(\Delta s+x)}{s}. \tag{8.88}$$

The integration limits are now from -1 to 1, as for the Legendre polynomials. The principal value in Eq. (8.88) 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

$$\int_{-1}^{+1}\frac{ds}{s} = 0. \tag{8.89}$$

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

$$f(x)\int_{-1}^{+1}\frac{ds}{s} = \int_{-1}^{+1} f(x)\frac{ds}{s} = 0. \tag{8.90}$$

Subtracting this equation from Eq. (8.88) yields

$$I_\Delta(x) = P\int_{-1}^{+1} ds\frac{f(\Delta s+x)}{s} = \int_{-1}^{+1} ds\frac{f(\Delta s+x)-f(x)}{s}, \tag{8.91}$$

and the integrand is now longer singular since we have that $\lim_{s\to x}(f(s+x)-f(x))=0$ and for the particular case $s=0$ the integrand is now finite.

Eq. (8.91) is now rewritten using the Gauss-Legendre method resulting in

$$\int_{-1}^{+1} ds\frac{f(\Delta s+x)-f(x)}{s} = \sum_{i=1}^{N}\omega_i\frac{f(\Delta s_i+x)-f(x)}{s_i}, \tag{8.92}$$

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. (8.92) will then diverge.

Let us apply this method to the integral

$$I(x) = P\int_{-1}^{+1} dt\frac{e^t}{t}. \tag{8.93}$$

The integrand diverges at $x = t = 0$. We rewrite it using Eq. (8.91) as

$$P\int_{-1}^{+1} dt\frac{e^t}{t} = \int_{-1}^{+1}\frac{e^t-1}{t}, \tag{8.94}$$

since $e^x = e^0 = 1$. With Eq. (8.92) we have then

$$\int_{-1}^{+1}\frac{e^t-1}{t} \approx \sum_{i=1}^{N}\omega_i\frac{e^{t_i}-1}{t_i}. \tag{8.95}$$

The exact results is 2.11450175075..... With just two mesh points we recall from the previous subsection that $\omega_1 = \omega_2 = 1$ and that the mesh points are the zeros of $L_2(x)$, namely $x_1 = -1/\sqrt{3}$ and $x_2 = 1/\sqrt{3}$. Setting $N = 2$ and inserting these values in the last equation gives

$$I_2(x=0) = \sqrt{3}\left(e^{1/\sqrt{3}} - e^{-1/\sqrt{3}}\right) = 2.1129772845.$$

With six mesh points we get even the exact result to the tenth digit

$$I_6(x=0) = 2.11450175075!$$

We can repeat the above subtraction trick for more complicated integrands. First we modify the integration limits to $\pm\infty$ and use the fact that

$$\int_{-\infty}^{\infty}\frac{dk}{k-k_0} = 0. \tag{8.96}$$

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 $k \rightarrow -k$ allows us to rewrite the last equation as

$$\int_0^\infty \frac{dk}{k^2-k_0^2} = 0. \tag{8.97}$$

We can use this to express a principal values integral as

$$\mathcal{P}\int_0^\infty \frac{f(k)dk}{k^2-k_0^2} = \int_0^\infty \frac{(f(k)-f(k_0))dk}{k^2-k_0^2}, \tag{8.98}$$

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 scattering equations in momentum space, which are nothing but mere rewriting, for the non-relativistic case, of the Schrödinger equation from coordinate space to momentum space. We are going to solve numerically the scattering equation in momentum space in the chapter on eigenvalue equations, see Chapter 13.

Chapter 9

Outline of the Monte-Carlo strategy

9.1 Introduction

Monte Carlo methods are widely used, from the integration of multi-dimensional integrals to problems in chemistry, physics, medicine, biology, or Dow-Jones forecasting!

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.

Statistical simulation methods may be contrasted to conventional numerical discretization methods, which typically are applied to ordinary or partial differential equations that describe some underlying physical or mathematical system. In many applications of Monte Carlo, the physical process is simulated directly, and there is no need to even write down the differential equations that describe the behavior of the system. The only requirement is that the physical (or mathematical) system be described by probability distribution functions (PDF's). Once the PDF's are known, the Monte Carlo simulation can proceed by random sampling from the PDF's. Many simulations are then performed (multiple "trials" or "histories") and the desired result is taken as an average over the number of observations (which may be a single observation or perhaps millions of observations). In many practical applications, one can predict the statistical error (the "variance") in this average result, and hence an estimate of the number of Monte Carlo trials that are needed to achieve a given error. If we assume that the physical system can be described by a given probability density function, then the Monte Carlo simulation can proceed by sampling from these PDF's, which necessitates a fast and effective way to generate random numbers uniformly distributed on the interval [0,1]. The outcomes of these random samplings, or trials, must be accumulated or tallied in an appropriate manner to produce the desired result, but the essential characteristic of Monte Carlo is the use of random sampling techniques (and perhaps other algebra to manipulate the outcomes) to arrive at a solution of the physical problem. In contrast, a conventional numerical solution approach would start with the mathematical model of the physical system, discretizing the differential equations and then solving a set of algebraic equations for the unknown state of the system. It should be kept in mind though that this general description of Monte Carlo methods may not directly apply to some applications. It is natural

to think that Monte Carlo methods are used to simulate random, or stochastic, processes, since these can be described by PDF's. However, this coupling is actually too restrictive because many Monte Carlo applications have no apparent stochastic content, such as the evaluation of a definite integral or the inversion of a system of linear equations. However, in these cases and others, one can pose the desired solution in terms of PDF's, and while this transformation may seem artificial, this step allows the system to be treated as a stochastic process for the purpose of simulation and hence Monte Carlo methods can be applied to simulate the system.

There are, at least four ingredients which are crucial in order to understand the basic Monte-Carlo strategy. These are

1. Random variables,
2. probability distribution functions (PDF),
3. moments of a PDF
4. and its pertinent variance σ.

All these topics will be discussed at length below. We feel however that a brief explanation may be appropriate in order to convey the strategy behind a Monte-Carlo calculation. Let us first demistify the somewhat obscure concept of a random variable. The example we choose is the classic one, the tossing of two dice, its outcome and the corresponding probability. In principle, we could imagine being able to exactly determining the motion of the two dice, and with given initial conditions determine the outcome of the tossing. Alas, we are not capable of pursuing this ideal scheme. However, it does not mean that we do not have a certain knowledge of the outcome. This partial knowledge is given by the probablity of obtaining a certain number when tossing the dice. To be more precise, the tossing of the dice yields the following possible values

$$[2,3,4,5,6,7,8,9,10,11,12]. \tag{9.1}$$

These values are called the *domain*. To this domain we have the corresponding *probabilities*

$$[1/36,2/36/3/36,4/36,5/36,6/36,5/36,4/36,3/36,2/36,1/36]. \tag{9.2}$$

The numbers in the domain are the outcomes of the physical process tossing the dice. *We cannot tell beforehand whether the outcome is 3 or 5 or any other number in this domain. This defines the randomness of the outcome, or unexpectedness or any other synonimous word which encompasses the uncertitude of the final outcome.* The only thing we can tell beforehand is that say the outcome 2 has a certain probability. If our favorite hobby is to spend an hour every evening throwing dice and registering the sequence of outcomes, we will note that the numbers in the above domain

$$[2,3,4,5,6,7,8,9,10,11,12], \tag{9.3}$$

appear in a random order. After 11 throws the results may look like

$$[10,8,6,3,6,9,11,8,12,4,5]. \tag{9.4}$$

Eleven new attempts may results in a totally different sequence of numbers and so forth. Repeating this exercise the next evening, will most likely never give you the same sequences. Thus, we say that the outcome of this hobby of ours is 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 say 10ms 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 the well-known exponential function

$$\lambda \exp -(\lambda x), \tag{9.5}$$

with λ being proportional with the half-life.

9.1.1 First illustration of the use of Monte-Carlo methods, crude integration

With this definition of a random variable and its associated PDF, we attempt now a clarification of the Monte-Carlo strategy by using the evaluation of an integral as our example.

In the previous chapter we discussed standard methods for evaluating an integral like

$$I = \int_0^1 f(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i), \tag{9.6}$$

where ω_i are the weights determined by the specific integration method (like Simpson's or Taylor's methods) with x_i the given mesh points. To give you a feeling of how we are to evaluate the above integral using Monte-Carlo, we employ here the crudest possible approach. Later on we will present slightly more refined approaches. This crude approach consists in setting all weights equal 1, $\omega_i = 1$. Recall also that $dx = h = (b-a)/N$ where $b = 1$, $a = 0$ in our case and h is the step size. We can then rewrite the above integral as

$$I = \int_0^1 f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i), \tag{9.7}$$

but this is nothing but the average of f over the interval [0,1], i.e.,

$$I = \int_0^1 f(x)dx \approx \langle f \rangle. \tag{9.8}$$

In addition to the average value $\langle f \rangle$ the other important quantity in a Monte-Carlo calculation is the variance σ^2 or the standard deviation σ. We define first the variance of the integral with f

to be

$$\sigma_f^2 = \frac{1}{N}\sum_{i=1}^{N} f(x_i)^2 - \left(\frac{1}{N}\sum_{i=1}^{N} f(x_i)\right)^2, \tag{9.9}$$

or

$$\sigma_f^2 = \left(\langle f^2\rangle - \langle f\rangle^2\right). \tag{9.10}$$

which is nothing but a measure of the extent to which f deviates from its average over the region of integration.

If we consider the results for a fixed value of N as a measurement, we could however recalculate the above average and variance for a series of different measurements. If each such measumerent produces a set of averages for the integral I denoted $\langle f\rangle_l$, we have for M measumerements that the integral is given by

$$\langle I\rangle_M = \frac{1}{M}\sum_{l=1}^{M}\langle f\rangle_l. \tag{9.11}$$

The variance for these series of measurements is then for $M = N$

$$\sigma_N^2 = \frac{1}{N}\left[\langle\left(\frac{1}{N}\sum_{i=1}^{N} f(x_i)\right)^2\rangle - \left(\langle\frac{1}{N}\sum_{i=1}^{N} f(x_i)\rangle\right)^2\right]. \tag{9.12}$$

Splitting the sum in the first term on the right hand side into a sum with $i = j$ and one with $i \neq j$ we assume that in the limit of a large number of measurements only terms with $i = j$ survive, yielding

$$\sigma_N^2 \approx \frac{1}{N}\left(\langle f^2\rangle - \langle f\rangle^2\right) = \frac{\sigma_f^2}{N}. \tag{9.13}$$

We note that

$$\sigma_N \sim \frac{1}{\sqrt{N}}. \tag{9.14}$$

The aim is to have σ_N as small as possible after N samples. The results from one sample represents, since we are using concepts from statistics, a 'measurement'.

The scaling in the previous equation is clearly unfavorable compared even with the trapezoidal rule. In the previous chapter we saw that the trapezoidal rule carries a truncation error $O(h^2)$, with h the step length. In general, methods based on a Taylor expansion such as the trapezoidal rule or Simpson's rule have a truncation error which goes like $\sim O(h^k)$, with $k \geq 1$. Recalling that the step size is defined as $h = (b-a)/N$, we have an error which goes like $\sim N^{-k}$.

However, Monte Carlo integration is more efficient in higher dimensions. To see this, let us assume that our integration volume is a hypercube with side L and dimension d. This cube contains hence $N = (L/h)^d$ points and therefore the error in the result scales as $N^{-k/d}$ for the traditional methods. The error in the Monte carlo integration is however independent of d and scales as $\sigma \sim 1/\sqrt{N}$, always! Comparing this error with that of the traditional methods, shows that Monte Carlo integration is more efficient than an order-k algorithm when $d > 2k$.

Below we list a program which integrates

$$\int_0^1 dx \frac{1}{1+x^2} = \frac{\pi}{4}, \tag{9.15}$$

where the input is the desired number of Monte Carlo samples. Note that we transfer the variable $idum$ in order to initialize the random number generator from the function $ran0$. The variable $idum$ gets changed for every sampling. This variable is called the *seed*.

What we are doing is to employ a random number generator to obtain numbers x_i in the interval $[0,1]$ through e.g., a call to one of the library functions $ran0$, $ran1$, $ran2$. 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 in a pseudo-random form numbers in the interval $[0,1]$ using the so-called uniform probability distribution $p(x)$ defined as

$$p(x) = \frac{1}{b-a}\Theta(x-a)\Theta(b-x), \tag{9.16}$$

with $a = 0$ og $b = 1$. If we have a general interval $[a,b]$, we can still use these random number generators through a variable change

$$z = a + (b-a)x, \tag{9.17}$$

with x in the interval $[0,1]$.

The present approach to the above integral is often called 'crude' or 'Brute-Force' Monte-Carlo. Later on in this chapter we will study refinements to this simple approach. The reason for doing so is that a random generator produces points that are distributed in a homogenous way in the interval $[0,1]$. If our function is peaked around certain values of x, we may end up sampling function values where $f(x)$ is small or near zero. Better schemes which reflect the properties of the function to be integrated are thence needed.

The algorithm is as follows

- Choose the number of Monte Carlo samples N.
- Perform a loop over N and for each step generate a a random number x_i in the interval $[0,1]$ trough a call to a random number generator.
- Use this number to evaluate $f(x_i)$.
- Evaluate the contributions to the mean value and the standard deviation for each loop.
- After N samples calculate the final mean value and the standard deviation.

The following program implements the above algorithm using the library function $ran0$. Note the inclusion of the $lib.h$ file.

```
#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/4. << endl;
}  // end of main program

// this function defines the function to integrate

double func(double x)
{
  double value;
  value = 1./(1.+x*x);
  return value;

} // end of function to evaluate
```

The following table list the results from the above program as function of the number of Monte Carlo samples.

Table 9.1: Results for $I = \int_0^1 dx 1/(1+x^2)$ as function of number of Monte Carlo samples N. The exact answer is $7.85398E-01$ with 6 digits.

N	I	σ_N
10	7.75656E-01	4.99251E-02
100	7.57333E-01	1.59064E-02
1000	7.83486E-01	5.14102E-03
10000	7.85488E-01	1.60311E-03
100000	7.85009E-01	5.08745E-04
1000000	7.85533E-01	1.60826E-04
10000000	7.85443E-01	5.08381E-05

We note that as N increases, the standard deviation decreases, however the integral itself never reaches more than an agreement to the third or fourth digit. Improvements to this crude Monte Carlo approach will be discussed.

As an alternative, we could have used the random number generator provided by the compiler through the function $srand$, as shown in the next example.

```
// crude mc function to calculate pi
#include <iostream>

using namespace std;

int main()
{
  const int n = 1000000;
  double x, fx, pi, invers_period, pi2;
  int i;

  invers_period = 1./RAND_MAX;
  srand(time(NULL));
  pi = pi2 = 0.;
  for (i=0; i<n;i++)
    {
      x = double(rand())*invers_period;
      fx = 4./(1+x*x);
      pi += fx;
      pi2 += fx*fx;
    }
  pi /= n;  pi2 = pi2/n - pi*pi;
```

```
  cout << "pi=" << pi << " sigma^2=" << pi2 << endl;
  return 0;
}
```

9.1.2 Second illustration, particles in a box

We give here an example of how a system evolves towards a well defined equilibrium state.

Consider a box divided into two equal halves separated by a wall. At the beginning, time $t = 0$, there are N particles on the left side. A small hole in the wall is then opened and one particle can pass through the hole per unit time.

After some time the system reaches its equilibrium state with equally many particles in both halves, $N/2$. Instead of determining complicated initial conditions for a system of N particles, we model the system by a simple statistical model. In order to simulate this system, which may consist of $N \gg 1$ particles, we assume that all particles in the left half have equal probabilities of going to the right half. We introduce the label n_l to denote the number of particles at every time on the left side, and $n_r = N - n_l$ for those on the right side. The probability for a move to the right during a time step Δt is n_l/N. The algorithm for simulating this problem may then look like as follows

- Choose the number of particles N.
- Make a loop over time, where the maximum time should be larger than the number of particles N.
- For every time step Δt there is a probability n_l/N for a move to the right. Compare this probability with a random number x.
- If $x \leq n_l/N$, decrease the number of particles in the left half by one, i.e., $n_l = n_l - 1$. Else, move a particle from the right half to the left, i.e., $n_l = n_l + 1$.
- Increase the time by one unit (the external loop).

In this case, a Monte Carlo sample corresponds to one time unit Δt.

The following simple C-program illustrates this model.

```
// Particles in a box
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace  std;

ofstream  ofile;
int main(int argc, char* argv[])
{
```

```
  char *outfilename;
  int initial_n_particles, max_time, time, random_n, nleft;
  long idum;
  // Read in output file, abort if there are too few command-line
     arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  // Read in data
  cout << "Initial number of particles = " << endl ;
  cin >> initial_n_particles;
  // setup of initial conditions
  nleft = initial_n_particles;
  max_time = 10*initial_n_particles;
  idum = -1;
  // sampling over number of particles
  for( time=0; time <= max_time; time++){
    random_n = ((int) initial_n_particles*ran0(&idum));
    if ( random_n <= nleft){
      nleft -= 1;
    }
    else{
      nleft += 1;
    }
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << setw(15) << time;
    ofile << setw(15) << nleft << endl;
  }
  return 0;
} // end main function
```

The enclosed figure shows the development of this system as function of time steps. We note that for $N = 1000$ after roughly 2000 time steps, the system has reached the equilibrium state. There are however noteworthy fluctuations around equilibrium.

If we denote $\langle n_l \rangle$ as the number of particles in the left half as a time average after *equilibrium is reached*, we can define the standard deviation as

$$\sigma = \sqrt{\langle n_l^2 \rangle - \langle n_l \rangle^2}. \tag{9.18}$$

This problem has also an analytic solution to which we can compare our numerical simula-

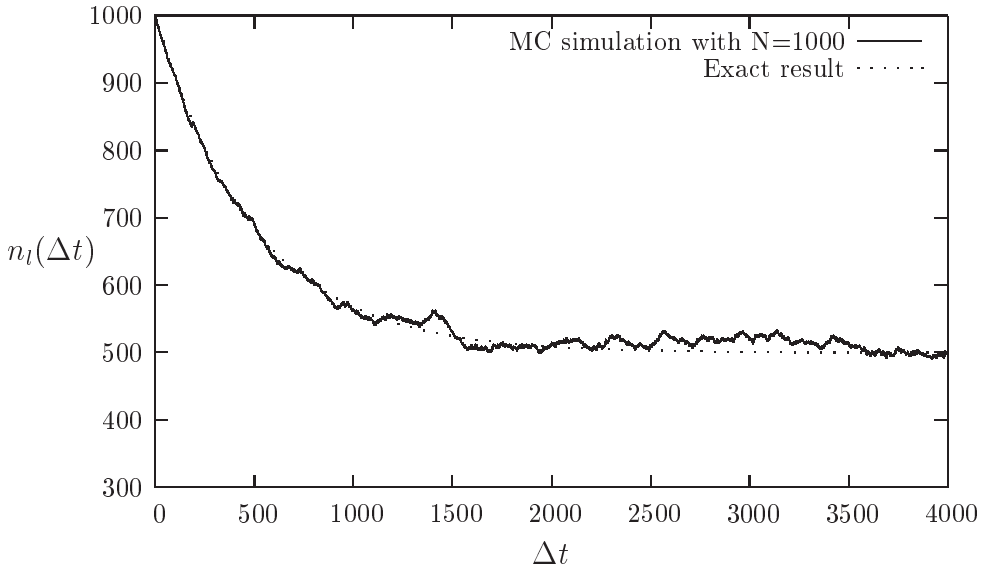


Figure 9.1: Number of particles in the left half of the container as function of the number of time steps. The solution is compared with the analytic expression. $N = 1000$.

tion. If $n_l(t)$ are the number of particles in the left half after t moves, the change in $n_l(t)$ in the time interval Δt is

$$\Delta n = \left(\frac{N - n_l(t)}{N} - \frac{n_l(t)}{N} \right) \Delta t, \tag{9.19}$$

and assuming that n_l and t are continuous variables we arrive at

$$\frac{dn_l(t)}{dt} = 1 - \frac{2n_l(t)}{N}, \tag{9.20}$$

whose solution is

$$n_l(t) = \frac{N}{2} \left(1 + e^{-2t/N} \right), \tag{9.21}$$

with the initial condition $n_l(t = 0) = N$.

9.1.3 Radioactive decay

Radioactive decay is among one of the classical examples on use of Monte-Carlo simulations. Assume that a the time $t = 0$ we have $N(0)$ nuclei of type X which can decay radioactively. At a time $t > 0$ we are left with $N(t)$ nuclei. With a transition probability ω, which expresses the probability that the system will make a transition to another state during oen second, we have the following first-order differential equation

$$dN(t) = -\omega N(t) dt, \tag{9.22}$$

whose solution is

$$N(t) = N(0)e^{-\omega t}, \tag{9.23}$$

where we have defined the mean lifetime τ of X as

$$\tau = \frac{1}{\omega}. \tag{9.24}$$

If a nucleus X decays to a daugther nucleus Y which also can decay, we get the following coupled equations

$$\frac{dN_X(t)}{dt} = -\omega_X N_X(t), \tag{9.25}$$

and

$$\frac{dN_Y(t)}{dt} = -\omega_Y N_Y(t) - \omega_X N_X(t). \tag{9.26}$$

The program example in the next subsection illustrates how we can simulate such a decay process through a Monte Carlo sampling procedure.

9.1.4 Program example for radioactive decay of one type of nucleus

The program is split in four tasks, a main program with various declarations,

```
// 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
```

the 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

```
void initialise(int& initial_n_particles, int& max_time,
                int& number_cycles, double& decay_probability)
{
  cout << "Initial number of particles = " << endl ;
  cin >> initial_n_particles;
  cout << "maximum time = " << endl;
  cin >> max_time;
  cout << "# MC steps= " << endl;
  cin >> number_cycles;
  cout << "# Decay probability= " << endl;
  cin >> decay_probability;
}  // end of function initialise
```

```
void output(int max_time, int number_cycles, int* ncumulative)
{
  int i;
  for( i=0; i <= max_time; i++){
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << setw(15) << i;
    ofile << setw(15) << setprecision(8);
    ofile  << ncumulative[i]/((double) number_cycles) << endl;
  }
}  // end of function output
```

9.1.5 Brief summary

In essence the Monte Carlo method contains the following ingredients

- A PDF which characterizes the system
- Random numbers which are generated so as to cover in a as uniform as possible way on the unity interval [0,1].
- A sampling rule
- An error estimation
- Techniques for improving the errors

Before we discuss various PDF's which may be of relevance here, we need to present some details about the way random numbers are generated. This is done in the next section. Thereafter we present some typical PDF's. Sections 5.4 and 5.5 discuss Monte Carlo integration in general, how to choose the correct weighting function and how to evaluate integrals with dimensions $d > 1$.

9.2 Physics Project: Decay of ^{210}Bi and ^{210}Po

In this project we are going to simulate the radioactive decay of these nuclei using sampling through random numbers. We assume that at $t = 0$ we have $N_X(0)$ nuclei of the type X which can decay radioactively. At a given time t we are left with $N_X(t)$ nuclei. With a transition rate ω_X, which is the probability that the system will make a transition to another state during a second, we get the following differential equation

$$dN_X(t) = -\omega_X N_X(t)dt, \tag{9.27}$$

whose solution is

$$N_X(t) = N_X(0)e^{-\omega_X t}, \tag{9.28}$$

and where the mean lifetime of the nucleus X is

$$\tau = \frac{1}{\omega_X}. \tag{9.29}$$

If the nucleus X decays to Y, which can also decay, we get the following coupled equations

$$\frac{dN_X(t)}{dt} = -\omega_X N_X(t), \tag{9.30}$$

and

$$\frac{dN_Y(t)}{dt} = -\omega_Y N_Y(t) + \omega_X N_X(t). \tag{9.31}$$

We assume that at $t = 0$ we have $N_Y(0) = 0$. In the beginning we will have an increase of N_Y nuclei, however, they will decay thereafter. In this project we let the nucleus ^{210}Bi represent X. It decays through β-decay to ^{210}Po, which is the Y nucleus in our case. The latter decays through emision of an α-particle to ^{206}Pb, which is a stable nucleus. ^{210}Bi has a mean lifetime of 7.2 days while ^{210}Po has a mean lifetime of 200 days.

a) Find analytic solutions for the above equations assuming continuous variables and setting the number of ^{210}Po nuclei equal zero at $t = 0$.

b) Make a program which solves the above equations. What is a reasonable choice of timestep Δt? You could use the program on radioactive decay from the web-page of the course as an example and make your own for the decay of two nuclei. Compare the results from your program with the exact answer as function of $N_X(0) = 10$, 100 and 1000. Make plots of your results.

c) When ^{210}Po decays it produces an α particle. At what time does the production of α particles reach its maximum? Compare your results with the analytic ones for $N_X(0) = 10$, 100 and 1000.

9.3 Random numbers

Uniform deviates are just random numbers that lie within a specified range (typically 0 to 1), with any one number in the range just as likely as any other. They are, in other words, what you probably think random numbers are. However, we want to distinguish uniform deviates from other sorts of random numbers, for example numbers drawn from a normal (Gaussian) distribution of specified mean and standard deviation. These other sorts of deviates are almost always generated by performing appropriate operations on one or more uniform deviates, as we will see in subsequent sections. So, a reliable source of random uniform deviates, the subject of this section, is an essential building block for any sort of stochastic modeling or Monte Carlo computer work. A disclaimer is however appropriate. It should be fairly obvious that something as deterministic as a computer cannot generate purely random numbers.

Numbers generated by any of the standard algorithm are in reality pseudo random numbers, hopefully abiding to the following criteria:

1. they produce a uniform distribution in the interval [0,1].
2. correlations between random numbers are negligible
3. the period before the same sequence of random numbers is repeated is as large as possible and finally
4. the algorithm should be fast.

That correlations, see below for more details, should be as small as possible resides in the fact that every event should be independent of the other ones. As an example, a particular simple system that exhibits a seemingly random behavior can be obtained from the iterative process

$$x_{i+1} = cx_i(1 - x_i), \tag{9.32}$$

which is often used as an example of a chaotic system. c is constant and for certain values of c and x_0 the system can settle down quickly into a regular periodic sequence of values $x_1, x_2, x_3, \ldots$. For $x_0 = 0.1$ and $c = 3.2$ we obtain a periodic pattern as shown in Fig. 5.2. Changing c to $c = 3.98$ yields a sequence which does not converge to any specific pattern. The values of x_i seem purely random. Although the latter choice of c yields a seemingly random sequence of values, the various values of x harbor subtle correlations that a truly random number sequence would not possess.

The most common random number generators are based on so-called Linear congruential relations of the type

$$N_i = (aN_{i-1} + c)\mathrm{MOD}(M), \tag{9.33}$$

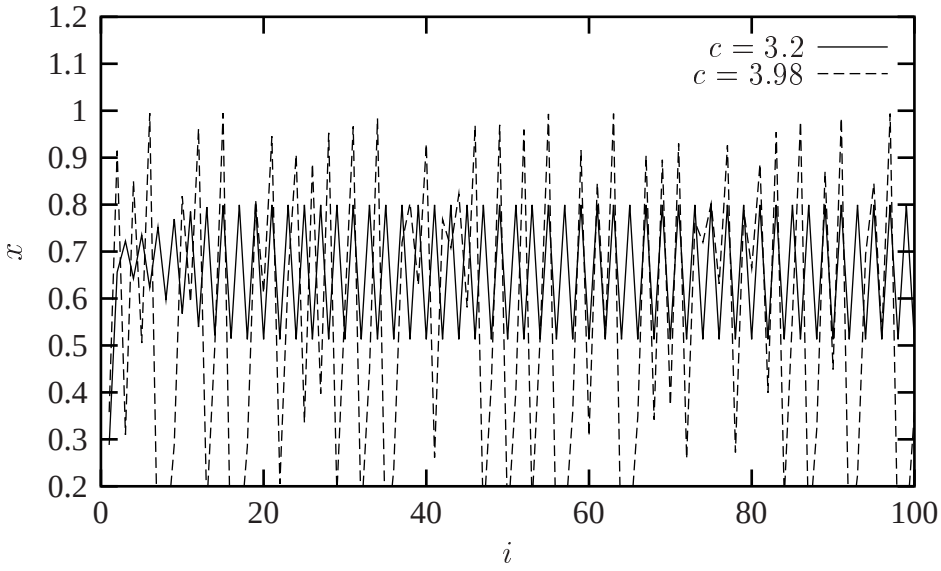


Figure 9.2: Plot of the logistic mapping $x_{i+1} = cx_i(1 - x_i)$ for $x_0 = 0.1$ and $c = 3.2$ and $c = 3.98$.

which yield a number in the interval [0,1] through

$$x_i = N_i/M \tag{9.34}$$

The number M is called the period and it should be as large as possible and N_0 is the starting value, or seed. The function MOD means the remainder, that is if we were to evaluate $(13)\mathrm{MOD}(9)$, the outcome is the remainder of the division $13/9$, namely 4.

The problem with such generators is that their outputs are periodic; they will start to repeat themselves with a period that is at most M. If however the parameters a and c are badly chosen, the period may be even shorter.

Consider the following example

$$N_i = (6N_{i-1} + 7)\mathrm{MOD}(5), \tag{9.35}$$

with a seed $N_0 = 2$. These 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

$$N_i = (27N_{i-1} + 11)\mathrm{MOD}(54), \tag{9.36}$$

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 $\sim 10^9$. Other random number generators which have become increasingly popular are so-called shift-register generators. In these generators each successive number depends on many preceding values (rather than the last values as in the linear congruential generator). For example, you could

make a shift register generator whose lth number is the sum of the $l-i$th and $l-j$th values with modulo M,

$$N_l = (aN_{l-i} + cN_{l-j})\mathrm{MOD}(M). \tag{9.37}$$

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 teh sum of each iteration. One example is the generator of Marsaglia and Zaman (Computers in Physics **8** (1994) 117) which consists of two congurential relations

$$N_l = (N_{l-3} - N_{l-1})\mathrm{MOD}(2^{31} - 69), \tag{9.38}$$

followed by

$$N_l = (69069N_{l-1} + 1013904243)\mathrm{MOD}(2^{32}), \tag{9.39}$$

which according to the authors has a period larger than 2^{94}.

Moreover, rather than using modular addition, we could use the bitwise exclusive-OR ($\oplus$) operation so that

$$N_l = (N_{l-i}) \oplus (N_{l-j}) \tag{9.40}$$

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 $\mathrm{IEOR}(m, n)$ where m and n are the input numbers, while in C it is given by m^n. The program below (from Numerical Recipes, chapter 7.1) shows the function $ran0$ which implements

$$N_i = (aN_{i-1})\mathrm{MOD}(M), \tag{9.41}$$

through Schrage's algorithm which approximates the multiplication of large integers through the factorization

$$M = aq + r,$$

or

$$q = [M/a],$$

where the brackets denote integer division and $r = (M)\mathrm{MOD}(a)$.

Note that the program uses the bitwise $\oplus$ operator to generate the starting point for each generation of a random number. The period of $ran0$ is $\sim 2.1 \times 10^9$. A special feature of this algorithm is that is should never be called with the initial seed set to 0.

```
/*
** The function
**            ran0()
** is an "Minimal" random number generator of Park and Miller
** (see Numerical recipe page 279). Set or reset the input value
** idum to any integer value (except the unlikely value MASK)
```

```
      ** to initialize the sequence; idum must not be altered between
      ** calls for sucessive deviates in a sequence.
      ** The function returns a uniform deviate between 0.0 and 1.0.
      */
double ran0(long &idum)
{
    const int a = 16807, m = 2147483647, q = 127773;
    const int r = 2836, MASK = 123459876;
    const double am = 1./m;
    long      k;
    double    ans;

    idum ^= MASK;
    k = (*idum)/q;
    idum = a*(idum - k*q) - r*k;
    if(idum < 0) idum += m;
    ans=am*(idum);
    idum ^= MASK;
    return ans;
} // End: function ran0()
```

The other random number generators $ran1$, $ran2$ and $ran3$ are described in detail in chapter 7.1 of Numerical Recipes.

Here we limit ourselves to study selected properties of these generators.

9.3.1 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 5.2 shows the distribution of $N = 10000$ random numbers generated by the functions in the program library. We note in this table that the number of points in the various intervals $0.0 - 0.1$, $0.1 - 0.2$ etc are fairly close to 1000, with some minor deviations.

Two additional measures are the standard deviation σ and the mean $\mu = \langle x \rangle$.

For the uniform distribution with N points we have that the average $\langle x^k \rangle$ is

$$\langle x^k \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i^k p(x_i), \tag{9.42}$$

and taking the limit $N \rightarrow \infty$ we have

$$\langle x^k \rangle = \int_0^1 dx p(x) x^k = \int_0^1 dx x^k = \frac{1}{k+1}, \tag{9.43}$$

since $p(x) = 1$. The mean value μ is then

$$\mu = \langle x \rangle = \frac{1}{2} \tag{9.44}$$

while the standard deviation is

$$\sigma = \sqrt{\langle x^2 \rangle - \mu^2} = \frac{1}{\sqrt{12}} = 0.2886. \tag{9.45}$$

The various random number generators produce results which agree rather well with these limiting values. In the next section, in our discussion of probability distribution functions and the central limit theorem, we are to going to see that the uniform distribution evolves towards a normal distribution in the limit $N \to \infty$.

Table 9.2: 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. Another important test is the calculation of the auto-correlation function C_k

$$C_k = \frac{\langle x_{i+k}x_i \rangle - \langle x_i \rangle^2}{\langle x_i^2 \rangle - \langle x_i \rangle^2}, \tag{9.46}$$

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. (9.13) is no longer valid.

The expectation values which enter the definition of C_k are given by

$$\langle x_{i+k}x_i \rangle = \frac{1}{N-k} \sum_{i=1}^{N-k} x_i x_{i+k}. \tag{9.47}$$

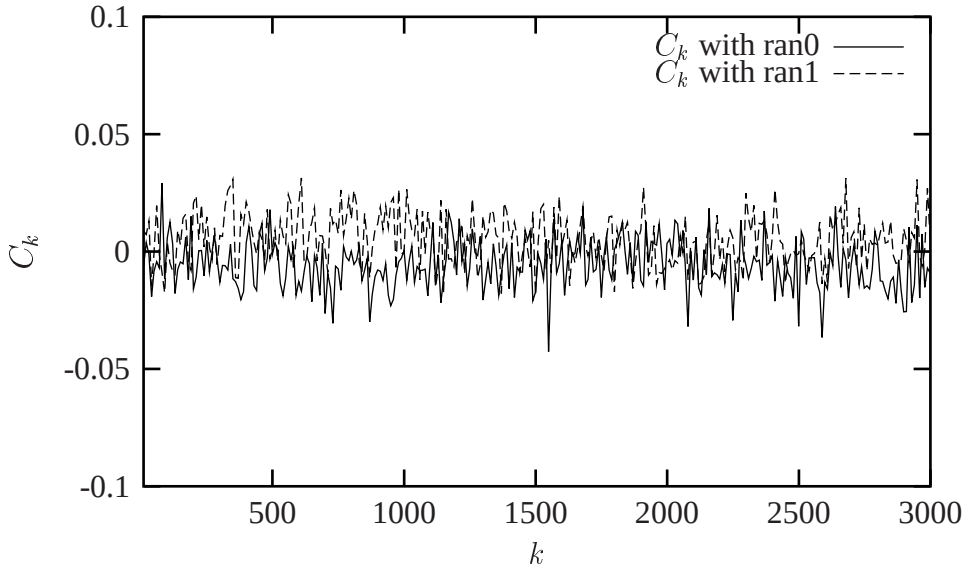


Figure 9.3: Plot of the auto-correlation function C_k for various k-values for $N = 10000$ using the random number generators $ran0$ and $ran1$.

Fig. 5.3 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 same way on the previous numbers.

Exercise 9.1
Make a program which computes random numbers according to the algorithm of Marsaglia and Zaman, Eqs. (9.38) and (9.39). Compute the correlation function C_k and compare with the auto-correlation function from the function $ran0$.

9.4 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 9.3: Important properties of PDFs.

	Discrete PDF	Continuous PDF
Domain	$\{x_1, x_2, x_3, \dots, x_N\}$	$[a, b]$
Probability	$p(x_i)$	$p(x)dx$
Cumulative	$P_i = \sum_{l=1}^{i} p(x_l)$	$P(x) = \int_a^x p(t)dt$
Positivity	$0 \leq p(x_i) \leq 1$	$p(x) \geq 0$
Positivity	$0 \leq P_i \leq 1$	$0 \leq P(x) \leq 1$
Monotonic	$P_i \geq P_j$ if $x_i \geq x_j$	$P(x_i) \geq P(x_j)$ if $x_i \geq x_j$
Normalization	$P_N = 1$	$P(b) = 1$

With a PDF we can compute expectation values of selected quantities such as

$$\langle x^k \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i^k p(x_i), \tag{9.48}$$

if we have a discrete PDF or

$$\langle x^k \rangle = \int_a^b x^k p(x)dx, \tag{9.49}$$

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 e.g.,

$$\langle f \rangle = \int_a^b f(x)p(x)dx. \tag{9.50}$$

We have already seen the use of the last equation when we applied the crude Monte Carlo approach to the evaluation of an integral.

There are at least three PDFs which one may encounter. These are the

1. uniform distribution
$$p(x) = \frac{1}{b-a}\Theta(x-a)\Theta(b-x), \tag{9.51}$$
yielding probabilities different from zero in the interval $[a, b]$,

2. the exponential distribution
$$p(x) = \alpha e^{-\alpha x}, \tag{9.52}$$
yielding probabilities different from zero in the interval $[0, \infty]$,

3. and the normal distribution
$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{9.53}$$
with probabilities different from zero in the interval $[-\infty, \infty]$,

The exponential and uniform distribution have simple cumulative functions, whereas the normal distribution does not, being proportional to the so-called error function $erf(x)$.

Exercise 9.2
Calculate the cumulative function $P(x)$ for the above three PDFs. Calculate also the corresponding mean values and standard deviations and give an interpretation of the latter.

9.4.1 The central limit theorem

subsec:centrallimit Suppose we have a PDF $p(x)$ from which we generate a series N of averages $\langle x_i \rangle$. Each mean value $\langle x_i \rangle$ is viewed as the average of a specific measurement, e.g., throwing dice 100 times and then taking the average value, or producing a certain amount of random numbers. For notational ease, we set $\langle x_i \rangle = x_i$ in the discussion which follows.

If we compute the mean z of N such mean values x_i

$$z = \frac{x_1 + x_2 + \cdots + x_N}{N}, \tag{9.54}$$

the question we pose is which is the PDF of the new variable z.

The probability of obtaining an average value z is the product of the probabilities of obtaining arbitrary individual mean values x_i, but with the constraint that the average is z. We can express this through the following expression

$$\tilde{p}(z) = \int dx_1 p(x_1) \int dx_2 p(x_2) \ldots \int dx_N p(x_N) \delta(z - \frac{x_1 + x_2 + \cdots + x_N}{N}), \tag{9.55}$$

where the δ-function enbodies the constraint that the mean is z. All measurements that lead to each individual x_i are expected to be independent, which in turn means that we can express $\tilde{p}$ as the product of individual $p(x_i)$.

If we use the integral expression for the δ-function

$$\delta(z - \frac{x_1 + x_2 + \cdots + x_N}{N}) = \frac{1}{2\pi} \int_{-\infty}^{\infty} dq e^{\left(iq(z - \frac{x_1 + x_2 + \cdots + x_N}{N})\right)}, \tag{9.56}$$

and inserting $e^{i\mu q - i\mu q}$ where μ is the mean value we arrive at

$$\tilde{p}(z) = \int_{-\infty}^{\infty} dq e^{(iq(z-\mu))} \left[\int_{-\infty}^{\infty} dx p(x) e^{(iq(\mu - x)/N)} \right]^N, \tag{9.57}$$

with the integral over x resulting in

$$\int_{-\infty}^{\infty} dx p(x) \exp\left(iq(\mu - x)/N\right) = \int_{-\infty}^{\infty} dx p(x) \left[1 + \frac{iq(\mu - x)}{N} - \frac{q^2(\mu - x)^2}{2N^2} + \ldots \right]. \tag{9.58}$$

The second term on the rhs disappears since this is just the mean and employing the definition of σ^2 we have

$$\int_{-\infty}^{\infty} dx p(x) e^{(iq(\mu-x)/N)} = 1 - \frac{q^2\sigma^2}{2N^2} + \ldots, \tag{9.59}$$

resulting in

$$\left[\int_{-\infty}^{\infty} dx p(x) \exp\left(iq(\mu-x)/N\right)\right]^N \approx \left[1 - \frac{q^2\sigma^2}{2N^2} + \ldots\right]^N, \tag{9.60}$$

and in the limit $N \to \infty$ we obtain

$$\tilde{p}(z) = \frac{1}{\sqrt{2\pi}(\sigma/\sqrt{N})} \exp\left(-\frac{(z-\mu)^2}{2(\sigma/\sqrt{N})^2}\right), \tag{9.61}$$

which is the normal distribution with variance $\sigma_N^2 = \sigma^2/N$, where σ is the variance of the PDF $p(x)$ and μ is also the mean of the PDF $p(x)$.

Thus, the central limit theorem states that the PDF $\tilde{p}(z)$ of the average of N random values corresponding to a PDF $p(x)$ is a normal distribution whose mean is the mean value of the PDF $p(x)$ and whose variance is the variance of the PDF $p(x)$ divided by N, the number of values used to compute z.

The theorem is satisfied by a large class of PDFs. Note however that for a finite N, it is not always possible to find a closed expression for $\tilde{p}(x)$.

9.5 Improved Monte Carlo integration

In section 5.1 we presented a simple brute force approach to integration with the Monte Carlo method. There we sampled over a given number of points distributed uniformly in the interval $[0,1]$

$$I = \int_0^1 f(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i) = \sum_{i=1}^{N} f(x_i) = \langle f \rangle,$$

with the weights $\omega_i = 1$.

Here we introduce two important topics which in most cases improve upon the above simple brute force approach with the uniform distribution $p(x) = 1$ for $x \in [0,1]$. With improvements we think of a smaller variance and the need for fewer Monte Carlo samples, although each new Monte Carlo sample will most likely be more times consuming than corresponding ones of the brute force method.

- The first topic deals with change of variables, and is linked to the cumulative function $P(x)$ of a PDF $p(x)$. Obviously, not all integration limits go from $x = 0$ to $x = 1$, rather, in physics we are often confronted with integration domains like $x \in [0,\infty]$ or $x \in [-\infty,\infty]$ etc. Since all random number generators give numbers in the interval $x \in [0,1]$, *we need a mapping from this integration interval to the explicit one under consideration.*

- The next topic deals with the shape of the integrand itself. Let us for the sake of simplicity just assume that the integration domain is again from $x = 0$ to $x = 1$. If the function to be integrated $f(x)$ has sharp peaks and is zero or small for many values of $x \in [0, 1]$, most samples of $f(x)$ give contributions to the integral I which are negligible. As a consequence we need many N samples to have a sufficient accuracy in the region where $f(x)$ is peaked. What do we do then? We try to find a new PDF $p(x)$ chosen so as to match $f(x)$ in order to render the integrand smooth. The new PDF $p(x)$ has in turn an x domain which most likely has to be mapped from the domain of the uniform distribution.

Why care at all and not be content with just a change of variables in cases where that is needed? Below we show several examples of how to improve a Monte Carlo integration through smarter choices of PDFs which render the integrand smoother. However one classic example from quantum mechanics illustrates the need for a good sampling function.

In quantum mechanics, the probability distribution function is given by $p(x) = \Psi(x)^*\Psi(x)$, where $\Psi(x)$ is the eigenfunction arising from the solution of e.g., the time-independent Schrödinger equation. If $\Psi(x)$ is an eigenfunction, the corresponding energy eigenvalue is given by

$$H(x)\Psi(x) = E\Psi(x), \tag{9.62}$$

where $H(x)$ is the hamiltonian under consideration. The expectation value of H, assuming that the quantum mechanical PDF is normalized, is given by

$$\langle H \rangle = \int dx \Psi(x)^* H(x) \Psi(x). \tag{9.63}$$

We could insert $\Psi(x)/\Psi(x)$ right to the left of H and rewrite the last equation as

$$\langle H \rangle = \int dx \Psi(x)^* \Psi(x) \frac{H(x)}{\Psi(x)} \Psi(x), \tag{9.64}$$

or

$$\langle H \rangle = \int dx p(x) \tilde{H}(x), \tag{9.65}$$

which is on the form of an expectation value with

$$\tilde{H}(x) = \frac{H(x)}{\Psi(x)} \Psi(x). \tag{9.66}$$

The crucial point to note is that if $\Psi(x)$ is the exact eigenfunction itself with eigenvalue E, then $\tilde{H}(x)$ reduces just to the constant E and we have

$$\langle H \rangle = \int dx p(x) E = E, \tag{9.67}$$

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.

This means in turn that when do our Monte Carlo sampling, we will hopefully pick only relevant values for $\tilde{H}$.

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.

9.5.1 Change of variables

The starting point is always the uniform distribution

$$p(x)dx = \begin{cases} dx & 0 \leq x \leq 1 \\ 0 & else \end{cases} \tag{9.68}$$

with $p(x) = 1$ and satisfying

$$\int_{-\infty}^{\infty} p(x)dx = 1. \tag{9.69}$$

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

$$p(y)dy = p(x)dx, \tag{9.70}$$

which for the uniform distribution implies

$$p(y)dy = dx. \tag{9.71}$$

Let us assume that $p(y)$ is a PDF different from the uniform PDF $p(x) = 1$ with $x \in [0, 1]$. If we integrate the last expression we arrive at

$$x(y) = \int_0^y p(y')dy', \tag{9.72}$$

which is nothing but the cumulative distribution of $p(y)$, i.e.,

$$x(y) = P(y) = \int_0^y p(y')dy'. \tag{9.73}$$

This is an important result which has consequences for eventual improvements over the brute force Monte Carlo.

To illustrate this approach, let us look at some examples.

Example 1

Suppose we have the general uniform distribution

$$p(y)dy = \begin{cases} \frac{dy}{b-a} & a \leq y \leq b \\ 0 & else \end{cases} \tag{9.74}$$

If we wish to relate this distribution to the one in the interval $x \in [0,1]$ we have

$$p(y)dy = \frac{dy}{b-a} = dx, \tag{9.75}$$

and integrating we obtain the cumulative function

$$x(y) = \int_a^y \frac{dy'}{b-a}, \tag{9.76}$$

yielding

$$y = a + (b-a)x, \tag{9.77}$$

a well-known result!

Example 2, the exponential distribution

Assume that

$$p(y) = e^{-y}, \tag{9.78}$$

which is the exponential distribution, important for the analysis of e.g., radioactive decay. Again, $p(x)$ is given by the uniform distribution with $x \in [0,1]$, and with the assumption that the probability is conserved we have

$$p(y)dy = e^{-y}dy = dx, \tag{9.79}$$

which yields after integration

$$x(y) = P(y) = \int_0^y \exp(-y')dy' = 1 - \exp(-y), \tag{9.80}$$

or

$$y(x) = -ln(1-x). \tag{9.81}$$

This gives us the new random variable y in the domain $y \in [0,\infty]$ determined through the random variable $x \in [0,1]$ generated by functions like $ran0$.

This means that if we can factor out $\exp -(y)$ from an integrand we may have

$$I = \int_0^\infty F(y)dy = \int_0^\infty \exp -(y)G(y)dy \tag{9.82}$$

which we rewrite as

$$\int_0^\infty \exp(-y)G(y)dy = \int_0^\infty \frac{dx}{dy}G(y)dy \approx \frac{1}{N}\sum_{i=1}^{N} G(y(x_i)), \tag{9.83}$$

where x_i is a random number in the interval [0,1].

The algorithm for the last example is rather simple. In the function which sets up the integral, we simply need to call one of the random number generators like $ran0$, $ran1$, $ran2$ or $ran3$ in order to obtain numbers in the interval [0,1]. We obtain y by the taking the logarithm of $(1-x)$. Our calling function which sets up the new random variable y may then include statements like

```
.....
idum=-1;
x=ran0(&idum);
y=-log(1.-x);
.....
```

Exercise 9.4
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 $ran0$.

Example 3

Another function which provides an example for a PDF is

$$p(y)dy = \frac{dy}{(a+by)^n}, \tag{9.84}$$

with $n > 1$. It is normalizable, positive definite, analytically integrable and the integral is invertible, allowing thereby the expression of a new variable in terms of the old one. The integral

$$\int_0^\infty \frac{dy}{(a+by)^n} = \frac{1}{(n-1)ba^{n-1}}, \tag{9.85}$$

gives

$$p(y)dy = \frac{(n-1)ba^{n-1}}{(a+by)^n}dy, \tag{9.86}$$

which in turn gives the cumulative function

$$x(y) = P(y) = \int_0^y \frac{(n-1)ba^{n-1}}{(a+bx)^n}dy' =, \tag{9.87}$$

resulting in

$$x(y) = 1 - \frac{1}{(1+b/ay)^{n-1}}, \tag{9.88}$$

or

$$y = \frac{a}{b}\left(x^{-1/(n-1)} - 1\right). \tag{9.89}$$

With the random variable $x \in [0,1]$ generated by functions like $ran0$, we have again the appropriate random variable y for a new PDF.

Example 4, the normal distribution

For the normal distribution, expressed here as

$$g(x,y) = \exp\left(-(x^2+y^2)/2\right)dxdy. \tag{9.90}$$

it is rather difficult to find an inverse since the cumulative distribution is given by the error function $erf(x)$.

If we however switch to polar coordinates, we have for x and y

$$r = \left(x^2+y^2\right)^{1/2} \qquad \theta = tan^{-1}\frac{x}{y}, \tag{9.91}$$

resulting in

$$g(r,\theta) = r\exp\left(-r^2/2\right)drd\theta, \tag{9.92}$$

where the angle θ could be given by a uniform distribution in the region $[0,2\pi]$. Following example 1 above, this implies simply multiplying random numbers $x \in [0,1]$ by 2π. The variable r, defined for $r \in [0,\infty]$ needs to be related to to random numbers $x' \in [0,1]$. To achieve that, we introduce a new variable

$$u = \frac{1}{2}r^2, \tag{9.93}$$

and define a PDF

$$\exp\left(-u\right)du, \tag{9.94}$$

with $u \in [0,\infty]$. Using the results from example 2, we have that

$$u = -ln(1-x'), \tag{9.95}$$

where x' is a random number generated for $x' \in [0,1]$. With

$$x = rcos(\theta) = \sqrt{2u}cos(\theta), \tag{9.96}$$

and

$$y = rsin(\theta) = \sqrt{2u}sin(\theta), \tag{9.97}$$

we can obtain new random numbers x, y through

$$x = \sqrt{-2ln(1-x')}cos(\theta), \tag{9.98}$$

and

$$y = \sqrt{-2ln(1-x')}sin(\theta), \tag{9.99}$$

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);
.....
```

Exercise 9.4
Make a function $normal\_random$ which computes random numbers for the normal distribution based on random numbers generated from the function $ran0$.

9.5.2 Importance sampling

With the aid of the above variable transformations we address now one of the most widely used approaches to Monte Carlo integration, namely importance sampling.

Let us assume that $p(y)$ is a PDF whose behavior resembles that of a function F defined in a certain interval $[a, b]$. The normalization condition is

$$\int_a^b p(y)dy = 1. \tag{9.100}$$

We can rewrite our integral as

$$I = \int_a^b F(y)dy = \int_a^b p(y)\frac{F(y)}{p(y)}dy. \tag{9.101}$$

This integral resembles our discussion on the evaluation of the energy for a quantum mechanical system in Eq. (9.64).

Since random numbers are generated for the uniform distribution $p(x)$ with $x \in [0, 1]$, we need to perform a change of variables $x \to y$ through

$$x(y) = \int_a^y p(y')dy', \tag{9.102}$$

where we used

$$p(x)dx = dx = p(y)dy. \tag{9.103}$$

If we can invert $x(y)$, we find $y(x)$ as well.

With this change of variables we can express the integral of Eq. (9.101) as

$$I = \int_a^b p(y)\frac{F(y)}{p(y)}dy = \int_a^b \frac{F(y(x))}{p(y(x))}dx, \tag{9.104}$$

meaning that a Monte Carlo evalutaion of the above integral gives

$$\int_a^b \frac{F(y(x))}{p(y(x))}dx = \frac{1}{N}\sum_{i=1}^{N}\frac{F(y(x_i))}{p(y(x_i))}. \tag{9.105}$$

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 standard deviation is now with the definition

$$\tilde{F} = \frac{F(y(x))}{p(y(x))}, \tag{9.106}$$

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}\left(\tilde{F}\right)^2 - \left(\frac{1}{N}\sum_{i=1}^{N}\tilde{F}\right)^2. \tag{9.107}$$

The algorithm for this procedure is

- Use the uniform distribution to find the random variable y in the interval [0,1]. $p(x)$ is auser provided PDF.
- Evaluate thereafter

$$I = \int_a^b F(x)dx = \int_a^b p(x)\frac{F(x)}{p(x)}dx, \tag{9.108}$$

by rewriting

$$\int_a^b p(x)\frac{F(x)}{p(x)}dx = \int_a^b \frac{F(x(y))}{p(x(y))}dy, \tag{9.109}$$

since

$$\frac{dy}{dx} = p(x). \tag{9.110}$$

- Perform then a Monte Carlo sampling for

$$\int_a^b \frac{F(x(y))}{p(x(y))}dy, \approx \frac{1}{N}\sum_{i=1}^{N}\frac{F(x(y_i))}{p(x(y_i))}, \tag{9.111}$$

with $y_i \in [0,1]$,

- and evaluate the variance as well according to Eq. (9.107).

Exercise 9.5

(a) Calculate the integral

$$I = \int_0^1 e^{-x^2} dx,$$

using brute force Monte Carlo with $p(x) = 1$ and importance sampling with $p(x) = ae^{-x}$ where a is a constant.

(b) Calculate the integral

$$I = \int_0^\pi \frac{1}{x^2 + cos^2(x)} dx,$$

with $p(x) = ae^{-x}$ where a is a constant. Determine the value of a which minimizes the variance.

9.5.3 Acceptance-Rejection method

This is rather simple and appealing method after von Neumann. Assume that we are looking at an interval $x \in [a, b]$, this being the domain of the PDF $p(x)$. Suppose also that the largest value our distribution function takes in this interval is M, that is

$$p(x) \leq M \qquad x \in [a, b]. \tag{9.112}$$

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

$$p(x) \geq s, \tag{9.113}$$

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.

9.6 Monte Carlo integration of multidimensional integrals

When we deal with multidimensional integrals of the form

$$I = \int_0^1 dx_1 \int_0^1 dx_2 \ldots \int_0^1 dx_d g(x_1, \ldots, x_d), \tag{9.114}$$

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 $g(x_1, \ldots, x_d)$ to

$$g(x_1, \ldots, x_d) \rightarrow g(a_1 + (b_1 - a_1)t_1, \ldots, a_d + (b_d - a_d)t_d).$$

As an example, consider the following sixth-dimensional integral

$$\int_{-\infty}^{\infty} \mathbf{dxdy}\, g(\mathbf{x}, \mathbf{y}), \tag{9.115}$$

where

$$g(\mathbf{x}, \mathbf{y}) = \exp\left(-\mathbf{x}^2 - \mathbf{y}^2 - (\mathbf{x} - \mathbf{y})^2/2\right), \tag{9.116}$$

with $d = 6$.

We can solve this integral by employing our brute 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

$$\frac{1}{\sqrt{\pi}} \exp\left(-x^2\right), \tag{9.117}$$

and through

$$\pi^3 \int \prod_{i=1}^{6} \left(\frac{1}{\sqrt{\pi}} \exp\left(-x_i^2\right)\right) \exp\left(-(\mathbf{x} - \mathbf{y})^2/2\right) dx_1 \ldots . dx_6, \tag{9.118}$$

we can rewrite our integral as

$$\int f(x_1, \ldots, x_d) F(x_1, \ldots, x_d) \prod_{i=1}^{6} dx_i, \tag{9.119}$$

where f is the gaussian distribution.

Below we list two codes, one for the brute force integration and the other employing importance sampling with a gaussian distribution.

9.6.1 Brute force integration

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace std;

double brute_force_MC(double *);
//      Main function begins here
int main()
{
      int n;
      double x[6], y, fx;
      double int_mc = 0.;  double variance = 0.;
      double sum_sigma= 0. ; long idum=-1;
      double length=5.; // we fix the max size of the box to L=5
      double volume=pow((2*length),6);
      cout << "Read in the number of Monte-Carlo samples" << endl;
      cin >> n;
//    evaluate the integral with importance sampling
      for ( int i = 1;  i <= n; i++){
//    x[] contains the random numbers for all dimensions
        for (int j = 0; j< 6; j++) {
            x[j]=-length+2*length*ran0(&idum);
        }
        fx=brute_force_MC(x);
        int_mc += fx;
        sum_sigma += fx*fx;
      }
      int_mc = int_mc/((double) n );
      sum_sigma = sum_sigma/((double) n );
      variance=sum_sigma-int_mc*int_mc;
//    final output
       cout << setiosflags(ios::showpoint | ios::uppercase);
       cout << " Monte carlo result= " << setw(10) << setprecision(8)
            << volume*int_mc;
       cout << " Sigma= " << setw(10) << setprecision(8) << volume*sqrt
          (variance/((double) n )) << endl;
      return 0;
}  // end of main program

// this function defines the integrand to integrate

double  brute_force_MC(double *x)
```

```
{
   double a = 1.; double b = 0.5;
// evaluate the different terms of the exponential
   double xx=x[0]*x[0]+x[1]*x[1]+x[2]*x[2];
   double yy=x[3]*x[3]+x[4]*x[4]+x[5]*x[5];
   double xy=pow((x[0]-x[3]),2)+pow((x[1]-x[4]),2)+pow((x[2]-x[5]),2);
   return exp(-a*xx-a*yy-b*xy);
} // end function for the integrand
```

9.6.2 Importance sampling

This code includes a call to the function $normal\_random$, which produces random numbers from a gaussian distribution. .

```
// importance sampling with gaussian deviates
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace std;

double gaussian_MC(double *);
double gaussian_deviate(long *);
//     Main function begins here
int main()
{
     int n;
     double x[6], y, fx;
     cout << "Read in the number of Monte-Carlo samples" << endl;
     cin >> n;
     double int_mc = 0.;  double variance = 0.;
     double sum_sigma= 0. ; long idum=-1;
     double length=5.; // we fix the max size of the box to L=5
     double volume=pow(acos(-1.),3.);
     double sqrt2 = 1./sqrt(2.);
//   evaluate the integral with importance sampling
     for ( int i = 1;  i <= n; i++){
//   x[] contains the random numbers for all dimensions
       for (int j = 0; j < 6; j++) {
         x[j] = gaussian_deviate(&idum)*sqrt2;
       }
       fx=gaussian_MC(x);
       int_mc += fx;
       sum_sigma += fx*fx;
     }
```

```
      int_mc = int_mc/((double) n );
      sum_sigma = sum_sigma/((double) n );
      variance=sum_sigma-int_mc*int_mc;
//    final output
       cout << setiosflags(ios::showpoint | ios::uppercase);
       cout << " Monte carlo result= " << setw(10) << setprecision(8)
            << volume*int_mc;
       cout << " Sigma= " << setw(10) << setprecision(8) << volume*sqrt
          (variance/((double) n )) << endl;
      return 0;
}  // end of main program

// this function defines the integrand to integrate

double  gaussian_MC(double *x)
{
   double a = 0.5;
// evaluate the different terms of the exponential
   double xy=pow((x[0]-x[3]),2)+pow((x[1]-x[4]),2)+pow((x[2]-x[5]),2);
   return exp(-a*xy);
} // end function for the integrand

// random numbers with gaussian distribution
double gaussian_deviate(long * idum)
{
  static int iset = 0;
  static double gset;
  double fac, rsq, v1, v2;

  if ( idum < 0) iset =0;
  if (iset == 0) {
    do {
      v1 = 2.*ran0(idum) -1.0;
      v2 = 2.*ran0(idum) -1.0;
      rsq = v1*v1+v2*v2;
    } while (rsq >= 1.0 || rsq == 0.);
    fac = sqrt(-2.*log(rsq)/rsq);
    gset = v1*fac;
    iset = 1;
    return v2*fac;
  } else {
    iset =0;
    return gset;
  }
} // end function for gaussian deviates
```

Chapter 10

Random walks and the Metropolis algorithm

10.1 Motivation

In the previous chapter we discussed technical aspects of Monte Carlo integration such as algorithms for generating random numbers and integration of multidimensional integrals. The latter topic served to illustrate two key topics in Monte Carlo simulations, namely a proper selection of variables and importance sampling. An intelligent selection of variables, good sampling techniques and guiding functions can be crucial for the outcome of our Monte Carlo simulations. Examples of this will be demonstrated in the chapters on statistical and quantum physics applications. Here we make a detour however from this main area of applications. The focus is on diffusion and random walks. The rationale for this is that the tricky part of an actual Monte Carlo simulation resides in the appropriate selection of random states, and thereby numbers, according to the probability distribution (PDF) at hand. With appropriate there is however much more to the picture than meets the eye.

Suppose our PDF is given by the well-known normal distribution. Think of for example the velocity distribution of an ideal gas in a container. In our simulations we could then accept or reject new moves with a probability proportional to the normal distribution. This would parallel our example on the sixth dimensional integral in the previous chapter. However, in this case we would end up rejecting basically all moves since the probabilities are exponentially small in most cases. The result would be that we barely moved from the initial position. Our statistical averages would then be significantly biased and most likely not very reliable.

Instead, all Monte Carlo schemes used are based on Markov processes in order to generate new random states. A Markov process is a random walk with a selected probability for making a move. The new move is independent of the previous history of the system. The Markov process is used repeatedly in Monte Carlo simulations in order to generate new random states. The reason for choosing a Markov process is that when it is run for a long enough time starting with a random state, we will eventually reach the most likely state of the system. In thermodynamics, this means that after a certain number of Markov processes we reach an equilibrium distribution.

This mimicks the way a real system reaches its most likely state at a given temperature of the surroundings.

To reach this distribution, the Markov process needs to obey two important conditions, that of ergodicity and detailed balance. These conditions impose then constraints on our algorithms for accepting or rejecting new random states. The Metropolis algorithm discussed here abides to both these constraints and is discussed in more detail in Section 10.4. 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 10.3 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.

10.2 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.

10.2.1 Diffusion equation

From experiment there are strong indications that the flux of particles $j(x,t)$, viz., the number of particles passing x at a time t is proportional to the gradient of $w(x,t)$. This proportionality is expressed mathematically through

$$j(x,t) = -D\frac{\partial w(x,t)}{\partial x}, \tag{10.1}$$

where D is the so-called diffusion constant, with dimensionality length^2 per time. If the number of particles is conserved, we have the continuity equation

$$\frac{\partial j(x,t)}{\partial x} = -\frac{\partial w(x,t)}{\partial t}, \tag{10.2}$$

which leads to

$$\frac{\partial w(x,t)}{\partial t} = D\frac{\partial^2 w(x,t)}{\partial x^2}, \tag{10.3}$$

which is the diffusion equation in one dimension.

With the probability distribution function $w(x,t)dx$ we can use the results from the previous chapter to compute expectation values such as the mean distance

$$\langle x(t)\rangle = \int_{-\infty}^{\infty} xw(x,t)dx, \tag{10.4}$$

or

$$\langle x^2(t)\rangle = \int_{-\infty}^{\infty} x^2 w(x,t)dx, \tag{10.5}$$

which allows for the computation of the variance $\sigma^2 = \langle x^2(t)\rangle - \langle x(t)\rangle^2$. Note well that these expectation values are time-dependent. In a similar way we can also define expectation values of functions $f(x,t)$ as

$$\langle f(x,t)\rangle = \int_{-\infty}^{\infty} f(x,t)w(x,t)dx. \tag{10.6}$$

Since $w(x,t)$ is now treated as a PDF, it needs to obey the same criteria as discussed in the previous chapter. However, the normalization condition

$$\int_{-\infty}^{\infty} w(x,t)dx = 1 \tag{10.7}$$

imposes significant constraints on $w(x,t)$. These are

$$w(x=\pm\infty,t) = 0 \qquad \frac{\partial^n w(x,t)}{\partial x^n}|_{x=\pm\infty} = 0, \tag{10.8}$$

implying that when we study the time-derivative $\partial\langle x(t)\rangle/\partial t$, we obtain after integration by parts and using Eq. (10.3)

$$\frac{\partial\langle x\rangle}{\partial t} = \int_{-\infty}^{\infty} x\frac{\partial w(x,t)}{\partial t}dx = D\int_{-\infty}^{\infty} x\frac{\partial^2 w(x,t)}{\partial x^2}dx, \tag{10.9}$$

leading to

$$\frac{\partial\langle x\rangle}{\partial t} = Dx\frac{\partial w(x,t)}{\partial x}|_{x=\pm\infty} - D\int_{-\infty}^{\infty}\frac{\partial w(x,t)}{\partial x}dx, \tag{10.10}$$

implying that

$$\frac{\partial\langle x\rangle}{\partial t} = 0. \tag{10.11}$$

This means in turn that $\langle x\rangle$ is independent of time. If we choose the initial position $x(t=0)=0$, the average displacement $\langle x\rangle = 0$. If we link this discussion to a random walk in one dimension with equal probability of jumping to the left or right and with an initial position $x = 0$, then our probability distribution remains centered around $\langle x\rangle = 0$ as function of time. However, the variance is not necessarily 0. Consider first

$$\frac{\partial\langle x^2\rangle}{\partial t} = Dx^2\frac{\partial w(x,t)}{\partial x}|_{x=\pm\infty} - 2D\int_{-\infty}^{\infty} x\frac{\partial w(x,t)}{\partial x}dx, \tag{10.12}$$

where we have performed an integration by parts as we did for $\frac{\partial\langle x\rangle}{\partial t}$. A further integration by parts results in

$$\frac{\partial\langle x^2\rangle}{\partial t} = -Dxw(x,t)|_{x=\pm\infty} + 2D\int_{-\infty}^{\infty} w(x,t)dx = 2D, \tag{10.13}$$

leading to

$$\langle x^2\rangle = 2Dt, \tag{10.14}$$

and the variance as

$$\langle x^2\rangle - \langle x\rangle^2 = 2Dt. \tag{10.15}$$

The root mean square displacement after a time t is then

$$\sqrt{\langle x^2\rangle - \langle x\rangle^2} = \sqrt{2Dt}. \tag{10.16}$$

This should be contrasted to the displacement of a free particle with initial velocity v_0. In that case the distance from the initial position after a time t is $x(t) = vt$ whereas for a diffusion process the root mean square value is $\sqrt{\langle x^2\rangle - \langle x\rangle^2} \propto \sqrt{t}$. Since diffusion is strongly linked with random walks, we could say that a random walker escapes much more slowly from the starting point than would a free particle. We can vizualize the above in the following figure. In Fig. 10.1 we have assumed that our distribution is given by a normal distribution with variance $\sigma^2 = 2Dt$, centered at $x = 0$. The distribution reads

$$w(x,t)dx = \frac{1}{\sqrt{4\pi Dt}}\exp{(-\frac{x^2}{4Dt})}dx. \tag{10.17}$$

At a time $t = 2$s the new variance is $\sigma^2 = 4D$s, implying that the root mean square value is $\sqrt{\langle x^2\rangle - \langle x\rangle^2} = 2\sqrt{D}$. At a further time $t = 8$ we have $\sqrt{\langle x^2\rangle - \langle x\rangle^2} = 4\sqrt{D}$. While time has elapsed by a factor of 4, the root mean square has only changed by a factor of 2. Fig. 10.1 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.

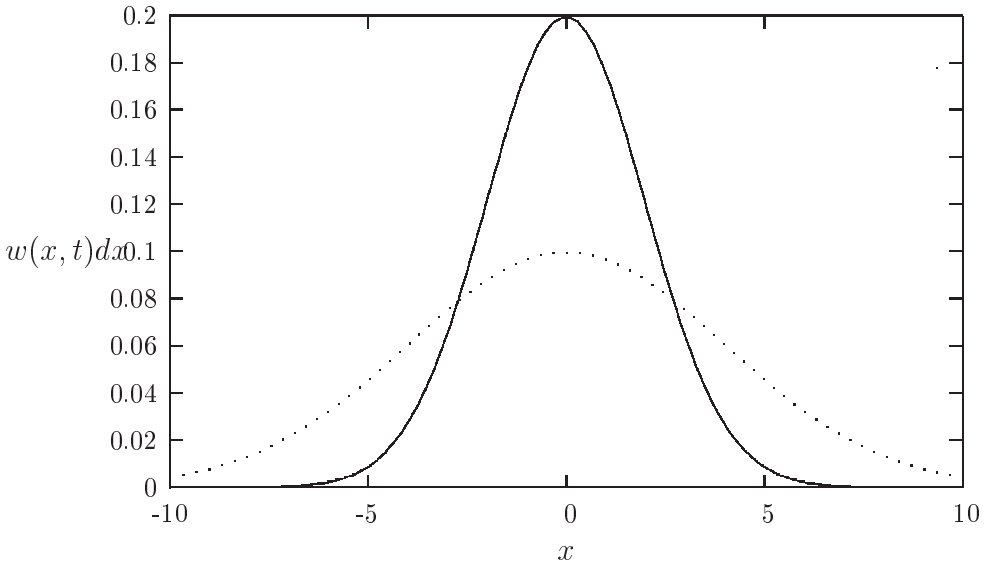


Figure 10.1: Time development of a normal distribution with variance $\sigma^2 = 2Dt$ and with $D = 1\text{m}^2/\text{s}$. The solid line represents the distribution at $t = 2\text{s}$ while the dotted line stands for $t = 8\text{s}$.

Figure 10.2: One-dimensional walker which can jump either to the left or to the right. Every step has length $\Delta x = l$.

10.2.2 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. 10.2. The walker can then jump, with the above probabilities, either to the left or to the right for each time step. Note that in principle we could also have the possibility that the walker remains in the same position. This is not implemented in this example. Every step has length $\Delta x = l$. Time is discretized and we have a jump either to the left or to the right at every time step. Let us now assume that we have equal probabilities for jumping to the left or to the right, i.e., $L = R = 1/2$. The average displacement after n time steps is

$$\langle x(n)\rangle = \sum_i^n \Delta x_i = 0 \qquad \Delta x_i = \pm l, \tag{10.18}$$

since we have an equal probability of jumping either to the left or to right. The value of $\langle x(n)^2\rangle$ is

$$\langle x(n)^2\rangle = \left(\sum_i^n \Delta x_i\right)^2 = \sum_i^n \Delta x_i^2 + \sum_{i\neq j}^N \Delta x_i \Delta x_j = l^2 N. \tag{10.19}$$

For many enough steps the non-diagonal contribution is

$$\sum_{i\neq j}^N \Delta x_i \Delta x_j = 0, \tag{10.20}$$

since $\Delta x_{i,j} = \pm l$. The variance is then

$$\langle x(n)^2\rangle - \langle x(n)\rangle^2 = l^2 n. \tag{10.21}$$

It is also rather straightforward to compute the variance for $L \neq R$. The result is

$$\langle x(n)^2\rangle - \langle x(n)\rangle^2 = 4LRl^2 n. \tag{10.22}$$

In Eq. (10.21) the variable n represents the number of time steps. If we define $n = t/\Delta t$, we can then couple the variance result from a random walk in one dimension with the variance from the diffusion equation of Eq. (10.15) by defining the diffusion constant as

$$D = \frac{l^2}{\Delta t}. \tag{10.23}$$

In the next section we show in detail that this is the case.

The program below demonstrates the simplicity of the one-dimensional random walk algorithm. It is straightforward to extend this program to two or three dimensions as well. The input is the number of time steps, the probability for a move to the left or to the right and the total number of Monte Carlo samples. It computes the average displacement and the variance for one random walker for a given number of Monte Carlo samples. Each sample is thus to be considered as one experiment with a given number of walks. The interesting part of the algorithm is described in the function mc_sampling. The other functions read or write the results from screen or file and are similar in structure to programs discussed previously. The main program reads the name of the output file from screen and sets up the arrays containing the walker's position after a given number of steps.

programs/chap10/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. 10.3 shows that the variance increases linearly as function of the number of time steps, as expected from the analytic results. Similarly, the mean displacement in Fig. 10.4 oscillates around zero.

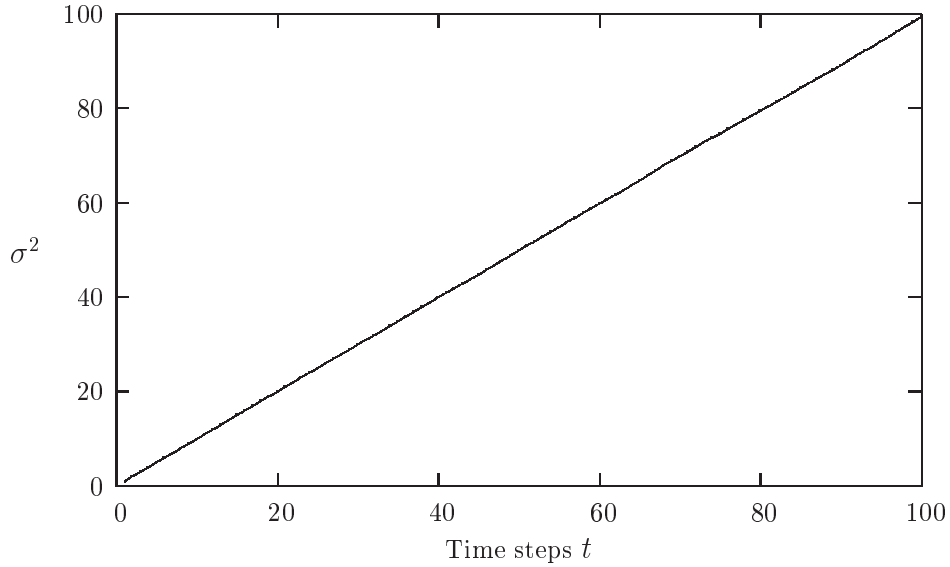


Figure 10.3: Time development of σ^2 for a random walker. 100000 Monte Carlo samples were used with the function ran1 and a seed set to -1.

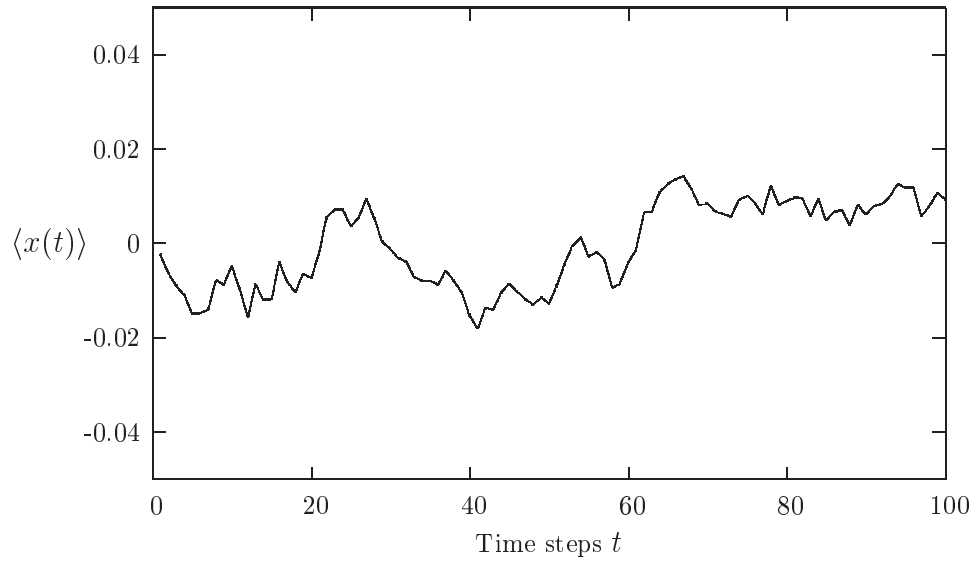


Figure 10.4: Time development of $\langle x(t) \rangle$ for a random walker. 100000 Monte Carlo samples were used with the function ran1 and a seed set to -1.

Exercise 10.1
Extend the above program 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.

10.3 Microscopic derivation of the diffusion equation

When solving partial differential equations such as the diffusion equation numerically, the derivatives are always discretized. Recalling our discussions from Chapter 3, we can rewrite the time derivative as

$$\frac{\partial w(x,t)}{\partial t} \approx \frac{w(i,n+1)+w(i,n)}{\Delta t}, \tag{10.24}$$

whereas the gradient is approximated as

$$D\frac{\partial^2 w(x,t)}{\partial x^2} \approx D\frac{w(i+1,n)+w(i-1,n)-w(i,n)}{(\Delta x)^2}, \tag{10.25}$$

resulting in the discretized diffusion equation

$$\frac{w(i,n+1)+w(i,n)}{\Delta t} = D\frac{w(i+1,n)+w(i-1,n)-w(i,n)}{(\Delta x)^2}, \tag{10.26}$$

where n represents a given time step and i a step in the x-direction. We will come back to the solution of such equations in our chapter on partial differential equations, see Chapter 16. 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.

10.3.1 Discretized diffusion equation and Markov chains

A Markov process allows in principle for a microscopic description of Brownian motion. As with the random walk studied in the previous section, we consider a particle which moves along the x-axis in the form of a series of jumps with step length $\Delta x = l$. Time and space are discretized and the subsequent moves are statistically indenpendent, i.e., the new move depends only on the previous step and not on the results from earlier trials. We start at a position $x = jl = j\Delta x$ and move to a new position $x = i\Delta x$ during a step $\Delta t = \epsilon$, where $i \geq 0$ and $j \geq 0$ are integers. The original probability distribution function (PDF) of the particles is given by $w_i(t = 0)$ where i refers to a specific position on the grid in Fig. 10.2, with $i = 0$ representing $x = 0$. The function $w_i(t = 0)$ is now the discretized version of $w(x,t)$. We can regard the discretized PDF as a

vector. For the Markov process we have a transition probability from a position $x = jl$ to a position $x = il$ given by

$$W_{ij}(\epsilon) = W(il - jl, \epsilon) = \begin{cases} \frac{1}{2} & |i-j| = 1 \\ 0 & \text{else} \end{cases} \tag{10.27}$$

We call W_{ij} for the transition probability and we can represent it, see below, as a matrix. Our new PDF $w_i(t = \epsilon)$ is now related to the PDF at $t = 0$ through the relation

$$w_i(t = \epsilon) = W(j \rightarrow i)w_j(t = 0). \tag{10.28}$$

This equation represents the discretized time-development of an original PDF. It is a microscopic way of representing the process shown in Fig. 10.1. Since both W and w represent probabilities, they have to be normalized, i.e., we require that at each time step we have

$$\sum_i w_i(t) = 1, \tag{10.29}$$

and

$$\sum_j W(j \rightarrow i) = 1. \tag{10.30}$$

The further constraints are $0 \leq W_{ij} \leq 1$ and $0 \leq w_j \leq 1$. Note that the probability for remaining at the same place is in general not necessarily equal zero. In our Markov process we allow only for jumps to the left or to the right.

The time development of our initial PDF can now be represented through the action of the transition probability matrix applied n times. At a time $t_n = n\epsilon$ our initial distribution has developed into

$$w_i(t_n) = \sum_j W_{ij}(t_n)w_j(0), \tag{10.31}$$

and defining

$$W(il - jl, n\epsilon) = (W^n(\epsilon))_{ij} \tag{10.32}$$

we obtain

$$w_i(n\epsilon) = \sum_j (W^n(\epsilon))_{ij} w_j(0), \tag{10.33}$$

or in matrix form

$$\hat{w(n\epsilon)} = \hat{W}^n(\epsilon)\hat{w}(0). \tag{10.34}$$

The matrix $\hat{W}$ can be written in terms of two matrices

$$\hat{W} = \frac{1}{2}\left(\hat{L} + \hat{R}\right), \tag{10.35}$$

where $\hat{L}$ and $\hat{R}$ represent the transition probabilities for a jump to the left or the right, respectively. For a 4×4 case we could write these matrices as

$$\hat{R} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \tag{10.36}$$

and

$$\hat{L} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \tag{10.37}$$

However, in principle these are infinite dimensional matrices since the number of time steps are very large or infinite. For the infinite case we can write these matrices $R_{ij} = \delta_{i,(j+1)}$ and $L_{ij} = \delta_{(i+1),j}$, implying that

$$\hat{L}\hat{R} = \hat{R}\hat{L} = 1, \tag{10.38}$$

and

$$\hat{L} = \hat{R}^{-1} \tag{10.39}$$

To see that $\hat{L}\hat{R} = \hat{R}\hat{L} = 1$, perform e.g., the matrix multiplication

$$\hat{L}\hat{R} = \sum_k \hat{L}_{ik}\hat{R}_{kj} = \sum_k \delta_{(i+1),k}\delta_{k,(j+1)} = \delta_{i+1,j+1} = \delta_{i,j}, \tag{10.40}$$

and only the diagonal matrix elements are different from zero.

For the first time step we have thus

$$\hat{W} = \frac{1}{2}\left(\hat{L} + \hat{R}\right), \tag{10.41}$$

and using the properties in Eqs. (10.38) and (10.39) we have after two time steps

$$\hat{W}^2(2\epsilon) = \frac{1}{4}\left(\hat{L}^2 + \hat{R}^2 + 2\hat{R}\hat{L}\right), \tag{10.42}$$

and similarly after three time steps

$$\hat{W}^3(3\epsilon) = \frac{1}{8}\left(\hat{L}^3 + \hat{R}^3 + 3\hat{R}\hat{L}^2 + 3\hat{R}^2\hat{L}\right). \tag{10.43}$$

Using the binomial formula

$$\sum_{k=0}^{n} \begin{pmatrix} n \\ k \end{pmatrix} \hat{a}^k \hat{b}^{n-k} = (a+b)^n, \tag{10.44}$$

we have that the transition matrix after n time steps can be written as

$$\hat{W}^n(n\epsilon)) = \frac{1}{2^n}\sum_{k=0}^{n} \begin{pmatrix} n \\ k \end{pmatrix} \hat{R}^k \hat{L}^{n-k}, \tag{10.45}$$

or

$$\hat{W}^n(n\epsilon)) = \frac{1}{2^n}\sum_{k=0}^{n} \begin{pmatrix} n \\ k \end{pmatrix} \hat{L}^{n-2k} = \frac{1}{2^n}\sum_{k=0}^{n} \begin{pmatrix} n \\ k \end{pmatrix} \hat{R}^{2k-n}, \tag{10.46}$$

and using $R^m_{ij} = \delta_{i,(j+m)}$ and $L^m_{ij} = \delta_{(i+m),j}$ we arrive at

$$W(il - jl, n\epsilon) = \begin{cases} \frac{1}{2^n} \begin{pmatrix} n \\ \frac{1}{2}(n+i-j) \end{pmatrix} & |i-j| \leq n \\ 0 & \text{else} \end{cases}, \tag{10.47}$$

and $n + i - j$ has to be an even number. We note that the transition matrix for a Markov process has three important properties:

- It depends only on the difference in space $i - j$, it is thus homogenous in space.
- It is also isotropic in space since it is unchanged when we go from (i, j) to $(-i, -j)$.
- It is homogenous in time since it depends only the difference between the initial time and final time.

If we place the walker at $x = 0$ at $t = 0$ we can represent the initial PDF with $w_i(0) = \delta_{i,0}$. Using Eq. (10.34) we have

$$w_i(n\epsilon) = \sum_j (W^n(\epsilon))_{ij} w_j(0) = \sum_j \frac{1}{2^n} \begin{pmatrix} n \\ \frac{1}{2}(n+i-j) \end{pmatrix} \delta_{j,0}, \tag{10.48}$$

resulting in

$$w_i(n\epsilon) = \frac{1}{2^n} \begin{pmatrix} n \\ \frac{1}{2}(n+i) \end{pmatrix} \qquad |i| \leq n \tag{10.49}$$

Using the recursion relation for the binomials

$$\begin{pmatrix} n+1 \\ \frac{1}{2}(n+1+i)) \end{pmatrix} = \begin{pmatrix} n \\ \frac{1}{2}(n+i+1) \end{pmatrix} + \begin{pmatrix} n \\ \frac{1}{2}(n+i) - 1 \end{pmatrix} \tag{10.50}$$

we obtain, defining $x = il$, $t = n\epsilon$ and setting

$$w(x,t) = w(il, n\epsilon) = w_i(n\epsilon), \tag{10.51}$$

$$w(x, t+\epsilon) = \frac{1}{2} w(x+l, t) + \frac{1}{2} w(x-l, t), \tag{10.52}$$

and adding and subtracting $w(x,t)$ and multiplying both sides with l^2/ϵ we have

$$\frac{w(x,t+\epsilon) - w(x,t)}{\epsilon} = \frac{l^2}{2\epsilon} \frac{w(x+l,t) - 2w(x,t) + w(x-l,t)}{l^2}, \tag{10.53}$$

and identifying $D = l^2/2\epsilon$ and letting $l = \Delta x$ and $\epsilon = \Delta t$ we see that this is nothing but the discretized version of the diffusion equation. Taking the limits $\Delta x \to 0$ and $\Delta t \to 0$ we recover

$$\frac{\partial w(x,t)}{\partial t} = D \frac{\partial^2 w(x,t)}{\partial x^2},$$

the diffusion equation.

10.3.2 Continuous equations

Hitherto we have considered discretized versions of all equations. Our initial probability distribution function was then given by

$$w_i(0) = \delta_{i,0},$$

and its time-development after a given time step $\Delta t = \epsilon$ is

$$w_i(t) = \sum_j W(j \to i) w_j(t = 0).$$

The continuous analog to $w_i(0)$ is

$$w(\mathbf{x}) \to \delta(\mathbf{x}), \tag{10.54}$$

where we now have generalized the one-dimensional position x to a generic-dimensional vector $\mathbf{x}$. The Kroenecker δ function is replaced by the δ distribution function $\delta(\mathbf{x})$ at $t = 0$.

The transition from a state j to a state i is now replaced by a transition to a state with position $\mathbf{y}$ from a state with position $\mathbf{x}$. The discrete sum of transition probabilities can then be replaced by an integral and we obtain the new distribution at a time $t + \Delta t$ as

$$w(\mathbf{y}, t + \Delta t) = \int W(\mathbf{y}, \mathbf{x}, \Delta t) w(\mathbf{x}, t) d\mathbf{x}, \tag{10.55}$$

and after m time steps we have

$$w(\mathbf{y}, t + m\Delta t) = \int W(\mathbf{y}, \mathbf{x}, m\Delta t) w(\mathbf{x}, t) d\mathbf{x}. \tag{10.56}$$

When equilibrium is reached we have

$$w(\mathbf{y}) = \int W(\mathbf{y}, \mathbf{x}, t) w(\mathbf{x}) d\mathbf{x}. \tag{10.57}$$

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

$$w(\mathbf{x}, t) = \int_{-\infty}^{\infty} d\mathbf{k} \exp(i\mathbf{k}\mathbf{x}) \tilde{w}(\mathbf{k}, t), \tag{10.58}$$

and using the definition of the δ-function

$$\delta(\mathbf{x}) = \frac{1}{2\pi} \int_{-\infty}^{\infty} d\mathbf{k} \exp(i\mathbf{k}\mathbf{x}), \tag{10.59}$$

we see that

$$\tilde{w}(\mathbf{k}, 0) = 1/2\pi. \tag{10.60}$$

We can then use the Fourier-transformed diffusion equation

$$\frac{\partial \tilde{w}(\mathbf{k}, t)}{\partial t} = -D\mathbf{k}^2 \tilde{w}(\mathbf{k}, t), \tag{10.61}$$

with the obvious solution

$$\tilde{w}(\mathbf{k},t) = \tilde{w}(\mathbf{k},0)\exp\left[-(D\mathbf{k}^2t)\right) = \frac{1}{2\pi}\exp\left[-(D\mathbf{k}^2t)\right]. \tag{10.62}$$

Using Eq. (10.58) we obtain

$$w(\mathbf{x},t) = \int_{-\infty}^{\infty} d\mathbf{k}\exp\left[i\mathbf{k}\mathbf{x}\right]\frac{1}{2\pi}\exp\left[-(D\mathbf{k}^2t)\right] = \frac{1}{\sqrt{4\pi Dt}}\exp\left[-(\mathbf{x}^2/4Dt)\right], \tag{10.63}$$

with the normalization condition

$$\int_{-\infty}^{\infty} w(\mathbf{x},t)d\mathbf{x} = 1. \tag{10.64}$$

It is rather easy to verify by insertion that Eq. (10.63) 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. (10.47). The results from the central limit theorem, see Sect. **??**, state that transition probability in the limit $n \rightarrow \infty$ converges to the normal distribution. It is then possible to show that

$$W(il - jl, n\epsilon) \rightarrow W(\mathbf{y},\mathbf{x},\Delta t) = \frac{1}{\sqrt{4\pi D\Delta t}}\exp\left[-((\mathbf{y}-\mathbf{x})^2/4D\Delta t)\right], \tag{10.65}$$

and that it satisfies the normalization condition and is itself a solution to the diffusion equation.

10.3.3 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 of this chapter. The array probability extends from −number_walks to +number_walks

programs/chap10/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. 10.5 shows the resulting probability distribution after n steps We see from Fig. 10.5 that the probability distribution function resembles a normal distribution.

Exercise 10.2
Use the above program and try to fit the computed probability distribution with a normal distribution using your calculated values of σ^2 and $\langle x \rangle$.

10.4 The Metropolis algorithm and detailed balance

An important condition we require that our Markov chain should satisfy is that of detailed balance. In statistical physics this condition ensures that it is e.g., the Boltzmann distribution which is generated when equilibrium is reached. The definition for being in equilibrium is that the rates at which a system makes a transition to or from a given state i have to be equal, that is

$$\sum_j W(j \rightarrow i) w_j = \sum_i W(i \rightarrow j) w_i. \tag{10.66}$$

Another way of stating that a Markow process has reached equilibrium is

$$\mathbf{w}(t = \infty) = \mathbf{W}\mathbf{w}(t = \infty). \tag{10.67}$$

However, the condition that the rates should equal each other is in general not sufficient to guarantee that we, after many simulations, generate the correct distribution. We therefore introduce an additional condition, namely that of detailed balance

$$W(j \rightarrow i) w_j = W(i \rightarrow j) w_i. \tag{10.68}$$

Satisfies the detailed balance condition. At equilibrium detailed balance gives thus

$$\frac{W(j \rightarrow i)}{W(i \rightarrow j)} = \frac{w_i}{w_j}. \tag{10.69}$$

We introduce the Boltzmann distribution

$$w_i = \frac{\exp\left(-\beta(E_i)\right.}{Z}, \tag{10.70}$$

which states that probability of finding the system in a state i with energy E_i at an inverse temperature $\beta = 1/k_B T$ is $w_i \propto \exp\left(-\beta(E_i)\right.$. The denominator Z is a normalization constant

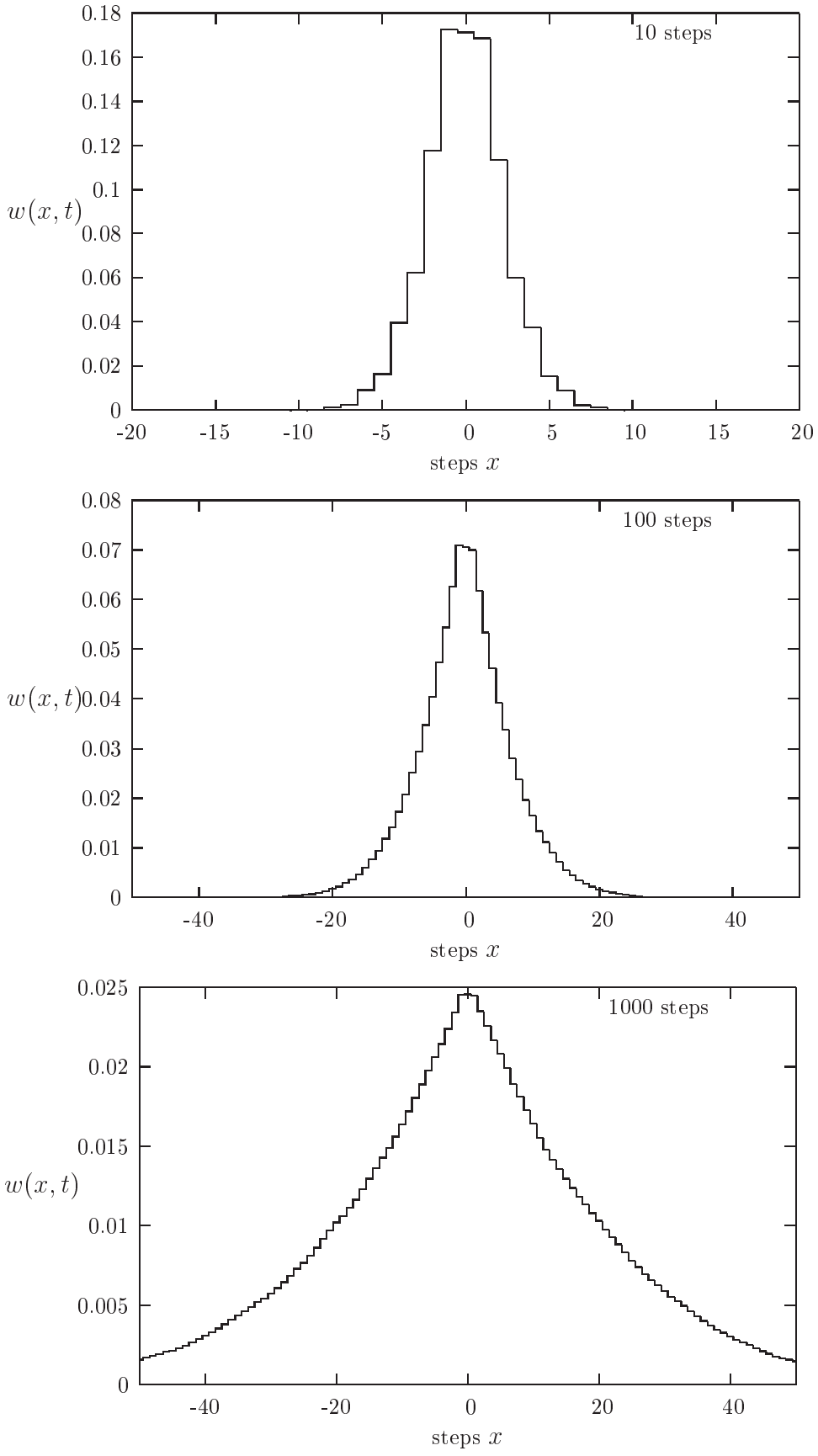


Figure 10.5: Probability distribution for one walker after 10, 100 and 1000 steps.

which ensures that the sum of all probabilities is normalized to one. It is defined as the sum of probabilities over all microstates j of the system

$$Z = \sum_j \exp\left(-\beta(E_i\right). \tag{10.71}$$

From the partition function we can in principle generate all interesting quantities for a given system in equilibrium with its surroundings at a temperature T. This is demonstrated in the next chapter.

With the probability distribution given by the Boltzmann distribution we are now in the position where we can generate expectation values for a given variable A through the definition

$$\langle A\rangle = \sum_j A_j w_j = \frac{\sum_j A_j \exp\left(-\beta(E_j\right)}{Z}. \tag{10.72}$$

In general, most systems have an infinity of microstates making thereby the computation of Z practically impossible and a brute force Monte Carlo calculation over a given number of randomly selected microstates may therefore not yield those microstates which are important at equilibrium. To select the most important contributions we need to use the condition for detailed balance. Since this is just given by the ratios of probabilities, we never need to evaluate the partition function Z. For the Boltzmann distribution, detailed balance results in

$$\frac{w_i}{w_j} = \exp\left(-\beta(E_i - E_j)\right). \tag{10.73}$$

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 in, when equilibrium is established, in the Boltzmann distribution. Else, our algorithm for selecting microstates has to be wrong.

Since we do not know the analytic form of the transition rate, we are free to model it as

$$W(i \to j) = g(i \to j)A(i \to j), \tag{10.74}$$

where g is a selection probability while A is the probability for accepting a move. It is also called the acceptance ratio. The selection probability should be same for all possible spin orientations, namely

$$g(i \rightarrow j) = \frac{1}{N}. \tag{10.75}$$

With detailed balance this gives

$$\frac{g(j \rightarrow i)A(j \rightarrow i)}{g(i \rightarrow j)A(i \rightarrow j)} = \exp\left(-\beta(E_i - E_j)\right), \tag{10.76}$$

but since the selection ratio is the same for both transitions, we have

$$\frac{A(j \rightarrow i)}{A(i \rightarrow j)} = \exp\left(-\beta(E_i - E_j)\right) \tag{10.77}$$

In general, we are looking for those spin orientations which correspond to the average energy at equilibrium.

We are in this case interested in a new state E_j whose energy is lower than E_i, viz., $\Delta E = E_j - E_i \leq 0$. A simple test would then be to accept only those microstates which lower the energy. Suppose we have ten microstates with energy $E_0 \leq E_1 \leq E_2 \leq E_3 \leq \cdots \leq E_9$. Our desired energy is E_0. At a given temperature T we start our simulation by randomly choosing state E_9. Flipping spins we may then find a path from $E_9 \rightarrow E_8 \rightarrow E_7 \cdots \rightarrow E_1 \rightarrow E_0$. This would however lead to biased statistical averages since it would violate the ergodic hypothesis which 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 $E_9 \rightarrow E_7 \rightarrow E_8 \cdots \rightarrow E_9 \rightarrow E_5 \rightarrow E_0$ and so forth. Even though such a path could have a negligible probability it is still a possibility, and if we simulate long enough it should be included in our computation of an expectation value.

Thus, we require that our algorithm should satisfy the principle of detailed balance and be ergodic. One possible way is the Metropolis algorithm, which reads

$$A(j \rightarrow i) = \begin{cases} \exp\left(-\beta(E_i - E_j)\right) & E_i - E_j > 0 \\ 1 & else \end{cases} \tag{10.78}$$

This algorithm satisfies the condition for detailed balance and ergodicity. It is implemented as follows:

- Establish an initial energy E_b
- Do a random change of this initial state by e.g., flipping an individual spin. This new state has energy E_t. Compute then $\Delta E = E_t - E_b$
- If $\Delta E \leq 0$ accept the new configuration.

- If $\Delta E > 0$, compute $w = e^{-(\beta \Delta E)}$.
- Compare w with a random number r. If $r \leq w$ accept, else keep the old configuration.
- Compute the terms in the sums $\sum A_s P_s$.
- Repeat the above steps in order to have a large enough number of microstates
- For a given number of MC cycles, compute then expectation values.

The application of this algorithm will be discussed in detail in the next two chapters.

10.5 Physics project: simulation of the Boltzmann distribution

In this project the aim is to show that the Metropolis algorithm generates the Boltzmann distribution

$$P(\beta) = \frac{e^{-\beta E}}{Z}, \tag{10.79}$$

with $\beta = 1/kT$ being the inverse temperature, E is the energy of the system and Z is the partition function. The only functions you will need are those to generate random numbers.

We are going to study one single particle in equilibrium with its surroundings, the latter modeled via a large heat bath with temperature T.

The model used to describe this particle is that of an ideal gas in **one** dimension and with velocity $-v$ or v. We are interested in finding $P(v)dv$, which expresses the probability for finding the system with a given velocity $v \in [v, v + dv]$. The energy for this one-dimensional system is

$$E = \frac{1}{2}kT = \frac{1}{2}v^2, \tag{10.80}$$

with mass $m = 1$. In order to simulate the Boltzmann distribution, your program should contain the following ingredients:

- Reads in the temperature T, the number of Monte Carlo cycles, and the initial velocity. You should also read in the change in velocity δv used in every Monte Carlo step. Let the temperature have dimension energy.
- Thereafter you choose a maximum velocity given by e.g., $v_{max} \sim 10\sqrt{T}$. Then you construct a velocity interval defined by v_{max} and divided it in small intervals through v_{max}/N, with $N \sim 100 - 1000$. For each of these intervals your task is to find out how many times a given velocity during the Monte Carlo sampling appears in each specific interval.
- The number of times a given velocity appears in a specific interval is used to construct a histogram representing $P(v)dv$. To achieve this you should construct a vector $P[N]$ which contains the number of times a given velocity appears in the subinterval $v, v + dv$.

In order to find the number of velocities appearing in each interval we will employ the Metropolis algorithm. A pseudocode for this is

```
for( montecarlo_cycles=1; Max_cycles; montecarlo_cycles++) {
    ...
    // change speed as function of delta v
    v_change = (2*ran1(&idum) -1 )* delta_v;
    v_new = v_old+v_change;
    // energy change
    delta_E = 0.5*(v_new*v_new - v_old*v_old) ;
    ......
    // Metropolis algorithm begins here
      if ( ran1(&idum) <= exp(-beta*delta_E)  ) {
          accept_step = accept_step + 1 ;
          v_old = v_new ;
          .....
      }
    // thereafter we must fill in  P[N] as a function of
    // the new speed
      P[?] = ...

    // upgrade mean velocity , energy and variance
       ...
}
```

a) Make your own algorithm which sets up the histogram $P(v)dv$, find mean velocity, energy, energy variance and the number of accepted steps for a given temperature. Study the change of the number of accepted moves as a function of δv. Compare the final energy with the analytic result $E = kT/2$ for one dimension. Use $T = 4$ and set the intial velocity to zero, i.e., $v_0 = 0$. Try different values of δv. A possible start value is $\delta v = 4$. Check the final result for the energy as a function of the number of Monte Carlo cycles.

b) Make thereafter a plot of $ln(P(v))$ as function of E and see if you get a straight line. Comment the result.

Chapter 11

Monte Carlo methods in statistical physics

The aim of this chapter is to present examples from the physical sciences where Monte Carlo methods are widely applied. Here we focus on examples from statistical physics. and discuss one of the most studied systems, the Ising model for the interaction among classical spins. This model exhibits both first and second order phase transitions and is perhaps one of the most studied cases in statistical physics and discussions of simulations of phase transitions.

11.1 Phase transitions in magnetic systems

11.1.1 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

$$E = -J \sum_{<kl>}^{N} s_k s_l - B \sum_{k}^{N} s_k, \tag{11.1}$$

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 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, to 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 limit the attention to cases with $B = 0$ only.

In order to calculate expectation values such as the mean energy $\langle E \rangle$ or magnetization $\langle \mathcal{M} \rangle$ in statistical physics at a given temperature, we need a probability distribution

$$P_i(\beta) = \frac{e^{-\beta E_i}}{Z} \tag{11.2}$$

with $\beta = 1/kT$ being the inverse temperature, k the Boltzmann constant, E_i is the energy of a state i while Z is the partition function for the canonical ensemble defined as

$$Z = \sum_{i=1}^{M} e^{-\beta E_i}, \tag{11.3}$$

where the sum extends over all states M. P_i expresses the probability of finding the system in a given configuration i.

The energy for a specific configuration i is given by

$$E_i = -J \sum_{<kl>}^{N} s_k s_l. \tag{11.4}$$

To better understand what is meant with a configuration, consider first the case of the one-dimensional Ising model with $B = 0$. 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, it means that in total we have $2^2 = 4$ possible arrangements of the two spins. These four possibilities are

$$1 = \uparrow\uparrow \qquad 2 = \uparrow\downarrow \qquad 3 = \downarrow\uparrow \qquad 4 = \downarrow\downarrow$$

What is the energy of each of these configurations?

For small systems, the way we treat the ends matters. Two cases are often used

1. In the first case we employ what is called free ends. For the one-dimensional case, the energy is then written as a sum over a single index

$$E_i = -J \sum_{j=1}^{N-1} s_j s_{j+1}, \tag{11.5}$$

If we label the first spin as s_1 and the second as s_2 we obtain the following expression for the energy

$$E = -J s_1 s_2. \tag{11.6}$$

The calculation of the energy for the one-dimensional lattice with free ends for one specific spin-configuration can easily be implemented in the following lines

```
for ( j=1; j < N; j++) {
    energy += spin[j]*spin[j+1];
}
```

where the vector $spin[]$ contains the spin value $s_k = \pm 1$. For the specific state E_1, we have chosen all spins up. The energy of this configuration becomes then

$$E_1 = E_{\uparrow\uparrow} = -J.$$

The other configurations give

$$E_2 = E_{\uparrow\downarrow} = +J,$$

$$E_3 = E_{\downarrow\uparrow} = +J,$$

and

$$E_4 = E_{\downarrow\downarrow} = -J.$$

2. We can also choose so-called periodic boundary conditions. This means that if $i = N$, we set the spin number to $i = 1$. In this case the energy for the one-dimensional lattice reads

$$E_i = -J \sum_{j=1}^{N} s_j s_{j+1}, \tag{11.7}$$

and we obtain the following expression for the two-spin case

$$E = -J(s_1 s_2 + s_2 s_1). \tag{11.8}$$

In this case the energy for E_1 is different, we obtain namely

$$E_1 = E_{\uparrow\uparrow} = -2J.$$

The other cases do also differ and we have

$$E_2 = E_{\uparrow\downarrow} = +2J,$$

$$E_3 = E_{\downarrow\uparrow} = +2J,$$

and

$$E_4 = E_{\downarrow\downarrow} = -2J.$$

If we choose to use periodic boundary conditions we can code the above expression as

```
jm=N;
for ( j=1; j <=N ; j++) {
     energy += spin[j]*spin[jm];
     jm = j ;
}
```

Table 11.1: 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 11.2: Degeneracy, energy and magnetization for the one-dimensional Ising model with $N = 2$ spins with free ends (FE) and periodic boundary conditions (PBC).

Number spins up	Degeneracy	Energy (FE)	Energy (PBC)	Magnetization
2	1	$-J$	$-2J$	2
1	2	J	$2J$	0
0	1	$-J$	$-2J$	-2

The magnetization is however the same, defined as

$$\mathcal{M}_i = \sum_{j=1}^{N} s_j, \tag{11.9}$$

where we sum over all spins for a given configuration i.

Table 11.1 lists the energy and magnetization for both free ends and periodic boundary conditions.

We can reorganize Table 11.1 according to the number of spins pointing up, as shown in Table 11.2. It is worth noting that for small dimensions of the lattice, the energy differs depending on whether we use periodic boundary conditions or fri ends. This means also that the partition functions will be different, as discussed below. In the thermodynamic limit however, $N \to \infty$, the final results do not depend on the kind of boundary conditions we choose.

For a one-dimensional lattice with periodic boundary conditions, each spin sees two neighbors. For a two-dimensional lattice each spin sees four neighboring spins. How many neighbors does a spin see in three dimensions?

In a similar way, we could enumerate the number of states for a two-dimensional system consisting of two spins, i.e., a 2×2 Ising model on a square lattice with *periodic boundary conditions*. In this case we have a total of $2^4 = 16$ states. Some examples of configurations with their respective energies are listed here

$$E = -8J \qquad \begin{matrix}\uparrow & \uparrow \\ \uparrow & \uparrow\end{matrix} \qquad E = 0 \qquad \begin{matrix}\uparrow & \uparrow \\ \uparrow & \downarrow\end{matrix} \qquad E = 0 \qquad \begin{matrix}\downarrow & \downarrow \\ \uparrow & \downarrow\end{matrix} \qquad E = -8J \qquad \begin{matrix}\downarrow & \downarrow \\ \downarrow & \downarrow\end{matrix}$$

In the Table 11.3 we group these configurations according to their total energy and magnetization.

Table 11.3: 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

Exercise 11.1
Convince yourself that the values listed in Table 11.3 are correct.

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 the probability distribution P_i as

$$\langle E \rangle = \sum_{i=1}^{M} E_i P_i(\beta) = \frac{1}{Z} \sum_{i=1}^{M} E_i e^{-\beta E_i}, \tag{11.10}$$

with a corresponding variance defined as

$$\sigma_E^2 = \langle E^2 \rangle - \langle E \rangle^2 = \frac{1}{Z} \sum_{i=1}^{M} E_i^2 e^{-\beta E_i} - \left(\frac{1}{Z} \sum_{i=1}^{M} E_i e^{-\beta E_i} \right)^2 . \tag{11.11}$$

If we divide the latter quantity with kT^2 we obtain the specific heat at constant volume

$$C_V = \frac{1}{kT^2} \left(\langle E^2 \rangle - \langle E \rangle^2 \right) . \tag{11.12}$$

Using the same prescription, we can also evaluate the mean magnetization through

$$\langle \mathcal{M} \rangle = \sum_{i}^{M} \mathcal{M}_i P_i(\beta) = \frac{1}{Z} \sum_{i}^{M} \mathcal{M}_i e^{-\beta E_i}, \tag{11.13}$$

and the corresponding variance

$$\sigma_{\mathcal{M}}^2 = \langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2 = \frac{1}{Z} \sum_{i=1}^{M} \mathcal{M}_i^2 e^{-\beta E_i} - \left(\frac{1}{Z} \sum_{i=1}^{M} \mathcal{M}_i e^{-\beta E_i} \right)^2 . \tag{11.14}$$

This quantity defines also the susceptibility χ

$$\chi = \frac{1}{kT}\left(\langle \mathcal{M}^2\rangle - \langle \mathcal{M}\rangle^2\right). \tag{11.15}$$

It is also possible to show that the partition function for the one-dimensional case for N spins with free ends is

$$Z_N = \left[2cosh(\beta J)\right]^{N-1}. \tag{11.16}$$

If we use periodic boundary conditions it becomes

$$Z_N = 2^N\left(\left[cosh(\beta J)\right]^N + \left[sinh(\beta J)\right]^N\right). \tag{11.17}$$

In the limit $N \rightarrow \infty$ the two results agree.

We can then calculate the mean energy with fri ends from the above formula or using

$$\langle E\rangle = -\frac{\partial lnZ}{\partial \beta} = -(N-1)Jtanh(\beta J). \tag{11.18}$$

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

$$Z_2 = \sum_{i=1}^{2} e^{-\beta E_i} = 2e^{-\beta J} + 2e^{\beta J} = 2cosh(\beta J) \tag{11.19}$$

If we take the limit $T \rightarrow 0$ ($\beta \rightarrow \infty$) and set $N = 2$, we obtain

$$\lim_{\beta\rightarrow\infty}\langle E\rangle = -\frac{e^{J\beta} - e^{-J\beta}}{e^{J\beta} + e^{-J\beta}} = -J, \tag{11.20}$$

which is the energy where all spins point in the same direction. At low T, the system tends towards a state with the highest possible degree of order.

The specific heat in one-dimension with free ends is

$$C_V = \frac{1}{kT^2}\frac{\partial^2}{\partial\beta^2} lnZ_N = (N-1)k\left(\frac{\beta J}{cosh(\beta J)}\right)^2. \tag{11.21}$$

Exercise 11.2
Calculate the exact partition function for a system of three spins with the one-dimensional Ising model using both free ends and periodic boundary conditions.

For our two-dimensional 2×2 lattice we obtain the following partition function

$$Z = 2e^{-8J\beta} + 2e^{8J\beta} + 12, \tag{11.22}$$

and mean energy

$$\langle E\rangle = -\frac{1}{Z}\left(16e^{8J\beta} - 16e^{-8J\beta}\right). \tag{11.23}$$

The analytical expression for the Ising model in two dimensions was obtained in 1944 by the Norwegian chemist Lars Onsager (Nobel prize in chemistry). The exact partition function for N spins is given by

$$Z_N = \left[2cosh(\beta J)e^I\right]^N, \tag{11.24}$$

with

$$I = \frac{1}{2\pi}\int_0^\pi d\phi ln\left[\frac{1}{2}\left(1+(1-\kappa^2 sin^2\phi)^{1/2}\right)\right], \tag{11.25}$$

and

$$\kappa = 2sinh(2\beta J)/cosh^2(2\beta J). \tag{11.26}$$

Exercise 11.3
Calculate the heat capacity and the mean magnetization per spin for the 2×2 ising model.

11.1.2 The Metropolis algorithm

The algorithm of choice for solving the Ising model is the approach proposed by Metropolis *et al.* in 1953. As discussed in chapter **??**, new configurations are generated from a previous one using a transition probability which depends on the energy difference between the initial and final states.

In our case we have as the Monte Carlo sampling function the probability for finding the system in a state s given by

$$P_s = \frac{e^{-(\beta E_s)}}{Z},$$

with energy E_s, $\beta = 1/kT$ and Z is a normalization constant which defines the partition function in the canonical ensemble. As discussed above

$$Z(\beta) = \sum_s e^{-(\beta E_s)}$$

is difficult to compute since we need all states. In a calculation of the Ising model in two dimensions, the number of configurations is given by 2^N with $N = L \times L$ the number of spins for a lattice of length L. Fortunately, the Metropolis algorithm considers only ratios between probabilities and we do not need to compute the partition function at all. The algorithm goes as follows

1. Establish an initial state with energy E_b by positioning yourself at a random position in the lattice

2. Change the initial configuration by flipping e.g., one spin only. Compute the energy of this trial state E_t.

3. Calculate $\Delta E = E_t - E_b$. The number of values ΔE is limited to five for the Ising model in two dimensions, see the discussion below.

4. If $\Delta E \leq 0$ we accept the new configuration, meaning that the energy is lowered and we are hopefully moving towards the energy minimum at a given temperature. Go to step 7.

5. If $\Delta E > 0$, calculate $w = e^{-(\beta \Delta E)}$.

6. Compare w with a random number r. If

$$r \leq w,$$

then accept the new configuration, else we keep the old configuration.

7. The next step is to update various expectations values.

8. The steps (2)-(7) are then repeated in order to obtain a sufficently good representation of states.

9. Each time you sweep through the lattice, i.e., when you have summed over all spins, constitutes what is called a Monte Carlo cyclus. You could think of one such cyclus 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 implementation of this algorithm is given in the next section. 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.

11.2 Program example

We list here an example of a C/C++-program which computes various thermodynamical properties of the Ising model in two dimensions You should especially pay attention to the function Metropolis which implements the algorithm described in the previous subsection and the function $DeltaE$ which calculates the energy difference between the previous state and the trial state by flipping one spin.

The program is set up with dynamic memory allocation for the matrix which contains the spin values at a position (x, y). One could alternatively have used a fixed size for the matrices to be used. But then one would need to recompile the program when larger systems are considered.

11.2.1 Program for the two-dimensional Ising Model

programs/chap11/program1.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 *);
```

```
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] = 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, 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;
}
```

```
// read in input data
void read_input(int& n_spins, int& mcs, double& initial_temp,
                double& final_temp, double& temp_step)
{
  cout << "Number of Monte Carlo trials =";
  cin >> mcs;
  cout << "Lattice size or number of spins (x and y equal) =";
  cin >> n_spins;
  cout << "Initial temperature with dimension energy=";
  cin >> initial_temp;
  cout << "Final temperature with dimension energy=";
  cin >> final_temp;
  cout << "Temperature step with dimension energy=";
  cin >> temp_step;
} // end of function read_input
```

```
// 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++){
      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
```

```
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++){
      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]);
      if ( ran1(&idum) <= w[deltaE+8] ) {
        spin_matrix[iy][ix] *= -1;  // flip one spin and accept new
           spin config
        M += (double) 2*spin_matrix[iy][ix];
        E += (double) deltaE;
      }
    }
  }
} // end of Metropolis sampling over spins
```

```
void output(int n_spins, int mcs, double temp, 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 - Mabsaverage*Mabsaverage)/n_spins/
```

```
    n_spins;
  ofile << setiosflags(ios::showpoint | ios::uppercase);
  ofile << setw(15) << setprecision(8) << temp;
  ofile << setw(15) << setprecision(8) << Eaverage/n_spins/n_spins;
  ofile << setw(15) << setprecision(8) << Evariance/temp/temp;
  ofile << setw(15) << setprecision(8) << Maverage/n_spins/n_spins;
  ofile << setw(15) << setprecision(8) << Mvariance/temp;
  ofile << setw(15) << setprecision(8) << Mabsaverage/n_spins/n_spins
    << endl;
} // end output function
```

11.3 Selected results for the Ising model

11.3.1 Phase transitions

The Ising model in two dimensions and with $B = 0$ undergoes a phase transition of second order. What it actually means is that below a given critical temperature T_C, the Ising model exhibits a spontaneous magnetization with $\langle \mathcal{M} \rangle \neq 0$. Above T_C the average magnetization is zero. The one-dimensional Ising model does not predict any spontaneous magnetization at any finite temperature. The physical reason for this can be understood from the following simple consideration. Assume that the ground state for an N-spin system in one dimension is characterized by the following configuration

$$\begin{array}{cccccccccc} \uparrow & \uparrow & \uparrow & \dots & \uparrow & \uparrow & \uparrow & \dots & \uparrow & \uparrow \\ 1 & 2 & 3 & \dots & i-1 & i & i+1 & \dots & N-1 & N \end{array}$$

which has a total energy $-NJ$ and magnetization N. If we flip half of the spins we arrive at a configuration

$$\begin{array}{cccccccccc} \uparrow & \downarrow & \uparrow & \dots & \uparrow & \downarrow & \uparrow & \dots & \uparrow & \downarrow \\ 1 & 2 & 3 & \dots & i-1 & i & i+1 & \dots & N-1 & N \end{array}$$

with energy $(-N+4)J$ and net magnetization zero. This state is an example of a disordered state. The change in energy is however too small to stabilize the disordered state. In two dimensions however the excitation energy to a disordered state is much higher, and this difference can be sufficient to stabilize the system. In fact, the Ising model exhibits a phase transition to a disordered phase both in two and three dimensions.

For the two-dimensional case, we move from a phase with finite magnetization $\langle \mathcal{M} \rangle \neq 0$ to a paramagnetic phase with $\langle \mathcal{M} \rangle = 0$ at a critical temperature T_C. At the critical temperature, quantities like the heat capacity C_V and the susceptibility χ diverge in the thermodynamic limit, i.e., with an infinitely large lattice. This means that the variance in energy and magnetization 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. Since our lattices will always be of a finite dimensions, the calculated C_V or χ will not exhibit a diverging behavior.

We will however notice a broad maximum in e.g., C_V near T_C. This maximum, as discussed below, becomes sharper and sharper as L is increased.

Near T_C we can characterize the behavior of many physical quantities by a power law behavior. As an example, the mean magnetization is given by

$$\langle \mathcal{M}(T) \rangle \sim (T - T_C)^{\beta}, \tag{11.27}$$

where β is a so-called critical exponent. A similar relation applies to the heat capacity

$$C_V(T) \sim |T_C - T|^{-\gamma}, \tag{11.28}$$

and the susceptibility

$$\chi(T) \sim |T_C - T|^{-\alpha}. \tag{11.29}$$

Another important quantity is the correlation length, which is expected to be of the order of the lattice spacing for $T >> T_C$. Because the spins become more and more correlated as T approaches T_C, the correlation length increases as we get closer to the critical temperature. The divergent behavior of ξ near T_C is

$$\xi(T) \sim |T_C - T|^{-\nu}. \tag{11.30}$$

A second-order phase transition is characterized by a correlation length which spans the whole system. Since we are always limited to a finite lattice, ξ will be proportional with the size of the lattice.

Through finite size scaling relations it is possible to relate the behavior at finite lattices with the results for an infinitely large lattice. The critical temperature scales then as

$$T_C(L) - T_C(L = \infty) \sim aL^{-1/\nu}, \tag{11.31}$$

with a a constant and ν is defined in Eq. (11.30). The correlation length is given by

$$\xi(T) \sim L \sim |T_C - T|^{-\nu}. \tag{11.32}$$

and if we set $T = T_C$ one obtains

$$\langle \mathcal{M}(T) \rangle \sim (T - T_C)^{\beta} \rightarrow L^{-\beta/\nu}, \tag{11.33}$$

$$C_V(T) \sim |T_C - T|^{-\gamma} \rightarrow L^{\alpha/\nu}, \tag{11.34}$$

and

$$\chi(T) \sim |T_C - T|^{-\alpha} \rightarrow L^{\gamma/\nu}. \tag{11.35}$$

11.3.2 Heat capacity and susceptibility as functions of number of spins

in preparation

11.3.3 Thermalization

in preparation

11.4 Other spin models

11.4.1 Potts model

11.4.2 XY-model

11.5 Physics project: simulation of the Ising model

In this project we will use the Metropolis algorithm to generate states according to the Boltzmann distribution. Each new configuration is given by the change of only one spin at the time, that is $s_k \rightarrow -s_k$. Use periodic boundary conditions and set the magnetic field $B = 0$.

a) Write a program which simulates the one-dimensional Ising model. Choose $J > 0$, the number of spins $N = 20$, temperature $T = 3$ and the number of Monte Carlo samples $mcs = 100$. Let the initial configuration consist of all spins pointing up, i.e., $s_k = 1$. Compute the mean energy and magnetization for each cycle and find the number of cycles needed where the fluctuation of these variables is negligible. What kind of criterium would you use in order to determine when the fluctuations are negligible?

   Change thereafter the initial condition by letting the spins take random values, either -1 or 1. Compute again the mean energy and magnetization for each cycle and find the number of cycles needed where the fluctuation of these variables is negligible.

   Explain your results.

b) Let $mcs \geq 1000$ and compute $\langle E \rangle$, $\langle E^2 \rangle$ and C_V as functions of T for $0.1 \leq T \leq 5$. Plot the results and compare with the exact ones for periodic boundary conditions.

c) Using the Metropolis sampling method you should now find the number of accepted configurations as function of the total number of Monte Carlo samplings. How does the number of accepted configurations behave as function of temperature T? Explain the results.

d) Compute thereafter the probability $P(E)$ for a system with $N = 50$ at $T = 1$. Choose $mcs \geq 1000$ and plot $P(E)$ as function of E. Count the number of times a specific energy appears and build thereafter up a histogram. What does the histogram mean?

Chapter 12

Quantum Monte Carlo methods

12.1 Introduction

The aim of this chapter is to present examples of applications of Monte Carlo methods in studies of quantum mechanical systems. We study systems such as the harmonic oscillator, the hydrogen atom, the hydrogen molecule, the helium atom and the nucleus ^{4}He.

The first section deals with variational methods, or what is commonly denoted as variational Monte Carlo (VMC). The required Monte Carlo techniques for VMC are conceptually simple, but the practical application may turn out to be rather tedious and complex, relying on a good starting point for the variational wave functions. These wave functions should include as much as possible of the inherent physics to the problem, since they form the starting point for a variational calculation of the expectation value of the hamiltonian H. Given a hamiltonian H and a trial wave function Ψ_T, the variational principle states that the expectation value of $\langle H \rangle$, defined through

$$\langle H \rangle = \frac{\int d\mathbf{R}\Psi_T^*(\mathbf{R})H(\mathbf{R})\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi_T^*(\mathbf{R})\Psi_T(\mathbf{R})}, \tag{12.1}$$

is an upper bound to the ground state energy E_0 of the hamiltonian H, that is

$$E_0 \leq \langle H \rangle. \tag{12.2}$$

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, viz.,

$$\Psi_T(\mathbf{R}) = \sum_i a_i \Psi_i(\mathbf{R}), \tag{12.3}$$

and assuming the set of eigenfunctions to be normalized, insertion of the latter equation in Eq. (12.1) results in

$$\langle H \rangle = \frac{\sum_{mn} a_m^* a_n \int d\mathbf{R}\Psi_m^*(\mathbf{R})H(\mathbf{R})\Psi_n(\mathbf{R})}{\sum_{mn} a_m^* a_n \int d\mathbf{R}\Psi_m^*(\mathbf{R})\Psi_n(\mathbf{R})} = \frac{\sum_{mn} a_m^* a_n \int d\mathbf{R}\Psi_m^*(\mathbf{R})E_n(\mathbf{R})\Psi_n(\mathbf{R})}{\sum_n a_n^2}, \tag{12.4}$$

which can be rewritten as

$$\frac{\sum_n a_n^2 E_n}{\sum_n a_n^2} \geq E_0 . \tag{12.5}$$

In general, the integrals involved in the calculation of various expectation values are multi-dimensional ones. Traditional integration methods such as the Gauss-Legendre will not be adequate for say the computation of the energy of a many-body system. The fact that we need to sample over a multi-dimensional density and that the probability density is to be normalized by the division of the norm of the wave function, suggests that e.g., the Metropolis algorithm may be appropriate.

We could briefly summarize the above variational procedure in the following three steps.

1. Construct first a trial wave function $\psi_T^{\alpha}(\mathbf{R})$, for say a many-body system consisting of N particles located at positions $\mathbf{R} = (\mathbf{R_1}, \ldots, \mathbf{R_N})$. The trial wave function depends on α variational parameters $\alpha = (\alpha_\mathbf{1}, \ldots, \alpha_\mathbf{N})$.

2. Then we evaluate the expectation value of the hamiltonian H

$$\langle H \rangle = \frac{\int d\mathbf{R} \Psi_{T_\alpha}^*(\mathbf{R}) H(\mathbf{R}) \Psi_{T_\alpha}(\mathbf{R})}{\int d\mathbf{R} \Psi_{T_\alpha}^*(\mathbf{R}) \Psi_{T_\alpha}(\mathbf{R})}.$$

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 VMC calculation is the search for the variational minimum. A good knowledge of the system is required in order to carry out reasonable VMC calculations. This is not always the case, and often VMC calculations serve rather as the starting point for so-called diffusion Monte Carlo calculations (DMC). DMC is a way of solving exactly the many-body Schrödinger equation by means of a stochastic procedure. A good guess on the binding energy and its wave function is however necessary. A carefully performed VMC calculation can aid in this context.

12.2 Variational Monte Carlo for quantum mechanical systems

The variational quantum Monte Carlo (VMC) 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 4 as well, consists in choosing a trial wave function $\psi_T(\mathbf{R})$ which we assume to be as realistic as possible. The variable $\mathbf{R}$ stands for the spatial coordinates, in total $3N$ if we have N particles present. The trial wave function serves then, following closely the discussion on importance sampling in section 9.5, as a mean to define the quantal probability distribution

$$P(\mathbf{R}) = \frac{|\psi_T(\mathbf{R})|^2}{\int |\psi_T(\mathbf{R})|^2 d\mathbf{R}}. \tag{12.6}$$

This is our new probability distribution function (PDF).

The expectation value of the energy E is given by

$$\langle E \rangle = \frac{\int d\mathbf{R}\Psi^*(\mathbf{R})H(\mathbf{R})\Psi(\mathbf{R})}{\int d\mathbf{R}\Psi^*(\mathbf{R})\Psi(\mathbf{R})}, \tag{12.7}$$

where Ψ is the exact eigenfunction. Using our trial wave function we define a new operator, the so-called local energy,

$$E_L(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})} H\psi_T(\mathbf{R}), \tag{12.8}$$

which, together with our trial PDF allows us to rewrite the expression for the energy as

$$\langle H \rangle = \int P(\mathbf{R})E_L(\mathbf{R})d\mathbf{R}. \tag{12.9}$$

This equation expresses the variational Monte Carlo approach. For most hamiltonians, H is a sum of kinetic energy, involving a second derivative, and a momentum independent potential. The contribution from the potential term is hence just the numerical value of the potential.

At this stage, we should note the similarity between Eq. (12.9) and the concept of importance sampling introduced in chapter 4, in connection with numerical integration and Monte Carlo methods.

In our discussion below, we base our numerical Monte Carlo solution on the Metropolis algorithm. The implementation is rather similar to the one discussed in connection with the Ising model, the main difference residing in the form of the PDF. The main test to be performed is a ratio of probabilities. Suppose we are attempting to move from position $\mathbf{R}$ to $\mathbf{R}'$. Then we perform the following two tests.

1. If

$$\frac{P(\mathbf{R}')}{P(\mathbf{R})} > 1,$$

where $\mathbf{R}'$ is the new position, the new step is accepted, or

2.

$$r \leq \frac{P(\mathbf{R}')}{P(\mathbf{R})},$$

where r is random number generated with uniform PDF such that $r \in [0,1]$, the step is also accepted.

In the Ising model we were flipping one spin at the time. Here we change the position of say a given particle to a trial position $\mathbf{R}'$, and then evaluate the ratio between two probabilities. We note again that we do not need to evaluate the norm[1] $\int |\psi_T(\mathbf{R})|^2 \, d\mathbf{R}$ (an in general impossible task), since we are only computing ratios.

When writing a VMC program, one should always prepare in advance the required formulae for the local energy E_L in Eq. (12.9) 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. (12.9), 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 $|\psi_T^{\alpha}(\mathbf{R})|^2$. Define also the value of the stepsize to be used when moving from one value of $\mathbf{R}$ to a new one.
- Initialise the energy and the variance.
- Start the Monte Carlo calculation
  1. Thermalise first.
  2. Thereafter start your Monte carlo sampling.
  3. Calculate a trial position $\mathbf{R}_p = \mathbf{R} + r * step$ where r is a random variable $r \in [0,1]$.
  4. 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.
  5. If the step is accepted, then we set $\mathbf{R} = \mathbf{R}_p$.
  6. 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.

The best way however to understand a specific method is however to study selected examples.

12.2.1 First illustration of VMC methods, the one-dimensional harmonic oscillator

The harmonic oscillator in one dimension lends itself nicely for illustrative purposes. The hamiltonian is

$$H = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + \frac{1}{2}kx^2, \tag{12.10}$$

[1]This corresponds to the partition function Z in statistical physics.

where m is the mass of the particle and k is the force constant, e.g., the spring tension for a classical oscillator. In this example we will make life simple and choose $m = \hbar = k = 1$. We can rewrite the above equation as

$$H = -\frac{d^2}{dx^2} + x^2, \tag{12.11}$$

The energy of the ground state is then $E_0 = 1$. The exact wave function for the ground state is

$$\Psi_0(x) = \frac{1}{\pi^{1/4}} e^{-x^2/2}, \tag{12.12}$$

but since we wish to illustrate the use of Monte Carlo methods, we choose the trial function

$$\Psi_T(x) = \frac{\sqrt{\alpha}}{\pi^{1/4}} e^{-x^2\alpha^2/2}. \tag{12.13}$$

Inserting this function in the expression for the local energy in Eq. (12.8), we obtain the following expression for the local energy

$$E_L(x) = \alpha^2 + x^2(1 - \alpha^4), \tag{12.14}$$

with the expectation value for the hamiltonian of Eq. (12.9) given by

$$\langle H \rangle = \int_{-\infty}^{\infty} |\psi_T(x)|^2 E_L(x) dx, \tag{12.15}$$

which reads with the above trial wave function

$$\langle H \rangle = \frac{\int_{-\infty}^{\infty} dx e^{-x^2\alpha^2} \alpha^2 + x^2(1 - \alpha^4)}{\int_{-\infty}^{\infty} dx e^{-x^2\alpha^2}}. \tag{12.16}$$

Using the fact that

$$\int_{-\infty}^{\infty} dx e^{-x^2\alpha^2} = \sqrt{\frac{\pi}{\alpha^2}},$$

we obtain

$$\langle H \rangle = \frac{\alpha^2}{2} + \frac{1}{2\alpha^2}. \tag{12.17}$$

and the variance

$$\sigma^2 = \frac{(\alpha^4 - 1)^2}{2\alpha^4}. \tag{12.18}$$

In solving this problem we can choose whether we wish to use the Metropolis algorithm and sample over relevant configurations, or just use random numbers generated from a normal distribution, since the harmonic oscillator wave functions follow closely such a distribution. The latter approach is easily implemented in few lines, namely

```
...   initialisations , declarations of variables
...   mcs = number of  Monte Carlo samplings
//    loop over Monte Carlo samples
      for ( i=0; i < mcs; i++) {
//    generate random variables from gaussian distribution
          x = normal_random(&idum)/sqrt2/alpha;
          local_energy = alpha*alpha + x*x*(1-pow(alpha,4));
          energy += local_energy;
          energy2 += local_energy*local_energy;
//    end of sampling
      }
//    write out the mean energy and the standard deviation
      cout << energy/mcs <<  sqrt((energy2/mcs-(energy/mcs)**2)/mcs));
```

This VMC calculation is rather simple, we just generate a large number N of random numbers corresponding to the gaussian PDF $\sim |\Psi_T|^2$ and for each random number we compute the local energy according to the approximation

$$\langle H \rangle = \int P(\mathbf{R}) E_L(\mathbf{R}) d\mathbf{R} \approx \frac{1}{N} \sum_{i=1}^{N} E_L(x_i), \tag{12.19}$$

and the energy squared through

$$\langle H^2 \rangle = \int P(\mathbf{R}) E_L^2(\mathbf{R}) d\mathbf{R} \approx \frac{1}{N} \sum_{i=1}^{N} E_L^2(x_i). \tag{12.20}$$

In a certain sense, this is nothing but the importance Monte Carlo sampling discussed in chapter 4. Before we proceed however, there is an important aside which is worth keeping in mind when computing the local energy. We could think of splitting the computation of the expectation value of the local energy into a kinetic energy part and a potential energy part. If we are dealing with a three-dimensional system, the expectation value of the kinetic energy is

$$-\frac{\int d\mathbf{R} \Psi_T^*(\mathbf{R}) \nabla^2 \Psi_T(\mathbf{R})}{\int d\mathbf{R} \Psi_T^*(\mathbf{R}) \Psi_T(\mathbf{R})}, \tag{12.21}$$

and we could be tempted to compute, if the wave function obeys spherical symmetry, just the second derivative with respect to one coordinate axis and then multiply by three. This will most likely increase the variance, and should be avoided, even if the final expectation values are similar. Recall that one of the subgoals of a Monte Carlo computation is to decrease the variance.

Another shortcut we could think of is to transform the numerator in the latter equation to

$$\int d\mathbf{R} \Psi_T^*(\mathbf{R}) \nabla^2 \Psi_T(\mathbf{R}) = -\int d\mathbf{R} (\nabla \Psi_T^*(\mathbf{R}))(\nabla \Psi_T(\mathbf{R})), \tag{12.22}$$

using integration by parts and the relation

$$\int d\mathbf{R}\nabla(\Psi_T^*(\mathbf{R})\nabla\Psi_T(\mathbf{R})) = 0, \tag{12.23}$$

where we have used the fact that the wave function is zero at $\mathbf{R} = \pm\infty$. This relation can in turn be rewritten through integration by parts to

$$\int d\mathbf{R}(\nabla\Psi_T^*(\mathbf{R}))(\nabla\Psi_T(\mathbf{R})) + \int d\mathbf{R}\Psi_T^*(\mathbf{R})\nabla^2\Psi_T(\mathbf{R})) = 0. \tag{12.24}$$

The rhs of Eq. (12.22) is easier and quicker to compute. However, in case the wave function is the exact one, or rather close to the exact one, the lhs yields just a constant times the wave function squared, implying zero variance. The rhs does not and may therefore increase the variance.

If we use integration by part for the harmonic oscillator case, the new local energy is

$$E_L(x) = x^2(1+\alpha^4), \tag{12.25}$$

and the variance

$$\sigma^2 = \frac{(\alpha^4+1)^2}{2\alpha^4}, \tag{12.26}$$

which is larger than the variance of Eq. (12.18).

We defer the study of the harmonic oscillator using the Metropolis algorithm till the after the discussion of the hydrogen atom.

12.2.2 The hydrogen atom

The radial Schrödinger equation for the hydrogen atom can be written as

$$-\frac{\hbar^2}{2m}\frac{\partial^2 u(r)}{\partial r^2} - \left(\frac{ke^2}{r} - \frac{\hbar^2 l(l+1)}{2mr^2}\right)u(r) = Eu(r), \tag{12.27}$$

where m is the mass of the electron, l its orbital momentum taking values $l = 0, 1, 2, \ldots$, and the term ke^2/r is the Coulomb potential. The first terms is the kinetic energy. The full wave function will also depend on the other variables θ and ϕ as well. The energy, with no external magnetic field is however determined by the above equation . We can then think of the radial Schrödinger equation to be equivalent to a one-dimensional movement conditioned by an effective potential

$$V_{\text{eff}}(r) = -\frac{ke^2}{r} + \frac{\hbar^2 l(l+1)}{2mr^2}. \tag{12.28}$$

When solving equations numerically, it is often convenient to rewrite the equation in terms of dimensionless variables. One reason is the fact that several of the constants may be differ largely in value, and hence result in potential losses of numerical precision. The other main reason for doing this is that the equation in dimensionless form is easier to code, sparing one for eventual

typographic errors. In order to do so, we introduce first the dimensionless variable $\rho = r/\beta$, where β is a constant we can choose. Schrödinger's equation is then rewritten as

$$-\frac{1}{2}\frac{\partial^2 u(\rho)}{\partial \rho^2} - \frac{mke^2\beta}{\hbar^2\rho}u(\rho) + \frac{l(l+1)}{2\rho^2}u(\rho) = \frac{m\beta^2}{\hbar^2}Eu(\rho). \tag{12.29}$$

We can determine β by simply requiring[2]

$$\frac{mke^2\beta}{\hbar^2} = 1 \tag{12.30}$$

With this choice, the constant β becomes the famous Bohr radius $a_0 = 0.05$ nm

$$a_0 = \beta = \frac{\hbar^2}{mke^2}.$$

We introduce thereafter the variable λ

$$\lambda = \frac{m\beta^2}{\hbar^2}E, \tag{12.31}$$

and inserting β and the exact energy $E = E_0/n^2$, with $E_0 = 13.6$ eV, we have that

$$\lambda = -\frac{1}{2n^2}, \tag{12.32}$$

n being the principal quantum number. The equation we are then going to solve numerically is now

$$-\frac{1}{2}\frac{\partial^2 u(\rho)}{\partial \rho^2} - \frac{u(\rho)}{\rho} + \frac{l(l+1)}{2\rho^2}u(\rho) - \lambda u(\rho) = 0, \tag{12.33}$$

with the hamiltonian

$$H = -\frac{1}{2}\frac{\partial^2}{\partial \rho^2} - \frac{1}{\rho} + \frac{l(l+1)}{2\rho^2}. \tag{12.34}$$

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. (12.33) is

$$u(\rho) = \rho e^{-\rho}, \tag{12.35}$$

which yields the energy $\lambda = -1/2$. Sticking to our variational philosophy, we could now introduce a variational parameter α resulting in a trial wave function

$$u_T^{\alpha}(\rho) = \alpha\rho e^{-\alpha\rho}. \tag{12.36}$$

Inserting this wave function into the expression for the local energy E_L of Eq. (12.8) yields (check it!)

$$E_L(\rho) = -\frac{1}{\rho} - \frac{\alpha}{2}\left(\alpha - \frac{2}{\rho}\right). \tag{12.37}$$

[2]Remember that we are free to choose β.

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. (12.9) now reads

$$\langle H \rangle = \int P(\mathbf{R}) E_L(\mathbf{R}) d\mathbf{R} = \int_0^\infty \alpha^2 \rho^2 e^{-2\alpha\rho} E_L(\rho) \rho^2 d\rho, \tag{12.38}$$

since $\rho \in [0, \infty]$. In this case we would use the exponential distribution instead of the normal distrubution, and our code would contain the following elements

```
...   initialisations , declarations of variables
...   mcs = number of  Monte Carlo samplings

//    loop over Monte Carlo samples
      for ( i=0; i < mcs; i++) {

//    generate random variables from the exponential
//    distribution using ran1 and transforming to
//    to an exponential mapping y = -ln(1-x)
          x=ran1(&idum);
          y=-log(1.-x);
//    in our case y = rho*alpha*2
          rho = y/alpha/2;
          local_energy = -1/rho -0.5*alpha*(alpha-2/rho);
          energy += (local_energy);
          energy2 += local_energy*local_energy;
//    end of sampling
      }
//    write out the mean energy and the standard deviation
      cout << energy/mcs << sqrt((energy2/mcs-(energy/mcs)**2)/mcs));
```

As for the harmonic oscillator case we just need to generate a large number N of random numbers corresponding to the exponential PDF $\alpha^2\rho^2 e^{-2\alpha\rho}$ and for each random number we compute the local energy and variance.

12.2.3 Metropolis sampling for the hydrogen atom and the harmonic oscillator

We present in this subsection results for the ground states of the hydrogen atom and harmonic oscillator using a variational Monte Carlo procedure. For the hydrogen atom, the trial wave function

$$u_T^\alpha(\rho) = \alpha\rho e^{-\alpha\rho},$$

depends only on the dimensionless radius ρ. It is the solution of a one-dimensional differential equation, as is the case for the harmonic oscillator as well. The latter has the trial wave function

$$\Psi_T(x) = \frac{\sqrt{\alpha}}{\pi^{1/4}} e^{-x^2\alpha^2/2}.$$

However, for the hydrogen atom we have $\rho \in [0, \infty]$, while for the harmonic oscillator we have $x \in [-\infty, \infty]$.

This has important consequences for the way we generate random positions. For the hydrogen atom we have a random position given by e.g.,

```
r_old = step_length*(ran1(&idum))/alpha;
```

which ensures that $\rho \geq 0$, while for the harmonic oscillator we have

```
r_old = step_length*(ran1(&idum)-0.5)/alpha;
```

in order to have $x \in [-\infty, \infty]$. This is however not implemented in the program below. There, importance sampling is not included. We simulate points in the x, y and z directions using random numbers generated by the uniform distribution and multiplied by the step length. Note that we have to define a step length in our calculations. Here one has to play around with different values for the step and as a rule of thumb (one of the golden Monte Carlo rules), the step length should be chosen so that roughly 50% of all new moves are accepted. In the program at the end of this section we have also scaled the random position with the variational parameter α. The reason for this particular choice is that we have an external loop over the variational parameter. Different variational parameters will obviously yield different acceptance rates if we use the same step length. An alternative to the code below is to perform the Monte Carlo sampling with just one variational parameter, and play around with different step lengths in order to achieve a reasonable acceptance ratio. Another possibility is to include a more advanced test which restarts the Monte Carlo sampling with a new step length if the specific variational parameter and chosen step length lead to a too low acceptance ratio.

In Figs. 12.1 and 12.2 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 12.1 and 12.2. In these tables we list the variance and the standard deviation as well. We note that at α we obtain the exact result, and the variance is zero, as it should. The reason is that we then have the exact wave function, and the action of the hamiltionan on the wave function

$$H\psi = \text{constant} \times \psi,$$

yields just a constant. The integral which defines various expectation values involving moments of the hamiltonian becomes then

$$\langle H^n \rangle = \frac{\int d\mathbf{R} \Psi_T^*(\mathbf{R}) H^n(\mathbf{R}) \Psi_T(\mathbf{R})}{\int d\mathbf{R} \Psi_T^*(\mathbf{R}) \Psi_T(\mathbf{R})} = \text{constant} \times \frac{\int d\mathbf{R} \Psi_T^*(\mathbf{R}) \Psi_T(\mathbf{R})}{\int d\mathbf{R} \Psi_T^*(\mathbf{R}) \Psi_T(\mathbf{R})} = \text{constant}. \quad (12.39)$$

This explains why the variance is zero for $\alpha = 1$. However, the hydrogen atom and the harmonic oscillator are some of the few cases where we can use a trial wave function proportional to the exact one. These two systems are also some of the few examples of cases where we can find an exact solution to the problem. In most cases of interest, we do not know a priori the exact wave function, or how to make a good trial wave function. In essentially all real problems a large amount of CPU time and numerical experimenting is needed in order to ascertain the validity of a Monte Carlo estimate. The next examples deal with such problems.

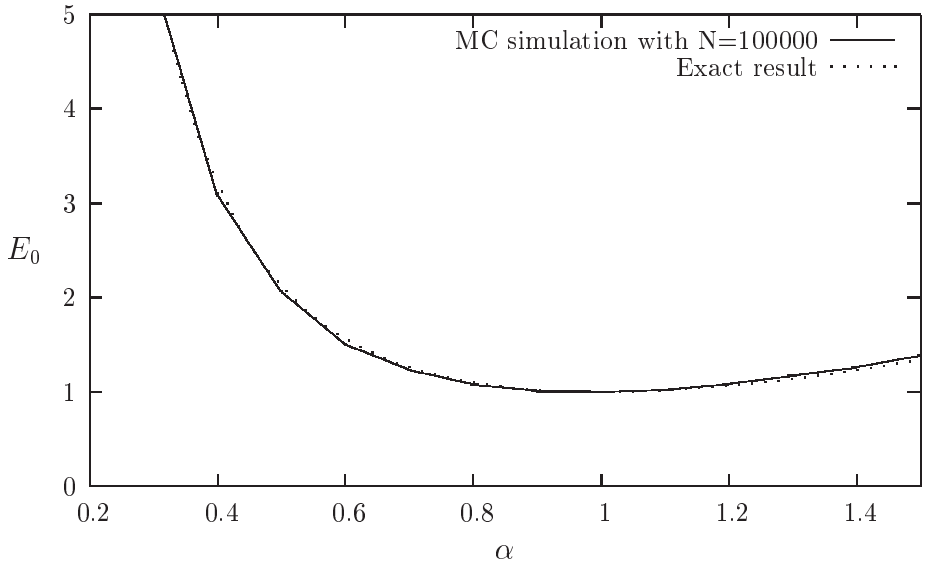


Figure 12.1: Result for ground state energy of the harmonic oscillator as function of the variational parameter α. The exact result is for $\alpha = 1$ with an energy $E = 1$. See text for further details

Table 12.1: 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 $E = 1$. The energy variance σ^2 and the standard deviation $\sigma/\sqrt{N}$ are also listed. The variable N is the number of Monte Carlo samples. In this calculation we set $N = 100000$ and a step length of 2 was used in order to obtain an acceptance of $\approx 50\%$.

α	$\langle H \rangle$	σ^2	$\sigma/\sqrt{N}$
5.00000E-01	2.06479E+00	5.78739E+00	7.60749E-03
6.00000E-01	1.50495E+00	2.32782E+00	4.82475E-03
7.00000E-01	1.23264E+00	9.82479E-01	3.13445E-03
8.00000E-01	1.08007E+00	3.44857E-01	1.85703E-03
9.00000E-01	1.01111E+00	7.24827E-02	8.51368E-04
1.00000E-00	1.00000E+00	0.00000E+00	0.00000E+00
1.10000E+00	1.02621E+00	5.95716E-02	7.71826E-04
1.20000E+00	1.08667E+00	2.23389E-01	1.49462E-03
1.30000E+00	1.17168E+00	4.78446E-01	2.18734E-03
1.40000E+00	1.26374E+00	8.55524E-01	2.92493E-03
1.50000E+00	1.38897E+00	1.30720E+00	3.61553E-03

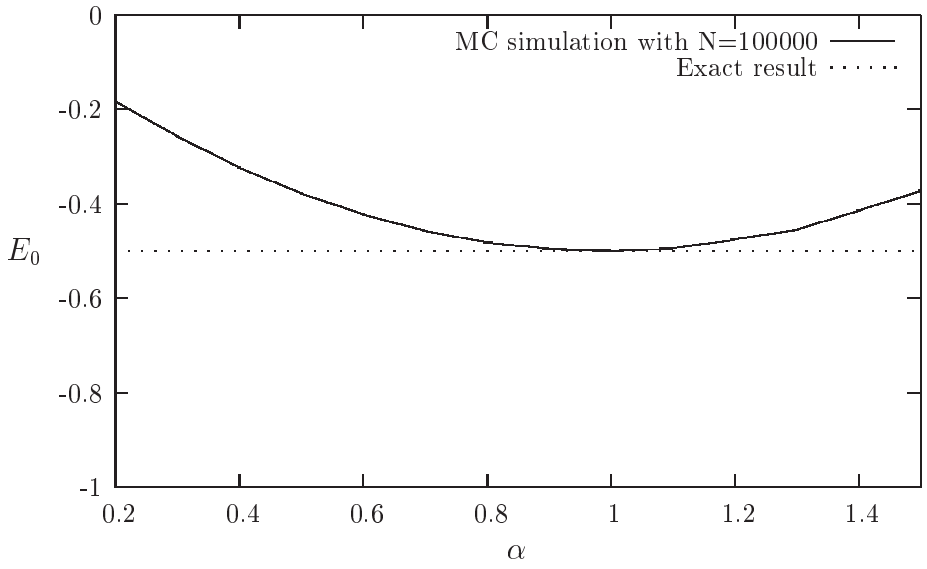


Figure 12.2: Result for ground state energy of the hydrogen atom as function of the variational parameter α. The exact result is for $\alpha = 1$ with an energy $E = -1/2$. See text for further details

Table 12.2: 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 $E = -1/2$. The energy variance σ^2 and the standard deviation $\sigma/\sqrt{N}$ are also listed. The variable N is the number of Monte Carlo samples. In this calculation we fixed $N = 100000$ and a step length of 4 Bohr radii was used in order to obtain an acceptance of $\approx 50\%$.

α	$\langle H \rangle$	σ^2	$\sigma/\sqrt{N}$
5.00000E-01	-3.76740E-01	6.10503E-02	7.81347E-04
6.00000E-01	-4.21744E-01	5.22322E-02	7.22718E-04
7.00000E-01	-4.57759E-01	4.51201E-02	6.71715E-04
8.00000E-01	-4.81461E-01	3.05736E-02	5.52934E-04
9.00000E-01	-4.95899E-01	8.20497E-03	2.86443E-04
1.00000E-00	-5.00000E-01	0.00000E+00	0.00000E+00
1.10000E+00	-4.93738E-01	1.16989E-02	3.42036E-04
1.20000E+00	-4.75563E-01	8.85899E-02	9.41222E-04
1.30000E+00	-4.54341E-01	1.45171E-01	1.20487E-03
1.40000E+00	-4.13220E-01	3.14113E-01	1.77232E-03
1.50000E+00	-3.72241E-01	5.45568E-01	2.33574E-03

12.2.4 A nucleon in a gaussian potential

This problem is a slight extension of the harmonic oscillator problem, since we are going to use an harmonic oscillator trial wave function.

The problem is that of a nucleon, a proton or neutron, in a nuclear medium, say a finite nucleus. The nuclear interaction is of an extreme short range compared to the more familiar Coulomb potential. Typically, a nucleon-nucleon interaction has a range of some few fermis, one fermi being 10^{-15} m (or just fm). Here we approximate the interaction between our lonely nucleon and the remaining nucleus with a gaussian potential

$$V(r) = V_0 e^{-r^2/a^2}, \tag{12.40}$$

where V_0 is a constant (fixed to $V_0 = -45$ MeV here) and the constant a represents the range of the potential. We set $a = 2$ fm. The mass of the nucleon is 938.926 MeV$/c^2$, with c the speed of light. This mass is the average of the proton and neutron masses. The constant in front of the kinetic energy operator is hence

$$\frac{\hbar^2}{m} = \frac{\hbar^2 c^2}{mc^2} = \frac{197.315^2}{938.926}\ \mathrm{MeVfm}^2 = 41.466\ \mathrm{MeVfm}^2. \tag{12.41}$$

We assume that the nucleon is in the $1s$ state and approximate the wave function of that a harmonic oscillator in the ground state, namely

$$\Psi_T(r) = \frac{\alpha^{3/2}}{\pi^{3/4}} e^{-r^2\alpha^2/2}. \tag{12.42}$$

This trial wave function results in the following local energy

$$E_L(r) = \frac{\hbar^2}{2m}(3\alpha^2 - r^2\alpha^4) + V(r). \tag{12.43}$$

With the wave function and the local energy we can obviously compute the variational energy from

$$\langle H \rangle = \int P(\mathbf{R}) E_L(\mathbf{R}) d\mathbf{R},$$

which yields a theoretical variational energy

$$\langle H \rangle = \frac{3\hbar^2}{4m}\alpha^2 + V_0 \left(\frac{(a\alpha)^2}{1+(a\alpha)^2} \right)^{3/2}. \tag{12.44}$$

Note well that this is not the exact energy from the above hamiltonian. The exact eigenvalue which follows from diagonalization of the Schrödinger equation is $E_0 = -16.3824$ MeV, which should be compared with the approximately ~ -9.2 MeV from Fig. **??**. The results are plotted as functions of the variational parameter α and compared them with the exact variational result of Eq. (12.44). The agreement is equally good as in the previous cases. However, the variance at the point where the energy reaches its minimum is different from zero, clearly indicating that the wave function we have chosen is not the exact one.

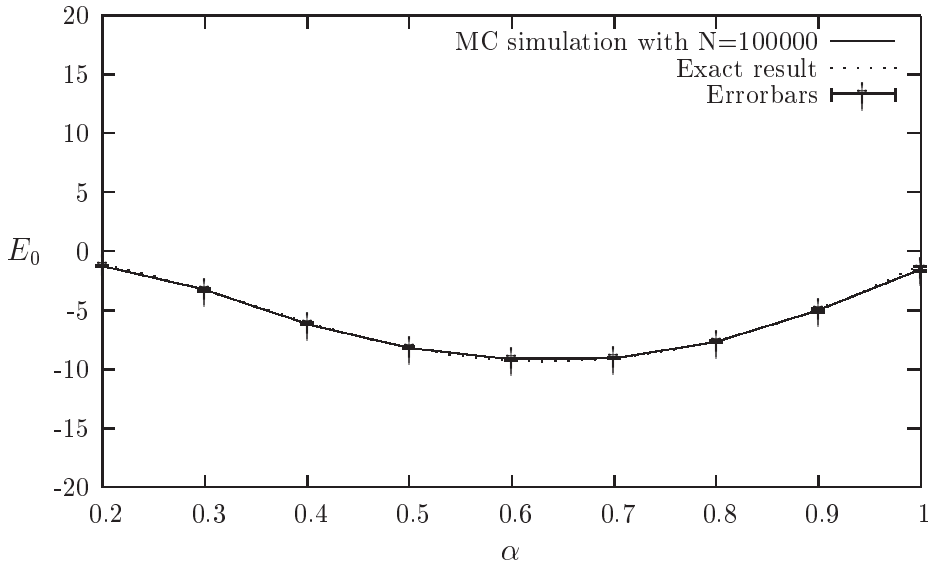


Figure 12.3: Results for the ground state energy of a nucleon in a gaussian potential as function of the variational parameter α. The exact variational result is also plotted.

12.2.5 The helium atom

Most physical problems of interest in atomic, molecular and solid state physics consist of a number of interacting electrons and ions. The total number of particles N is usually sufficiently large that an exact solution cannot be found. Typically, the expectation value for a chosen hamiltonian for a system of N particles is

$$\langle H\rangle = \frac{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N) H(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)\Psi(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)}{\int d\mathbf{R}_1 d\mathbf{R}_2 \dots d\mathbf{R}_N \Psi^*(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)\Psi(\mathbf{R_1},\mathbf{R}_2,\dots,\mathbf{R}_N)}, \tag{12.45}$$

an in general intractable problem. Controlled and well understood approximations are sought to reduce the complexity to a tractable level. Once the equations are solved, a large number of properties may be calculated from the wave function. Errors or approximations made in obtaining the wave function will be manifest in any property derived from the wave function. Where high accuracy is required, considerable attention must be paid to the derivation of the wave function and any approximations made.

The helium atom consists of two electrons and a nucleus with charge $Z = 2$. In setting up the hamiltonian of this system, we need to account for the repulsion between the two electrons as well.

A common and very reasonable approximation used in the solution of equation of the Schrödinger equation for systems of interacting electrons and ions is the Born-Oppenheimer approximation. In a system of interacting electrons and nuclei there will usually be little momentum transfer between the two types of particles due to their greatly differing masses. The forces between the

particles are of similar magnitude due to their similar charge. If one then assumes that the momenta of the particles are also similar, then the nuclei must have much smaller velocities than the electrons due to their far greater mass. On the time-scale of nuclear motion, one can therefore consider the electrons to relax to a ground-state with the nuclei at fixed locations. This separation of the electronic and nuclear degrees of freedom is known as the Born-Oppenheimer approximation. But even this simplified electronic Hamiltonian remains very difficult to solve. No analytic solutions exist for general systems with more than one electron.

If we label the distance between electron 1 and the nucleus as r_1. Similarly we have r_2 for electron 2. The contribution to the potential energy due to the attraction from the nucleus is

$$-\frac{2ke^2}{r_1}-\frac{2ke^2}{r_2}, \tag{12.46}$$

and if we add the repulsion arising from the two interacting electrons, we obtain the potential energy

$$V(r_1,r_2)=-\frac{2ke^2}{r_1}-\frac{2ke^2}{r_2}+\frac{ke^2}{r_{12}}, \tag{12.47}$$

with the electrons separated at a distance $r_{12}=|\mathbf{r}_1-\mathbf{r}_2|$. The hamiltonian becomes then

$$\widehat{\mathbf{H}}=-\frac{\hbar^2\nabla_1^2}{2m}-\frac{\hbar^2\nabla_2^2}{2m}-\frac{2ke^2}{r_1}-\frac{2ke^2}{r_2}+\frac{ke^2}{r_{12}}, \tag{12.48}$$

and Schrödingers equation reads

$$\widehat{\mathbf{H}}\psi=E\psi. \tag{12.49}$$

Note that this equation has been written in atomic units $a.u.$ which are more convenient for quantum mechanical problems. This means that the final energy has to be multiplied by a $2\times E_0$, where $E_0=13.6$ eV, the binding energy of the hydrogen atom.

A very simple first approximation to this system is to omit the repulsion between the two electrons. The potential energy becomes then

$$V(r_1,r_2)\approx-\frac{Zke^2}{r_1}-\frac{Zke^2}{r_2}. \tag{12.50}$$

The advantage of this approximation is that each electron can be treated as being independent of each other, implying that each electron sees just a centrally symmetric potential, or central field.

To see whether this gives a meaningful result, we set $Z=2$ and neglect totally the repulsion between the two electrons. Electron 1 has the following hamiltonian

$$\widehat{\mathbf{h}}_1=-\frac{\hbar^2\nabla_1^2}{2m}-\frac{2ke^2}{r_1}, \tag{12.51}$$

with pertinent wave function and eigenvalue

$$\widehat{\mathbf{h}}_1\psi_a=E_a\psi_a, \tag{12.52}$$

where $a = \{n_a l_a m_{l_a}\}$, are its quantum numbers. The energy E_a is

$$E_a = -\frac{Z^2 E_0}{n_a^2}, \tag{12.53}$$

med $E_0 = 13.6$ eV, being the ground state energy of the hydrogen atom. In a similar way, we obatin for electron 2

$$\widehat{\mathbf{h}}_2 = -\frac{\hbar^2 \nabla_2^2}{2m} - \frac{2ke^2}{r_2}, \tag{12.54}$$

with wave function

$$\widehat{\mathbf{h}}_2 \psi_b = E_b \psi_b, \tag{12.55}$$

and $b = \{n_b l_b m_{l_b}\}$, and energy

$$E_b = \frac{Z^2 E_0}{n_b^2}. \tag{12.56}$$

Since the electrons do not interact, we can assume that the ground state wave function of the helium atom is given by

$$\psi = \psi_a \psi_b, \tag{12.57}$$

resulting in the following approximation to Schrödinger's equation

$$\left(\widehat{\mathbf{h}}_1 + \widehat{\mathbf{h}}_2\right) \psi = \left(\widehat{\mathbf{h}}_1 + \widehat{\mathbf{h}}_2\right) \psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2) = E_{ab}\psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2). \tag{12.58}$$

The energy becomes then

$$\left(\widehat{\mathbf{h}}_1 \psi_a(\mathbf{r}_1)\right) \psi_b(\mathbf{r}_2) + \left(\widehat{\mathbf{h}}_2 \psi_b(\mathbf{r}_2)\right) \psi_a(\mathbf{r}_1) = \left(E_a + E_b\right) \psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2), \tag{12.59}$$

yielding

$$E_{ab} = Z^2 E_0 \left(\frac{1}{n_a^2} + \frac{1}{n_b^2}\right). \tag{12.60}$$

If we insert $Z = 2$ and assume that the ground state is determined by two electrons in the lowest-lying hydrogen orbit with $n_a = n_b = 1$, the energy becomes

$$E_{ab} = 8E_0 = -108.8 \text{ eV}, \tag{12.61}$$

while the experimental value is -78.8 eV. Clearly, this discrepancy is essentially due to our omission of the repulsion arising from the interaction of two electrons.

Choice of trial wave function

The choice of trial wave function is critical in VMC calculations. How to choose it is however a highly non-trivial task. All observables are evaluated with respect to the probability distribution

$$P(\mathbf{R}) = \frac{|\psi_T(\mathbf{R})|^2}{\int |\psi_T(\mathbf{R})|^2 d\mathbf{R}}. \tag{12.62}$$

generated by the trial wave function. The trial wave function must approximate an exact eigenstate in order that accurate results are to be obtained. Improved trial wave functions also improve the importance sampling, reducing the cost of obtaining a certain statistical accuracy.

Quantum Monte Carlo methods are able to exploit trial wave functions of arbitrary forms. Any wave function that is physical and for which the value, gradient and laplacian of the wave function may be efficiently computed can be used. The power of Quantum Monte Carlo methods lies in the flexibility of the form of the trial wave function.

It is important that the trial wave function satisfies as many known properties of the exact wave function as possible. A good trial wave function should exhibit much of the same features as does the exact wave function. Especially, it should be well-defined at the origin, that is $\Psi(|\mathbf{R}| = 0) \neq 0$, and its derivative at the origin should also be well-defined . One possible guideline in choosing the trial wave function is the use of constraints about the behavior of the wave function when the distance between one electron and the nucleus or two electrons approaches zero. These constraints are the so-called "cusp conditions" and are related to the derivatives of the wave function.

To see this, let us single out one of the electrons in the helium atom and assume that this electron is close to the nucleus, i.e., $r_1 \to 0$. We assume also that the two electrons are far from each other and that $r_2 \neq 0$. The local energy can then be written as

$$E_L(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})} H\psi_T(\mathbf{R}) = \frac{1}{\psi_T(\mathbf{R})}\left(-\frac{1}{2}\nabla_1^2 - \frac{Z}{r_1}\right)\psi_T(\mathbf{R}) + \text{finite terms}. \tag{12.63}$$

Writing out the kinetic energy term in the spherical coordinates of electron 1, we arrive at the following expression for the local energy

$$E_L(R) = \frac{1}{\mathcal{R}_T(r_1)}\left(-\frac{1}{2}\frac{d^2}{dr_1^2} - \frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1}\right)\mathcal{R}_T(r_1) + \text{finite terms}, \tag{12.64}$$

where $\mathcal{R}_T(r_1)$ is the radial part of the wave function for electron 1. We have also used that the orbital momentum of electron 1 is $l = 0$. For small values of r_1, the terms which dominate are

$$\lim_{r_1 \to 0} E_L(R) = \frac{1}{\mathcal{R}_T(r_1)}\left(-\frac{1}{r_1}\frac{d}{dr_1} - \frac{Z}{r_1}\right)\mathcal{R}_T(r_1), \tag{12.65}$$

since the second derivative does not diverge due to the finiteness of Ψ at the origin. The latter implies that in order for the kinetic energy term to balance the divergence in the potential term, we must have

$$\frac{1}{\mathcal{R}_T(r_1)}\frac{d\mathcal{R}_T(r_1)}{dr_1} = -Z, \tag{12.66}$$

implying that

$$\mathcal{R}_T(r_1) \propto e^{-Zr_1}. \tag{12.67}$$

A similar condition applies to electron 2 as well. For orbital momenta $l > 0$ we have (show this!)

$$\frac{1}{\mathcal{R}_T(r)}\frac{d\mathcal{R}_T(r)}{dr} = -\frac{Z}{l+1}. \tag{12.68}$$

Another constraint on the wave function is found for two electrons approaching each other. In this case it is the dependence on the separation r_{12} between the two electrons which has to reflect the correct behavior in the limit $r_{12} \rightarrow 0$. The resulting radial equation for the r_{12} dependence is the same for the electron-nucleus case, except that the attractive Coulomb interaction between the nucleus and the electron is replaced by a repulsive interaction and the kinetic energy term is twice as large. We obtain then

$$\lim_{r_{12}\rightarrow 0} E_L(R) = \frac{1}{\mathcal{R}_T(r_{12})}\left(-\frac{4}{r_{12}}\frac{d}{dr_{12}} + \frac{2}{r_{12}}\right)\mathcal{R}_T(r_{12}), \tag{12.69}$$

with still $l = 0$. This yields the so-called 'cusp'-condition

$$\frac{1}{\mathcal{R}_T(r_{12})}\frac{d\mathcal{R}_T(r_{12})}{dr_{12}} = \frac{1}{2}, \tag{12.70}$$

while for $l > 0$ we have

$$\frac{1}{\mathcal{R}_T(r_{12})}\frac{d\mathcal{R}_T(r_{12})}{dr_{12}} = \frac{1}{2(l+1)}. \tag{12.71}$$

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

$$\psi_T(\mathbf{R}) = e^{-\alpha(r_1+r_2)}, \tag{12.72}$$

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

$$\psi_T(\mathbf{R}) = e^{-\alpha(r_1+r_2)}e^{r_{12}/2}. \tag{12.73}$$

The last equation can be generalized to

$$\psi_T(\mathbf{R}) = \phi(\mathbf{r}_1)\phi(\mathbf{r}_2)\ldots\phi(\mathbf{r}_N)\prod_{i<j} f(r_{ij}), \tag{12.74}$$

for a system with N electrons or particles. The wave function $\phi(\mathbf{r}_i)$ is the single-particle wave function for particle i, while $f(r_{ij})$ account for more complicated two-body correlations. For the helium atom, we placed both electrons in the hydrogenic orbit $1s$. We know that the ground state for the helium atom has a symmetric spatial part, while the spin wave function is anti-symmetric in order to obey the Pauli principle. In the present case we need not to deal with spin degrees of freedom, since we are mainly trying to reproduce the ground state of the system. However, adopting such a single-particle representation for the individual electrons means that for atoms beyond helium, we cannot continue to place electrons in the lowest hydrogenic orbit. This is a consenquence of the Pauli principle, which states that the total wave function for a system of identical particles such as fermions, has to be anti-symmetric. The program we include below can use either Eq. (12.72) or Eq. (12.73) for the trial wave function. One or two electrons can be placed in the lowest hydrogen orbit, implying that the program can only be used for studies of the ground state of hydrogen or helium.

12.2.6 Program example for atomic systems

The VMC algorithm consists of two distinct phases. In the first a walker, a single electron in our case, consisting of an initially random set of electron positions is propagated according to the Metropolis algorithm, in order to equilibrate it and begin sampling . In the second phase, the walker continues to be moved, but energies and other observables are also accumulated for later averaging and statistical analysis. In the program below, the electrons are moved individually and not as a whole configuration. This improves the efficiency of the algorithm in larger systems, where configuration moves require increasingly small steps to maintain the acceptance ratio.

programs/chap12/program1.cpp

```
// Variational Monte Carlo for atoms with up to two electrons
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace  std;
// output file as global variable
ofstream ofile;
// the step length and its squared inverse for the second derivative
#define h 0.001
#define h2 1000000

// declaraton of functions

// Function to read in data from screen, note call by reference
void initialise(int&, int&, int&, int&, int&, int&, double&) ;

// The Mc sampling for the variational Monte Carlo
void  mc_sampling(int, int, int, int, int, int, double, double *,
   double *);

// The variational wave function
double  wave_function(double **, double, int, int);

// The local energy
double  local_energy(double **, double, double, int, int, int);

// prints to screen the results of the calculations
void  output(int, int, int, double *, double *);


// Begin of main program

//int main()
int main(int argc, char* argv[])
```

```
{
  char *outfilename;
  int number_cycles, max_variations, thermalization, charge;
  int dimension, number_particles;
  double step_length;
  double *cumulative_e, *cumulative_e2;

  // Read in output file, abort if there are too few command-line
     arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  //    Read in data
  initialise(dimension, number_particles, charge,
             max_variations, number_cycles,
             thermalization, step_length) ;
  cumulative_e = new double[max_variations+1];
  cumulative_e2 = new double[max_variations+1];

  //  Do the mc sampling
  mc_sampling(dimension, number_particles, charge,
              max_variations, thermalization,
              number_cycles, step_length, cumulative_e, cumulative_e2)
                 ;
  // Print out results
  output(max_variations, number_cycles, charge, cumulative_e,
     cumulative_e2);
  delete [] cumulative_e; delete [] cumulative_e;
  ofile.close();  // close output file
  return 0;
}
```

```
// 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
```

```
// 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;
}
```

```
// 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;
}
```

```
void initialise(int& dimension, int& number_particles, int& charge,
                int& max_variations, int& number_cycles,
                int& thermalization, double& step_length)
{
  cout << "number of particles = ";
  cin >> number_particles;
  cout << "charge of nucleus = ";
  cin >> charge;
  cout << "dimensionality = ";
  cin >> dimension;
  cout << "maximum variational parameters = ";
  cin >> max_variations;
  cout << "# Thermalization  steps= ";
  cin >> thermalization;
  cout << "# MC steps= ";
  cin >> number_cycles;
  cout << "# step length= ";
  cin >> step_length;
}  // end of function initialise

void output(int max_variations, int number_cycles, int charge,
            double *cumulative_e, double *cumulative_e2)
{
  int i;
```

```
  double alpha, variance, error;
  alpha = 0.5*charge;
  for( i=1; i <= max_variations; i++){
    alpha += 0.1;
    variance = cumulative_e2[i]-cumulative_e[i]*cumulative_e[i];
    error=sqrt(variance/number_cycles);
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << setw(15) << setprecision(8) << alpha;
    ofile << setw(15) << setprecision(8) << cumulative_e[i];
    ofile << setw(15) << setprecision(8) << variance;
    ofile << setw(15) << setprecision(8) << error << endl;
  }
//  fclose (output_file);
}  // end of function output
```

In the program above one has to the possibility to study both the hydrogen atom and the helium atom by setting the number of particles to either 1 or 2. In addition, we have not used the analytic expression for the kinetic energy in the evaluation of the local energy. Rather, we have used the numerical expression of Eq. (3.15), i.e.,

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

in order to compute

$$-\frac{1}{2\psi_T(\mathbf{R})}\nabla^2\psi_T(\mathbf{R}). \tag{12.75}$$

The variable h is a chosen step length. For helium, since it is rather easy to evaluate the local energy, the above is an unnecessary complication. However, for many-electron or other many-particle systems, the derivation of an analytic expression for the kinetic energy can be quite involved, and the numerical evaluation of the kinetic energy using Eq. (3.15) may result in a simpler code and/or even a faster one. The way we have rewritten Schrödinger's equation results in energies given by 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. (12.72) for the trial wave function, we obtain an energy minimum at $\alpha \approx 1.75$. The ground state is $E = -2.85$ in atomic units or $E = -77.5$ eV. The experimental value is -78.8 eV. Obviously, improvements to the wave function such as including the 'cusp'-condition for the two electrons as well, see Eq. (12.73), 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. 12.4 that the variance is not zero at the energy minimum.

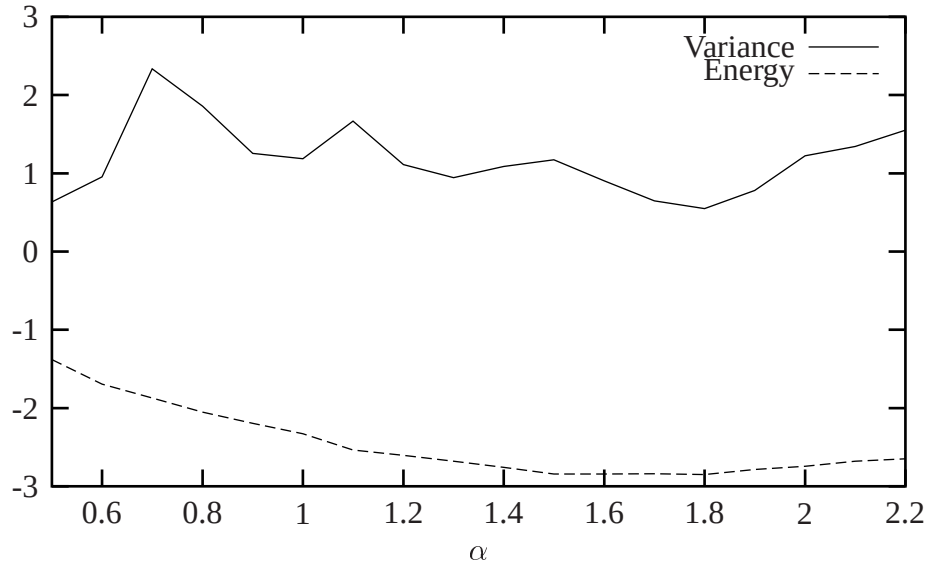


Figure 12.4: Result for ground state energy of the helium atom using Eq. (12.72) for the trial wave function. The variance is also plotted. A total of 100000 Monte Carlo moves were used with a step length of 2 Bohr radii.

12.3 Simulation of molecular systems

12.3.1 The H_2^+ molecule

The H_2^+ molecule consists of two protons and one electron, with binding energy $E_B = -2.8$ eV and an equilibrium position $r_0 = 0.106$ nm between the two protons.

We define our system through the following variables. The electron is at a distance $\mathbf{r}$ from a chosen origo, one of the protons is at the distance $-\mathbf{R}/2$ while the other one is placed at $\mathbf{R}/2$ from origo, resulting in a distance to the electron of $\mathbf{r} - \mathbf{R}/2$ and $\mathbf{r} + \mathbf{R}/2$, respectively.

In our solution of Schrödinger's equation for this system we are going to neglect the kinetic energies of the protons, since they are 2000 times heavier than the electron. We assume thus that their velocities are negligible compared to the velocity of the electron. In addition we omit contributions from nuclear forces, since they act at distances of several orders of magnitude smaller than the equilibrium position.

We can then write Schrödinger's equation as follows

$$\left\{ -\frac{\hbar^2 \nabla_r^2}{2m_e} - \frac{ke^2}{|\mathbf{r} - \mathbf{R}/2|} - \frac{ke^2}{|\mathbf{r} + \mathbf{R}/2|} + \frac{ke^2}{R} \right\} \psi(\mathbf{r}, \mathbf{R}) = E\psi(\mathbf{r}, \mathbf{R}), \tag{12.76}$$

where the first term is the kinetic energy of the electron, the second term is the potential energy the electron feels from the proton at $-\mathbf{R}/2$ while the third term arises from the potential energy contribution from the proton at $\mathbf{R}/2$. The last term arises due to the repulsion between the two

protons. In Fig. 12.5 we show a plot of the potential energy

$$V(\mathbf{r},\mathbf{R}) = -\frac{ke^2}{|\mathbf{r}-\mathbf{R}/2|} - \frac{ke^2}{|\mathbf{r}+\mathbf{R}/2|} + \frac{ke^2}{R}. \tag{12.77}$$

Here we have fixed $|\mathbf{R}| = 2a_0$ og $|\mathbf{R}| = 8a_0$, being 2 and 8 Bohr radii, respectively. Note that in the region between $|\mathbf{r}| = -|\mathbf{R}|/2$ (units are r/a_0 in this figure, with $a_0 = 0.0529$) and $|\mathbf{r}| = |\mathbf{R}|/2$ the electron can tunnel through the potential barrier. Recall that $-\mathbf{R}/2$ og $\mathbf{R}/2$ correspond to the positions of the two protons. We note also that if R is increased, the potential becomes less attractive. This has consequences for the binding energy of the molecule. The binding energy decreases as the distance $\mathbf{R}$ increases. Since the potential is symmetric with

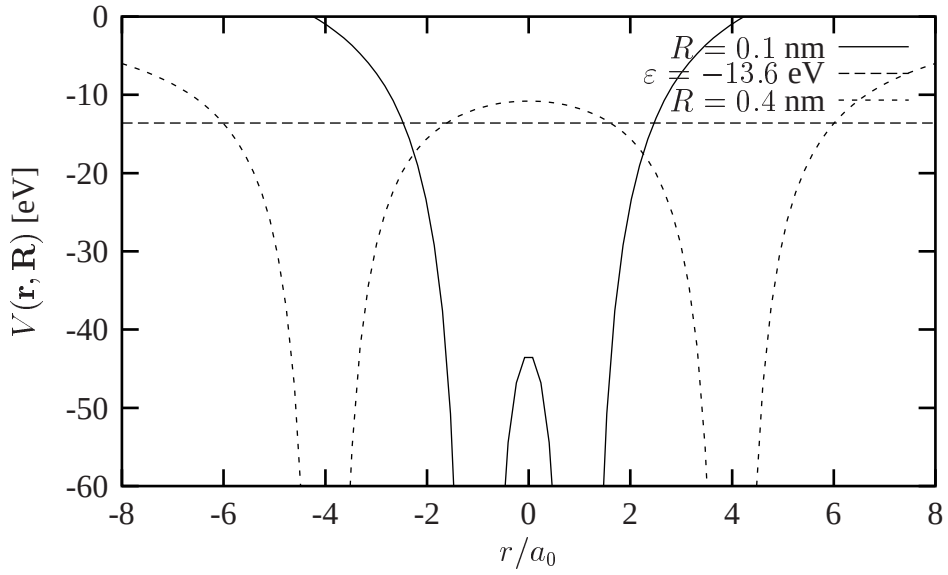


Figure 12.5: Plot of $V(r,R)$ for $|\mathbf{R}|$=0.1 and 0.4 nm. Units along the x-axis are r/a_0 . The straight line is the binding energy of the hydrogen atom, $\varepsilon = -13.6$ eV.

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

$$\psi_{100}(r) = \left(\frac{1}{\pi a_0^3}\right)^{1/2} e^{-r/a_0}. \tag{12.78}$$

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

$$\psi_1(\mathbf{r},\mathbf{R}) = \left(\frac{1}{\pi a_0^3}\right)^{1/2} e^{-|\mathbf{r}-\mathbf{R}/2|/a_0}, \tag{12.79}$$

and

$$\psi_2(\mathbf{r},\mathbf{R}) = \left(\frac{1}{\pi a_0^3}\right)^{1/2} e^{-|\mathbf{r}+\mathbf{R}/2|/a_0}. \tag{12.80}$$

Based on these two wave functions, which represent where the electron can be, we attempt at the following linear combination

$$\psi_\pm(\mathbf{r},\mathbf{R}) = C_\pm\left(\psi_1(\mathbf{r},\mathbf{R}) \pm \psi_2(\mathbf{r},\mathbf{R})\right), \tag{12.81}$$

with $C_\pm$ a constant.

12.3.2 Physics project: the H_2 molecule

in preparation

12.4 Many-body systems

12.4.1 Liquid ^{4}He

Liquid ^{4}He is an example of a so-called extended system, with an infinite number of particles. The density of the system varies from dilute to extremely dense. It is fairly obvious that we cannot attempt a simulation with such degrees of freedom. There are however ways to circumvent this problem. The usual way of dealing with such systems, using concepts from statistical Physics, consists in representing the system in a simulation cell with e.g., periodic boundary conditions, as we did for the Ising model. If the cell has length L, the density of the system is determined by putting a given number of particles N in a simulation cell with volume L^3. The density becomes then $\rho = N/L^3$.

In general, when dealing with such systems of many interacting particles, the interaction itself is not known analytically. Rather, we will have to rely on parametrizations based on e.g., scattering experiments in order to determine a parametrization of the potential energy. The interaction between atoms and/or molecules can be either repulsive or attractive, depending on the distance R between two atoms or molecules. One can approximate this interaction as

$$V(R) = -\frac{A}{R^m} - \frac{B}{R^n}, \tag{12.82}$$

where m, n are some integers and A, B constans with dimension energy and length, and with units in e.g., eVnm. The constants A, B and the integers m, n are determined by the constraints

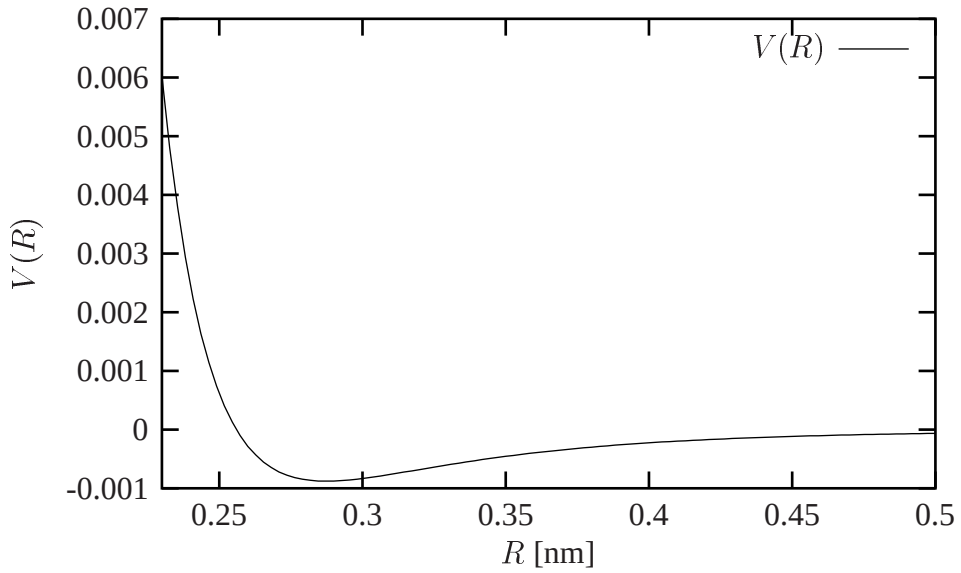


Figure 12.6: Plot for the Van der Waals interaction between helium atoms. The equilibrium position is $r_0 = 0.287$ nm.

that we wish to reproduce both scattering data and the binding energy of say a given molecule. It is thus an example of a parametrized interaction, and does not enjoy the status of being a fundamental interaction such as the Coulomb interaction does.

A well-known parametrization is the so-called Lennard-Jones potential

$$V_{LJ}(R) = 4\epsilon\left\{\left(\frac{\sigma}{R}\right)^{12} - \left(\frac{\sigma}{R}\right)^{6}\right\}, \tag{12.83}$$

where $\epsilon = 8.79 \times 10^{-4}$ eV and $\sigma = 0.256$ nm for helium atoms. Fig. 12.6 displays this interaction model. The interaction is both attractive and repulsive and exhibits a minimum at r_0. The reason why we have repulsion at small distances is that the electrons in two different helium atoms start repelling each other. In addition, the Pauli exclusion principle forbids two electrons to have the same set of quantum numbers.

Let us now assume that we have a simple trial wave function of the form

$$\psi_T(\mathbf{R}) = \prod_{i<j}^{N} f(r_{ij}), \tag{12.84}$$

where we assume that the correlation function $f(r_{ij})$ can be written as

$$f(r_{ij}) = e^{-\frac{1}{2}(b/r_{ij})^n}, \tag{12.85}$$

with b being the only variational parameter. Can we fix the value of n using the 'cusp'-conditions discussed in connection with the helium atom? We see from the form of the potential, that it

diverges at small interparticle distances. Since the energy is finite, it means that the kinetic energy term has to cancel this divergence at small r. Let us assume that electrons i and j are very close to each other. For the sake of convenience, we replace $r_{ij} = r$. At small r we require then that

$$-\frac{1}{f(r)}\nabla^2 f(r) + V(r) = 0. \tag{12.86}$$

In the limit $r \to 0$ we have

$$-\frac{n^2 b^n}{4r^{2n+2}} + \left(\frac{\sigma}{r}\right)^{12} = 0, \tag{12.87}$$

resulting in $n = 5$ and thus

$$f(r_{ij}) = e^{-\frac{1}{2}(b/r_{ij})^5}, \tag{12.88}$$

with

$$\psi_T(\mathbf{R}) = \prod_{i<j}^{N} e^{-\frac{1}{2}(b/r_{ij})^5}, \tag{12.89}$$

as trial wave function. We can rewrite the above equation as

$$\psi_T(\mathbf{R}) = e^{-\frac{1}{2}\sum_{i<j}^{N}(b/r_{ij})^5} = e^{-\frac{1}{2}\sum_{i<j}^{N} u(r_{ij})}, \tag{12.90}$$

with

$$u(r_{ij}) = (b/r_{ij})^5 .$$

For this variational wave function, the analytical expression for the local energy is rather simple. The tricky part comes again from the kinetic energy given by

$$-\frac{1}{\psi_T(\mathbf{R})}\nabla^2\psi_T(\mathbf{R}). \tag{12.91}$$

It is possible to show, after some tedious algebra, that

$$-\frac{1}{\psi_T(\mathbf{R})}\nabla^2\psi_T(\mathbf{R}) = -\sum_{k=1}^{N}\frac{1}{\psi_T(\mathbf{R})}\nabla_k^2\psi_T(\mathbf{R}) = -10b^5\sum_{i<k}^{N}\frac{1}{r_{ik}^7}. \tag{12.92}$$

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.

12.4.2 Bose-Einstein condensation

in preparation

12.4.3 Quantum dots

in preparation

12.4.4 Multi-electron atoms

in preparation

Chapter 13

Eigensystems

13.1 Introduction

In this chapter we discuss methods which are useful in solving eigenvalue problems in physics.

13.2 Eigenvalue problems

Let us consider the matrix **A** of dimension n. The eigenvalues of **A** is defined through the matrix equation

$$\mathbf{A}\mathbf{x}^{(\nu)} = \lambda^{(\nu)}\mathbf{x}^{(\nu)}, \tag{13.1}$$

where $\lambda^{(\nu)}$ are the eigenvalues and $\mathbf{x}^{(\nu)}$ the corresponding eigenvectors. This is equivalent to a set of n equations with n unknowns x_i

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= \lambda x_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= \lambda x_2 \\ \cdots & \quad \cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= \lambda x_n. \end{aligned}$$

W can rewrite eq (13.1) as

$$\left(\mathbf{A} - \lambda^{(\nu)}I\right)\mathbf{x}^{(\nu)} = 0,$$

with I being the unity matrix. This equation provides a solution to the problem if and only if the determinant is zero, namely

$$\left|\mathbf{A} - \lambda^{(\nu)}\mathbf{I}\right| = 0,$$

which in turn means that the determinant is a polynomial of degree n in λ and in general we will have n distinct zeros, viz.,

$$P_\lambda = \prod_{i=1}^{n}\left(\lambda_i - \lambda\right).$$

Procedures based on these ideas con be used if only a small fraction of all eigenvalues and eigenvectors are required, but the standard approach to solve eq. (13.1) is to perform a given number of similarity transformations so as to render the original matrix $\mathbf{A}$ in: 1) a diagonal form or: 2) as a tri-diagonal matrix which then can be be diagonalized by computational very effective procedures.

The first method leads us to e.g., Jacobi's method whereas the second one is e.g., given by Householder's algorithm for tri-diagonal transformations. We will discuss both methods below.

13.2.1 Similarity transformations

In the present discussion we assume that our matrix is real and symmetric, although it is rather straightforward to extend it to the case of a hermitian matrix. The matrix $\mathbf{A}$ has n eigenvalues $\lambda_1 \dots \lambda_n$ (distinct or not). Let $\mathbf{D}$ be the diagonal matrix with the eigenvalues on the diagonal

$$\mathbf{D} = \begin{pmatrix} \lambda_1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \lambda_{n-1} & \\ 0 & \dots & \dots & \dots & \dots & 0 & \lambda_n \end{pmatrix}. \tag{13.2}$$

The algorithm behind all current methods for obtaning eigenvalues is to perform a series of similarity transformations on the original matrix $\mathbf{A}$ to reduce it either into a diagonal form as above or into a tri-diagonal form.

We say that a matrix $\mathbf{B}$ is a similarity transform of $\mathbf{A}$ if

$$\mathbf{B} = \mathbf{S}^T\mathbf{A}\mathbf{S}, \qquad \text{where} \qquad \mathbf{S}^T\mathbf{S} = \mathbf{S}^{-1}\mathbf{S} = \mathbf{I}. \tag{13.3}$$

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, suppose that

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \qquad \text{and} \qquad \mathbf{B} = \mathbf{S}^T\mathbf{A}\mathbf{S}. \tag{13.4}$$

Multiply the first equation on the left by $\mathbf{S}^T$ and insert $\mathbf{S}^T\mathbf{S} = \mathbf{I}$ between $\mathbf{A}$ and $\mathbf{x}$. Then we get

$$(\mathbf{S^T}\mathbf{A}\mathbf{S})(\mathbf{S^T}\mathbf{x}) = \lambda\mathbf{S^T}\mathbf{x}, \tag{13.5}$$

which is the same as

$$\mathbf{B}\left(\mathbf{S^T}\mathbf{x}\right) = \lambda\left(\mathbf{S^T}\mathbf{x}\right). \tag{13.6}$$

Thus λ is an eigenvalue of $\mathbf{B}$ as well, but with eigenvector $\mathbf{S^T}\mathbf{x}$.

Now the basic philosophy is to

- either apply subsequent similarity transformations so that

$$\mathbf{S_N^T} \dots \mathbf{S_1^T}\mathbf{A}\mathbf{S_1} \dots \mathbf{S_N} = \mathbf{D}, \tag{13.7}$$

- or apply subsequent similarity transformations so that $\mathbf{A}$ becomes tri-diagonal. Thereafter, techniques for obtaining eigenvalues from tri-diagonal matrices can be used.

Let us look at the first method, better known as Jacobi's method.

13.2.2 Jacobi's method

Consider a $(n \times n)$ orthogonal transformation matrix

$$\mathbf{Q} = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\ 0 & 0 & \dots & cos\theta & 0 & \dots & 0 & sin\theta \\ 0 & 0 & \dots & 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\ 0 & 0 & \dots & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & -sin\theta & \dots & \dots & 0 & cos\theta \end{pmatrix} \tag{13.8}$$

with property $\mathbf{Q^T} = \mathbf{Q^{-1}}$. It performs a plane rotation around an angle θ in the Euclidean $n-$dimensional space. It means that its matrix elements different from zero are given by

$$q_{kk} = q_{ll} = cos\theta, q_{kl} = -q_{lk} = -sin\theta, q_{ii} = -q_{ii} = 1 \quad i \neq k \quad i \neq l, \tag{13.9}$$

A similarity transformation

$$\mathbf{B} = \mathbf{Q}^T\mathbf{A}\mathbf{Q}, \tag{13.10}$$

results in

$$\begin{aligned} b_{ik} &= a_{ik}cos\theta - a_{il}sin\theta, i \neq k, i \neq l \\ b_{il} &= a_{il}cos\theta + a_{ik}sin\theta, i \neq k, i \neq l \\ b_{kk} &= a_{kk}cos^2\theta - 2a_{kl}cos\theta sin\theta + a_{ll}sin^2\theta \\ b_{ll} &= a_{ll}cos^2\theta + 2a_{kl}cos\theta sin\theta + a_{kk}sin^2\theta \\ b_{kl} &= (a_{kk} - a_{ll})cos\theta sin\theta + a_{kl}(cos^2\theta - sin^2\theta) \end{aligned} \tag{13.11}$$

The angle θ is arbitrary. Now the recipe is to choose θ so that all non-diagonal matrix elements b_{pq} become zero which gives

$$tan2\theta = \frac{2a_{kl}}{a_{kk} - a_{ll}}. \tag{13.12}$$

If the denominator is zero, we can choose $\theta = \pm\pi/4$. Having defined θ through $z = tan2\theta$, we do not need to evaluate the other trigonometric functions, we can simply use relations like e.g.,

$$cos^2\theta = \frac{1}{2}\left(1 + \frac{1}{\sqrt{1+z^2}}\right), \tag{13.13}$$

and

$$sin^2\theta = \frac{1}{2}\left(1 - \frac{1}{\sqrt{1+z^2}}\right). \tag{13.14}$$

The algorithm is then quite simple. We perform a number of iterations untill the sum over the squared non-diagonal matrix elements are less than a prefixed test (ideally equal zero). The algorithm is more or less foolproof for all real symmetric matrices, but becomes much slower than methods based on tri-diagonalization for large matrices. We do therefore not recommend the use of this method for large scale problems. The philosophy however, performing a series of similarity transformations pertains to all current models for matrix diagonalization.

13.2.3 Diagonalization through the Householder's method for tri-diagonalization

In this case the energy diagonalization is performed in two steps: First, the matrix is transformed into a tri-diagonal form by the Householder similarity transformation and second, the tri-diagonal matrix is then diagonalized. The reason for this two-step process is that diagonalising a tri-diagonal matrix is computational much faster then the corresponding diagonalization of a general symmetric matrix. Let us discuss the two steps in more detail.

The Householder's method for tri-diagonalization

The first step consists in finding an orthogonal matrix $\mathbf{Q}$ which is the product of $(n-2)$ orthogonal matrices

$$\mathbf{Q} = \mathbf{Q}_1\mathbf{Q}_2\dots\mathbf{Q}_{n-2}, \tag{13.15}$$

each of which successively transforms one row and one column of $\mathbf{A}$ into the required tri-diagonal form. Only $n-2$ transformations are required, since the last two elements are already in tri-diagonal form. In order to determine each $\mathbf{Q_i}$ let us see what happens after the first multiplication, namely,

$$\mathbf{Q}_1^T\mathbf{A}\mathbf{Q}_1 = \begin{pmatrix} a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & a'_{22} & a'_{22} & \dots & \dots & \dots & a'_{2n} \\ 0 & a'_{32} & a'_{33} & \dots & \dots & \dots & a'_{3n} \\ 0 & \dots & \dots & \dots & \dots & \dots & \\ 0 & a'_{n2} & a'_{n3} & \dots & \dots & \dots & a'_{nn} \end{pmatrix} \tag{13.16}$$

where the primed quantities represent a matrix $\mathbf{A}'$ of dimension $n-1$ which will subsequentely be transformed by $\mathbf{Q_2}$. The factor e_1 is a possibly non-vanishing element. The next transformation produced by $\mathbf{Q_2}$ has the same effect as $\mathbf{Q_1}$ but now on the submatirx $\mathbf{A}^{'}$ only

$$\left(\mathbf{Q}_1\mathbf{Q}_2\right)^T\mathbf{A}\mathbf{Q}_1\mathbf{Q}_2 = \begin{pmatrix} a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & a'_{22} & e_2 & 0 & \dots & \dots & 0 \\ 0 & e_2 & a''_{33} & \dots & \dots & \dots & a''_{3n} \\ 0 & \dots & \dots & \dots & \dots & \dots & \\ 0 & 0 & a''_{n3} & \dots & \dots & \dots & a''_{nn} \end{pmatrix} \tag{13.17}$$

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 performed on the full size of the original matrix.

After a series of such transformations, we end with a set of diagonal matrix elements

$$a_{11}, a'_{22}, a''_{33}\dots a^{n-1}_{nn}, \tag{13.18}$$

and off-diagonal matrix elements

$$e_1, e_2, e_3, \dots, e_{n-1}. \tag{13.19}$$

The resulting matrix reads

$$\mathbf{Q}^T\mathbf{A}\mathbf{Q} = \begin{pmatrix} a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & a'_{22} & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & a''_{33} & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & a^{(n-1)}_{n-2} & e_{n-1} \\ 0 & \dots & \dots & \dots & \dots & e_{n-1} & a^{(n-1)}_{n-1} \end{pmatrix}. \tag{13.20}$$

Now it remains to find a recipe for determining the transformation $\mathbf{Q}_n$ all of which has basicly the same form, but operating on a lower dimensional matrix. We illustrate the method for $\mathbf{Q}_1$ which we assume takes the form

$$\mathbf{Q_1} = \begin{pmatrix} 1 & \mathbf{0^T} \\ \mathbf{0} & \mathbf{P} \end{pmatrix}, \tag{13.21}$$

with $\mathbf{0^T}$ being a zero row vector, $\mathbf{0^T} = \{0, 0, \cdots\}$ of dimension $(n-1)$. The matrix $\mathbf{P}$ is symmetric with dimension $((n-1)\times(n-1))$ satisfying $\mathbf{P}^2 = \mathbf{I}$ and $\mathbf{P}^T = \mathbf{P}$. A possible choice which fullfils the latter two requirements is

$$\mathbf{P} = \mathbf{I} - 2\mathbf{u}\mathbf{u}^T, \tag{13.22}$$

where $\mathbf{I}$ is the $(n-1)$ unity matrix and $\mathbf{u}$ is an $n-1$ column vector with norm $\mathbf{u}^T\mathbf{u}$(inner product. Note that $\mathbf{u}\mathbf{u}^T$ is an outer product giving a awith dimension $((n-1)\times(n-1))$. Each matrix element of $\mathbf{P}$ then reads

$$P_{ij} = \delta_{ij} - 2u_i u_j, \tag{13.23}$$

where i and j range from 1 to $n-1$. Applying the transformation $\mathbf{Q}_1$ results in

$$\mathbf{Q}_1^T\mathbf{A}\mathbf{Q}_1 = \begin{pmatrix} a_{11} & (\mathbf{P}\mathbf{v})^T \\ \mathbf{P}\mathbf{v} & \mathbf{A}' \end{pmatrix}, \tag{13.24}$$

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

$$\mathbf{P}\mathbf{v} = \mathbf{v} - 2\mathbf{u}(\mathbf{u}^T\mathbf{v}) = k\mathbf{e}, \tag{13.25}$$

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

$$(\mathbf{P}\mathbf{v})^T\mathbf{P}\mathbf{v} = k^2 = \mathbf{v}^T\mathbf{v} = |v|^2 = \sum_{i=2}^{n} a_{i1}^2, \tag{13.26}$$

which determines the constant $k = \pm v$. Nowwe can rewrite Eq. (13.25) as

$$\mathbf{v} - k\mathbf{e} = 2\mathbf{u}(\mathbf{u}^T\mathbf{v}), \tag{13.27}$$

and taking the scalar product of this equation with itself and obtain

$$2(\mathbf{u}^T\mathbf{v})^2 = (v^2 \pm a_{21}v), \tag{13.28}$$

which finally determines

$$\mathbf{u} = \frac{\mathbf{v}-\mathbf{e}}{2(\mathbf{u}^T\mathbf{v})}. \tag{13.29}$$

In solving Eq. (13.28) 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 tri-diagonal matrix suitable for obtaining the eigenvalues.

Diagonalization of a tri-diagonal matrix

The matrix is now transformed into tri-diagonal form and the last step is to transform it into a diagonal matrix giving the eigenvalues on the diagonal. The programs which performs these transformations are matrix $\mathbf{A} \longrightarrow$ tri-diagonal matrix $\longrightarrow$ diagonal matrix

C:	void trd2(double $**$a, int n, double d[], double e[])
	void tqli(double d[], double[], int n, double $**$z)
Fortran:	CALL tred2(a, n, d, e)
	CALL tqli(d, e, n, z)

The last step through the function *tqli()* involves several techniqcal details, but let us describe the basic idea in a four-dimensional example. The current tri-diagonal matrix takes the form

$$\mathbf{A} = \begin{pmatrix} d_1 & e_1 & 0 & 0 \\ e_1 & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{pmatrix}.$$

As a first observation, if any of the elements e_i are zero the matrix can be separated into smaller pieces before diagonalization. Specifically, if $e_1 = 0$ then d_1 is an eigenvalue. Thus, let us introduce a transformation $\mathbf{Q_1}$

$$\mathbf{Q_1} = \begin{pmatrix} \cos\theta & 0 & 0 & \sin\theta \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -\sin\theta & 0 & 0 & \cos\theta \end{pmatrix}$$

Then the similarity transformation

$$\mathbf{Q_1^T A Q_1} = \mathbf{A}' = \begin{pmatrix} d_1' & e_1' & 0 & 0 \\ e_1' & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e'3 \\ 0 & 0 & e_3' & d_4' \end{pmatrix}$$

produces a matrix where the primed elements in $\mathbf{A}'$ has 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.

13.3 Schrödinger's equation (SE) through diagonalization

Instead of solving the SE 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 SE as a matrix diagonalization problem, let us study the radial part of the SE. The radial part of the wave function, $R(r)$, is a solution to

$$-\frac{\hbar^2}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr}-\frac{l(l+1)}{r^2}\right)R(r)+V(r)R(r)=ER(r). \tag{13.30}$$

Then we substitute $R(r) = (1/r)u(r)$ and obtain

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r)+\left(V(r)+\frac{l(l+1)}{r^2}\frac{\hbar^2}{2m}\right)u(r)=Eu(r). \tag{13.31}$$

We introduce a dimensionless variable $\rho = (1/\alpha)r$ where α is a constant with dimension length and get

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(r)+\left(V(\rho)+\frac{l(l+1)}{\rho^2}\frac{\hbar^2}{2m\alpha^2}\right)u(\rho)=Eu(\rho). \tag{13.32}$$

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

$$V(x)=\frac{1}{2}kx^2, \tag{13.33}$$

with k being a constant. In our solution we will use units so that $k = \hbar = m = \alpha = 1$ and the SE for the one-dimensional harmonic oscillator becomes

$$-\frac{d^2}{dx^2}u(x)+x^2u(x)=2Eu(x). \tag{13.34}$$

Let us now see how we can rewrite this equation as a matrix eigenvalue problem. First we need to compute the second derivative. We use here the following expression for the second derivative of a function f

$$f''=\frac{f(x+h)-2f(x)+f(x-h)}{h^2}+O(h^2), \tag{13.35}$$

where h is our step. Next we define minimum and maximum values for the variable x, $R_{\rm min}$ and $R_{\rm max}$, respectively. With a given number of steps, $N_{\rm step}$, we then define the step h as

$$h = \frac{R_{\rm max} - R_{\rm min}}{N_{\rm step}}. \tag{13.36}$$

If we now define an arbitrary value of x as

$$x_i = R_{\rm min} + ih \qquad i = 1, 2, \ldots, N_{\rm step} - 1 \tag{13.37}$$

we can rewrite the SE for x_i as

$$-\frac{u(x_k + h) - 2u(x_k) + u(x_k - h)}{h^2} + x_k^2 u(x_k) = 2Eu(x_k), \tag{13.38}$$

or in a more compact way

$$-\frac{u_{k+1} - 2u_k + u_{k-1}}{h^2} + x_k^2 u_k = -\frac{u_{k+1} - 2u_k + u_{k-1}}{h^2} + V_k u_k = 2Eu_k, \tag{13.39}$$

where $u_k = u(x_k)$, $u_{k\pm1} = u(x_k \pm h)$ and $V_k = x_k^2$, the given potential. Let us see how this recipe may lead to a matrix reformulation of the SE. Define first the diagonal matrix element

$$d_k = \frac{2}{h^2} + V_k, \tag{13.40}$$

and the non-diagonal matrix element

$$e_k = -\frac{1}{h^2}. \tag{13.41}$$

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 SE takes the following form

$$d_k u_k + e_{k-1} u_{k-1} + e_{k+1} u_{k+1} = 2Eu_k, \tag{13.42}$$

where u_k is unknown. Since we have $N_{\rm step} - 1$ values of k we can write the latter equation as a matrix eigenvalue problem

$$\begin{pmatrix} d_1 & e_1 & 0 & 0 & \ldots & 0 & 0 \\ e_1 & d_2 & e_2 & 0 & \ldots & 0 & 0 \\ 0 & e_2 & d_3 & e_3 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & \ldots & \ldots & d_{N_{\rm step}-2} & e_{N_{\rm step}-1} \\ 0 & \ldots & \ldots & \ldots & \ldots & e_{N_{\rm step}-1} & d_{N_{\rm step}-1} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \ldots \\ \ldots \\ \ldots \\ u_{N_{\rm step}-1} \end{pmatrix} = 2E \begin{pmatrix} u_1 \\ u_2 \\ \ldots \\ \ldots \\ \ldots \\ u_{N_{\rm step}-1} \end{pmatrix} \tag{13.43}$$

or if we wish to be more detailed, we can write the tri-diagonal matrix as

$$\begin{pmatrix} \frac{2}{h^2} + V_1 & -\frac{1}{h^2} & 0 & 0 & \ldots & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2} + V_2 & -\frac{1}{h^2} & 0 & \ldots & 0 & 0 \\ 0 & -\frac{1}{h^2} & \frac{2}{h^2} + V_3 & -\frac{1}{h^2} & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & \ldots & \ldots & \frac{2}{h^2} + V_{N_{\rm step}-2} & -\frac{1}{h^2} \\ 0 & \ldots & \ldots & \ldots & \ldots & -\frac{1}{h^2} & \frac{2}{h^2} + V_{N_{\rm step}-1} \end{pmatrix} \tag{13.44}$$

This is a matrix problem with a tri-diagonal matrix of dimension $N_{\rm step}-1 \times N_{\rm step}-1$ and will thus yield $N_{\rm step}-1$ eigenvalues. It is important to notice that we do not set up a matrix of dimension $N_{\rm step} \times N_{\rm step}$ since we can fix the value of the wave function at $k = N_{\rm step}$. Similarly, we know the wave function at the other end point, that is for x_0.

The above equation represents an alternative to the numerical solution of the differential equation for the SE.

The eigenvalues of the harmonic oscillator in one dimension are well known. In our case, with all constants set equal to 1, we have

$$E_n = n + \frac{1}{2}, \tag{13.45}$$

with the ground state being $E_0 = 1/2$. Note however that we have rewritten the SE 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 $1, 3, 5, 7 \ldots \ldots$.

In the next subsection we will try to delineate how to solve the above equation. A program listing is also included.

Numerical solution of the SL by diagonalization

The algorithm for solving Eq. (13.43) may take the following form

- Define values for $N_{\rm step}$, $R_{\rm min}$ and $R_{\rm max}$. These values define in turn the step size h. Typical values for $R_{\rm max}$ and $R_{\rm min}$ could be 10 and -10 respectively for the lowest-lying states. The number of mesh points $N_{\rm step}$ could be in the range 100 to some thousands. You can check the stability of the results as functions of $N_{\rm step}-1$ and $R_{\rm max}$ and $R_{\rm min}$ against the exact solutions.

- Construct then two one-dimensional arrays which contain all values of x_k and the potential V_k. For the latter it can be convenient to write a small function which sets up the potential as function of x_k. For the three-dimensional case you may also need to include the centrifugal potential. The dimension of these two arrays should go from 0 up to $N_{\rm step}$.

- Construct thereafter the one-dimensional vectors d and e, where d stands for the diagonal matrix elements and e the non-diagonal ones. Note that the dimension of these two arrays runs from 1 up to $N_{\rm step}-1$, since we know the wave function u at both ends of the chosen grid.

- We are now ready to obtain the eigenvalues by calling the function *tqli* which can be found on the web page of the course. Calling *tqli*, you have to transfer the matrices d and e, their dimension $n = N_{\rm step}-1$ and a matrix z of dimension $N_{\rm step}-1 \times N_{\rm step}-1$ which returns the eigenfunctions. On return, the array d contains the eigenvalues. If z is given as the unity matrix on input, it returns the eigenvectors. For a given eigenvalue k, the eigenvector is given by the column k in z, that is z[][k] in C, or z(:,k) in Fortran 90.

- 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

$$\int_{R_{\min}}^{R_{\max}} |u(x)|^2 \, dx \rightarrow h \sum_{i=0}^{N_{\rm step}} u_i^2 = 1,$$

  and we have used the trapezoidal rule for integration discussed in chapter 4.

Program example and results for the one-dimensional harmonic oscillator

We present here a program example which encodes the above algorithm.

```
/*
  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];     // tri-diagonal 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 13.1: 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.

13.4 Physics projects: Bound states in momentum space

In this problem we will solve the Schrödinger equation (SE) in momentum space for the deuteron. The deuteron has only one bound state at an energy of -2.223 MeV. The ground state is given by the quantum numbers $l = 0$, $S = 1$ and $J = 1$, with l, S, and J the relative orbital momentum, the total spin and the total angular momentum, respectively. These quantum numbers are the sum of the single-particle quantum numbers. The deuteron consists of a proton and neutron, with mass (average) of 938 MeV. The electron is not included in the solution of the SE since its mass is much smaller than those of the proton and the neutron. We can neglect it here. This means that e.g., the total spin S is the sum of the spin of the neutron and the proton. The above three quantum numbers can be summarized in the spectroscopic notation $^{2S+1}l_J = ^3S_1$, where S represents $l = 0$ here. It is a spin triplet state. The spin wave function is thus symmetric. This also applies to the spatial part, since $l = 0$. To obtain a totally anti-symmetric wave function we need to introduce another quantum number, namely isospin. The deuteron has isospin $T = 0$, which gives a final wave function which is anti-symmetric.

We are going to use a simplified model for the interaction between the neutron and the proton. We will assume that it goes like

$$V(r) = V_0 \frac{\exp(-\mu r)}{r}, \tag{13.46}$$

where μ has units m^{-1} and serves to screen the potential for large values of r. The variable r is the distance between the proton and the neutron. It is the relative coordinate, the centre of mass is not needed in this problem. The nucleon-nucleon interaction has a finite and small range, typically of some few fm[1]. We will in this exercise set $\mu = 0.7$ fm^{-1}. It is then proportional to the mass of the pion. The pion is the lightest meson, and sets therefore the range of the nucleon-nucleon interaction. For low-energy problems we can describe the nucleon-nucleon interaction through meson-exchange models, and the pion is the lightest known meson, with mass of approximately 138 MeV.

Since we are going to solve the SE in momentum, we need the Fourier transform of $V(r)$. In a partial wave basis for $l = 0$ it becomes

$$V(k',k) = \frac{V_0}{4k'k} ln\left(\frac{(k'+k)^2+\mu^2}{(k'-k)^2+\mu^2}\right), \tag{13.47}$$

where k' and k are the relative momenta for the proton and neutron system.

For relative coordinates, the SE in momentum space becomes

$$\frac{k^2}{m}\psi(k) + \frac{2}{\pi}\int_0^\infty dpp^2 V(k,p)\psi(p) = E\psi(k). \tag{13.48}$$

Here we have used units $\hbar = c = 1$. This means that k has dimension energy. This is the equation we are going to solve, with eigenvalue E and eigenfunction $\psi(k)$. The approach to solve this equations goes then as follows.

[1] 1 fm = 10^{-15} m.

First we need to evaluate the integral over p using e.g., gaussian quadrature. This means that we rewrite an integral like

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

where we have fixed N lattice points through the corresponding weights ω_i and points x_i. The integral in Eq. (13.48) is rewritten as

$$\frac{2}{\pi}\int_0^\infty dpp^2V(k,p)\psi(p) \approx \frac{2}{\pi}\sum_{i=1}^{N} \omega_i p_i^2 V(k,p_i)\psi(p_i). \tag{13.49}$$

We can then rewrite the SE as

$$\frac{k^2}{m}\psi(k) + \frac{2}{\pi}\sum_{j=1}^{N} \omega_j p_j^2 V(k,p_j)\psi(p_j) = E\psi(k). \tag{13.50}$$

Using the same mesh points for k as we did for p in the integral evaluation, we get

$$\frac{p_i^2}{m}\psi(p_i) + \frac{2}{\pi}\sum_{j=1}^{N} \omega_j p_j^2 V(p_i,p_j)\psi(p_j) = E\psi(p_i), \tag{13.51}$$

with $i,j = 1,2,\ldots,N$. This is a matrix eigenvalue equation and if we define an $N \times N$ matrix $\mathbf{H}$ to be

$$H_{ij} = \frac{p_i^2}{m}\delta_{ij} + \frac{2}{\pi}\omega_j p_j^2 V(p_i,p_j), \tag{13.52}$$

where δ_{ij} is the Kronecker delta, and an $N \times 1$ vector

$$\Psi = \begin{pmatrix} \psi(p_1) \\ \psi(p_2) \\ \ldots \\ \ldots \\ \psi(p_N) \end{pmatrix}, \tag{13.53}$$

we have the eigenvalue problem

$$\mathbf{H}\Psi = E\Psi. \tag{13.54}$$

The algorithm for solving the last equation may take the following form

- Fix the number of mesh points N.
- Use the function $gauleg$ in the program library to set up the weights ω_i and the points p_i. Before you go on you need to recall that $gauleg$ uses the Legendre polynomials to fix the mesh points and weights. This means that the integral is for the interval [-1,1]. Your integral is for the interval [0,∞]. You will need to map the weights from $gauleg$ to your interval. To do this, call first $gauleg$, with $a = -1$, $b = 1$. It returns the mesh points and

weights. You then map these points over to the limits in your integral. You can then use the following mapping

$$p_i = \text{const} \times tan\left\{\frac{\pi}{4}(1+x_i)\right\},$$

and

$$\omega_i = \text{const}\frac{\pi}{4}\frac{w_i}{cos^2\left(\frac{\pi}{4}(1+x_i)\right)}.$$

$const$ is a constant which we discuss below.

- Construct thereafter the matrix $\mathbf{H}$ with

$$V(p_i,p_j) = \frac{V_0}{4p_ip_j}ln\left(\frac{(p_j+p_i)^2+\mu^2}{(p_j-p_i)^2+\mu^2}\right).$$

- We are now ready to obtain the eigenvalues. We need first to rewrite the matrix $\mathbf{H}$ in tri-diagonal form. Do this by calling the library function *tred2*. This function returns the vector d with the diagonal matrix elements of the tri-diagonal matrix while e are the non-diagonal ones. To obtain the eigenvalues we call the function $tqli$. On return, the array d contains the eigenvalues. If z is given as the unity matrix on input, it returns the eigenvectors. For a given eigenvalue k, the eigenvector is given by the column k in z, that is z[][k] in C, or z(:,k) in Fortran 90.

The problem to solve

1. Before you write the main program for the above algorithm make a dimensional analysis of Eq. (13.48)! You can choose units so that p_i and ω_i are in fm-1. This is the standard unit for the wave vector. Recall then to insert $\hbar c$ in the appropriate places. For this case you can set the value of $const = 1$. You could also choose units so that the units of p_i and ω_i are in MeV. (we have previously used so-called natural units $\hbar = c = 1$). You will then need to multiply μ with $\hbar c = 197$ MeVfm to obtain the same units in the expression for the potential. Why? Show that $V(p_i,p_j)$ must have units MeV^{-2}. What is the unit of V_0? If you choose these units you should also multiply the mesh points and the weights with $\hbar c = 197$. That means, set the constant $const = 197$.

2. Write your own program so that you can solve the SE in momentum space.

3. Adjust the value of V_0 so that you get close to the experimental value of the binding energy of the deuteron, -2.223 MeV. Which sign should V_0 have?

4. Try increasing the number of mesh points in steps of 8, for example 16, 24, etc and see how the energy changes. Your program returns equally many eigenvalues as mesh points N. Only the true ground state will be at negative energy.

13.5 Physics projects: Quantum mechanical scattering

We are now going to solve the SE for the neutron-proton system in momentum space for positive energies E in order to obtain the phase shifts. In the previous physics project on bound states in momentum space, we obtained the SE in momentum space by Eq. (13.48). k was the relative momentum between the two particles. A partial wave expansion was used in order to reduce the problem to an integral over the magnitude of momentum only. The subscript l referred therefore to a partial wave with a given orbital momentum l. To obtain the potential in momentum space we used the Fourier-Bessel transform (Hankel transform)

$$V_l(k,k') = \int j_l(kr)V(r)j_l(k'r)r^2dr, \tag{13.55}$$

where j_l is the spherical Bessel function. We will just study the case $l = 0$, which means that $j_0(kr) = sin(kr)/kr$.

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

$$R_l(k,k') = V_l(k,k') + \frac{2}{\pi}\mathcal{P}\int_0^\infty dqq^2V_l(k,q)\frac{1}{E-q^2/m}R_l(q,k'), \tag{13.56}$$

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

$$E = \frac{k_0^2}{m}. \tag{13.57}$$

The symbol $\mathcal{P}$ 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. Eq. (13.56) represents then the problem you will have to solve numerically.

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

$$R_l(k_0,k_0) = -\frac{tan\delta_l}{mk_0}. \tag{13.58}$$

The principal value in Eq. (13.56) 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 introduce first the calculus relation

$$\int_{-\infty}^{\infty}\frac{dk}{k-k_0} = 0. \tag{13.59}$$

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

$$\int_0^{\infty}\frac{dk}{k^2-k_0^2} = 0. \tag{13.60}$$

We can use this to express a principal values integral as

$$\mathcal{P}\int_0^\infty \frac{f(k)dk}{k^2-k_0^2} = \int_0^\infty \frac{(f(k)-f(k_0))dk}{k^2-k_0^2}, \tag{13.61}$$

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. (13.61) to rewrite Eq. (13.56) as

$$R(k,k') = V(k,k') + \frac{2}{\pi}\int_0^\infty dq \frac{q^2V(k,q)R(q,k') - k_0^2V(k,k_0)R(k_0,k')}{(k_0^2-q^2)/m}. \tag{13.62}$$

Using the mesh points k_j and the weights ω_j, we can rewrite Eq. (13.62) as

$$R(k,k') = V(k,k') + \frac{2}{\pi}\sum_{j=1}^N \frac{\omega_j k_j^2 V(k,k_j)R(k_j,k')}{(k_0^2-k_j^2)/m} - \frac{2}{\pi}k_0^2V(k,k_0)R(k_0,k')\sum_{n=1}^N \frac{\omega_n}{(k_0^2-k_n^2)/m}. \tag{13.63}$$

This equation contains now the unknowns $R(k_i,k_j)$ (with dimension $N\times N$) and $R(k_0,k_0)$. We can turn Eq. (13.63) into an equation with dimension $(N+1)\times(N+1)$ with a mesh 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 $k_{N+1} = k_0$. With these new mesh points we define the matrix

$$A_{i,j} = \delta_{i,j} + V(k_i,k_j)u_j, \tag{13.64}$$

where δ is the Kronecker δ and

$$u_j = \frac{2}{\pi}\frac{\omega_j k_j^2}{(k_0^2-k_j^2)/m} \qquad j = 1,N \tag{13.65}$$

and

$$u_{N+1} = -\frac{2}{\pi}\sum_{j=1}^N \frac{k_0^2\omega_j}{(k_0^2-k_j^2)/m}. \tag{13.66}$$

With the matrix A we can rewrite Eq. (13.63) as a matrix problem of dimension $(N+1)\times(N+1)$. All matrices R, A and V have this dimension and we get

$$A_{i,l}R_{l,j} = V_{i,j}, \tag{13.67}$$

or just

$$AR = V. \tag{13.68}$$

Since we already have defined A and V (these are stored as $(N+1)\times(N+1)$ matrices) Eq. (13.68) involves only the unknown R. We obtain it by matrix inversion, i.e.,

$$R = A^{-1}V. \tag{13.69}$$

Thus, to obtain R, we need to set up the matrices A and V and invert the matrix A. To do that one can use the function *matinv* in the program library. With the inverse A^{-1}, performing a matrix multiplication with V results in R.

With R we obtain subsequently the phaseshifts using the relation

$$R(k_{N+1}, k_{N+1}) = R(k_0, k_0) = -\frac{tan\delta}{mk_0}. \tag{13.70}$$

Chapter 14

Differential equations

14.1 Introduction

Historically, differential equations have originated in chemistry, physics and engineering. More recently they have also been used widely in medicine, biology etc. In this chapter we restrict the attention to ordinary differential equations. We focus on initial value and boundary value problems and present some of the more commonly used methods for solving such problems numerically.

The physical systems which are discussed range from a simple cooling problem to the physics of a neutron star.

14.2 Ordinary differential equations (ODE)

In this section we will mainly deal with ordinary differential equations and numerical methods suitable for dealing with them. However, before we proceed, a brief remainder on differential equations may be appropriate.

- The order of the ODE refers to the order of the derivative on the left-hand side in the equation

$$\frac{dy}{dt} = f(t, y). \tag{14.1}$$

This equation is of first order and f is an arbitrary function. A second-order equation goes typically like

$$\frac{d^2y}{dt^2} = f(t, \frac{dy}{dt}, y). \tag{14.2}$$

A well-known second-order equation is Newton's second law

$$m\frac{d^2x}{dt^2} = -kx, \tag{14.3}$$

where k is the force constant. ODE depend only on one variable, whereas

- partial differential equations like the time-dependent Schrödinger equation

$$i\hbar\frac{\partial\psi(\mathbf{x},t)}{\partial t} = \frac{\hbar^2}{2m}\left(\frac{\partial^2\psi(\mathbf{r},t)}{\partial x^2} + \frac{\partial^2\psi(\mathbf{r},t)}{\partial y^2} + \frac{\partial^2\psi(\mathbf{r},t)}{\partial z^2}\right) + V(\mathbf{x})\psi(\mathbf{x},t), \tag{14.4}$$

  may depend on several variables. In certain cases, like the above equation, the wave function can be factorized in functions of the separate variables, so that the Schrödinger equation can be rewritten in terms of sets of ordinary differential equations.

- We distinguish also between linear and non-linear differential equation where e.g.,

$$\frac{dy}{dt} = g^3(t)y(t), \tag{14.5}$$

  is an example of a linear equation, while

$$\frac{dy}{dt} = g^3(t)y(t) - g(t)y^2(t), \tag{14.6}$$

  is a non-linear ODE. Another concept which dictates the numerical method chosen for solving an ODE, is that of initial and boundary conditions. To give an example, in our study of neutron stars below, we will need to solve two coupled first-order differential equations, one for the total mass m and one for the pressure P as functions of ρ

$$\frac{dm}{dr} = 4\pi r^2\rho(r)/c^2,$$

  and

$$\frac{dP}{dr} = -\frac{Gm(r)}{r^2}\rho(r)/c^2.$$

  where ρ is the mass-energy density. The initial conditions are dictated by the mass being zero at the center of the star, i.e., when $r = 0$, yielding $m(r = 0) = 0$. The other condition is that the pressure vanishes at the surface of the star. This means that at the point where we have $P = 0$ in the solution of the integral equations, we have the total radius R of the star and the total mass $m(r = R)$. These two conditions dictate the solution of the equations. Since the differential equations are solved by stepping the radius from $r = 0$ to $r = R$, so-called one-step methods (see the next section) or Runge-Kutta methods may yield stable solutions.

  In the solution of the Schrödinger equation for a particle in a potential, we may need to apply boundary conditions as well, such as demanding continuity of the wave function and its derivative.

- In many cases it is possible to rewrite a second-order differential equation in terms of two first-order differential equations. Consider again the case of Newton's second law in Eq. (14.3). If we define the position $x(t) = y^{(1)}(t)$ and the velocity $v(t) = y^{(2)}(t)$ as its derivative

$$\frac{dy^{(1)}(t)}{dt} = \frac{dx(t)}{dt} = y^{(2)}(t), \tag{14.7}$$

we can rewrite Newton's second law as two coupled first-order differential equations

$$m\frac{dy^{(2)}(t)}{dt} = -kx(t) = -ky^{(1)}(t), \tag{14.8}$$

and

$$\frac{dy^{(1)}(t)}{dt} = y^{(2)}(t). \tag{14.9}$$

14.3 Finite difference methods

These methods fall under the general class of one-step methods. The algoritm is rather simple. Suppose we have an initial value for the function $y(t)$ given by

$$y_0 = y(t = t_0). \tag{14.10}$$

We are interested in solving a differential equation in a region in space [a,b]. We define a step h by splitting the interval in N sub intervals, so that we have

$$h = \frac{b-a}{N}. \tag{14.11}$$

With this step and the derivative of y we can construct the next value of the function y at

$$y_1 = y(t_1 = t_0 + h), \tag{14.12}$$

and so forth. If the function is rather well-behaved in the domain [a,b], we can use a fixed step size. If not, adaptive steps may be needed. Here we concentrate on fixed-step methods only. Let us try to generalize the above procedure by writing the step y_{i+1} in terms of the previous step y_i

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h\Delta(t_i, y_i(t_i)) + O(h^{p+1}), \tag{14.13}$$

where $O(h^{p+1})$ represents the truncation error. To determine Δ, we Taylor expand our function y

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}), \tag{14.14}$$

where we will associate the derivatives in the parenthesis with

$$\Delta(t_i, y_i(t_i)) = (y'(t_i) + \cdots + y^{(p)}(t_i)\frac{h^{p-1}}{p!}). \tag{14.15}$$

We define

$$y'(t_i) = f(t_i, y_i) \tag{14.16}$$

and if we truncate Δ at the first derivative, we have

$$y_{i+1} = y(t_i) + hf(t_i, y_i) + O(h^2), \tag{14.17}$$

which when complemented with $t_{i+1} = t_i + h$ forms the algorithm for the well-known Euler method. Note that at every step we make an approximation error of the order of $O(h^2)$, however the total error is the sum over all steps $N = (b-a)/h$, yielding thus a global error which goes like $NO(h^2) \approx O(h)$. To make Euler's method more precise we can obviously decrease h (increase N). However, if we are computing the derivative f numerically by e.g., the two-steps formula

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

we can enter into roundoff error problems when we subtract two almost equal numbers $f(x + h) - f(x) \approx 0$. Euler's method is not recommended for precision calculation, although it is handy to use in order to get a first view on how a solution may look like. As an example, consider Newton's equation rewritten in Eqs. (14.8) and (14.9). We define $y_0 = y^{(1)}(t = 0)$ an $v_0 = y^{(2)}(t = 0)$. The first steps in Newton's equations are then

$$y_1^{(1)} = y_0 + hv_0 + O(h^2) \tag{14.18}$$

and

$$y_1^{(2)} = v_0 - hy_0k/m + O(h^2). \tag{14.19}$$

The Euler method is asymmetric in time, since it uses information about the derivative at the beginning of the time interval. This means that we evaluate the position at $y_1^{(1)}$ using the velocity at $y_0^{(2)} = v_0$. A simple variation is to determine $y_{n+1}^{(1)}$ using the velocity at $y_{n+1}^{(2)}$, that is (in a slightly more generalized form)

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1}^{(2)} + O(h^2) \tag{14.20}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2). \tag{14.21}$$

The acceleration a_n is a function of $a_n(y_n^{(1)}, y_n^{(2)}, t)$ and needs to be evaluated as well. This is the Euler-Cromer method.

Let us then include the second derivative in our Taylor expansion. We have then

$$\Delta(t_i, y_i(t_i)) = f(t_i) + \frac{h}{2}\frac{df(t_i, y_i)}{dt} + O(h^3). \tag{14.22}$$

The second derivative can be rewritten as

$$y'' = f' = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}f \tag{14.23}$$

and we can rewrite Eq. (14.14) as

$$y_{i+1} = y(t = t_i + h) = y(t_i) + hf(t_i) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial y}f\right) + O(h^3), \tag{14.24}$$

which has a local approximation error $O(h^3)$ and a global error $O(h^2)$. These approximations can be generalized by using the derivative f to arbitrary order so that we have

$$y_{i+1} = y(t = t_i + h) = y(t_i) + h(f(t_i, y_i) + \ldots f^{(p-1)}(t_i, y_i)\frac{h^{p-1}}{p!}) + O(h^{p+1}). \tag{14.25}$$

These methods, based on higher-order derivatives, are in general not used in numerical computation, since they rely on evaluating derivatives several times. Unless one has analytical expressions for these, the risk of roundoff errors is large.

14.3.1 Improvements to Euler's algorithm, higher-order methods

The most obvious improvements to Euler's and Euler-Cromer's algorithms, avoiding in addition the need for computing a second derivative, is the so-called midpoint method. We have then

$$y_{n+1}^{(1)} = y_n^{(1)} + \frac{h}{2}\left(y_{n+1}^{(2)} + y_n^{(2)}\right) + O(h^2) \tag{14.26}$$

and

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_n + O(h^2), \tag{14.27}$$

yielding

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_n^{(2)} + \frac{h^2}{2}a_n + O(h^3) \tag{14.28}$$

implying that the local truncation error in the position is now $O(h^3)$, whereas Euler's or Euler-Cromer's methods have a local error of $O(h^2)$. Thus, the midpoint method yields a global error with second-order accuracy for the position and first-order accuracy for the velocity. However, although these methods yield exact results for constant accelerations, the error increases in general with each time step.

One method that avoids this is the so-called half-step method. Here we define

$$y_{n+1/2}^{(2)} = y_{n-1/2}^{(2)} + ha_n + O(h^2), \tag{14.29}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{14.30}$$

Note that this method needs the calculation of $y_{1/2}^{(2)}$. This is done using e.g., Euler's method

$$y_{1/2}^{(2)} = y_0^{(2)} + ha_0 + O(h^2). \tag{14.31}$$

As this method is numerically stable, it is often used instead of Euler's method. Another method which one may encounter is the Euler-Richardson method with

$$y_{n+1}^{(2)} = y_n^{(2)} + ha_{n+1/2} + O(h^2), \tag{14.32}$$

and

$$y_{n+1}^{(1)} = y_n^{(1)} + hy_{n+1/2}^{(2)} + O(h^2). \tag{14.33}$$

14.4 More on finite difference methods, Runge-Kutta methods

Runge-Kutta (RK) methods are based on Taylor expansion formulae, but yield in general better algorithms for solutions of an ODE. The basic philosophy is that it provides an intermediate step in the computation of y_{i+1}.

To see this, consider first the following definitions

$$\frac{dy}{dt} = f(t, y), \tag{14.34}$$

and

$$y(t) = \int f(t, y)dt, \tag{14.35}$$

and

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt. \tag{14.36}$$

To demonstrate the philosophy behind RK methods, let us consider the second-order RK method, RK2. The first approximation consists in Taylor expanding $f(t, y)$ around the center of the integration interval t_i to t_{i+1}, i.e., at $t_i + h/2$, h being the step. Using the midpoint formula for an integral, defining $y(t_i + h/2) = y_{i+1/2}$ and $t_i + h/2 = t_{i+1/2}$, we obtain

$$\int_{t_i}^{t_{i+1}} f(t, y)dt \approx hf(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{14.37}$$

This means in turn that we have

$$y_{i+1} = y_i + hf(t_{i+1/2}, y_{i+1/2}) + O(h^3). \tag{14.38}$$

However, we do not know the value of $y_{i+1/2}$. Here comes thus the next approximation, namely, we use Euler's method to approximate $y_{i+1/2}$. We have then

$$y_{(i+1/2)} = y_i + \frac{h}{2}\frac{dy}{dt} = y(t_i) + \frac{h}{2}f(t_i, y_i). \tag{14.39}$$

This means that we can define the following algorithm for the second-order Runge-Kutta method, RK2.

$$k_1 = hf(t_i, y_i), \tag{14.40}$$

$$k_2 = hf(t_{i+1/2}, y_i + k_1/2), \tag{14.41}$$

with the final value

$$y_{i+i} \approx y_i + k_2 + O(h^3). \tag{14.42}$$

The difference between the previous one-step methods is that we now need an intermediate step in our evaluation, namely $t_i + h/2 = t_{(i+1/2)}$ where we evaluate the derivative f. This

involves more operations, but the gain is a better stability in the solution. The fourth-order Runge-Kutta, RK4, which we will employ in the solution of various differential equations below, has the following algorithm

$$k_1 = hf(t_i, y_i), \tag{14.43}$$

$$k_2 = hf(t_i + h/2, y_i + k_1/2), \tag{14.44}$$

$$k_3 = hf(t_i + h/2, y_i + k_2/2) \tag{14.45}$$

$$k_4 = hf(t_i + h, y_i + k_3) \tag{14.46}$$

with the final value

$$y_{i+1} = y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \tag{14.47}$$

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.

14.5 Adaptive Runge-Kutta and multistep methods

in preparation

14.6 Physics examples

14.6.1 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. 14.1. If the spring is not compressed or stretched too far, the force on the block at a given position x is

$$F = -kx. \tag{14.48}$$

The negative sign means that the force acts to restore the object to an equilibrium position. Newton's equation of motion for this idealized system is then

$$m\frac{d^2x}{dt^2} = -kx, \tag{14.49}$$

or we could rephrase it as

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x = -\omega_0^2 x, \tag{14.50}$$

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

$$x(t) = Acos(\omega_0 t + \nu),$$

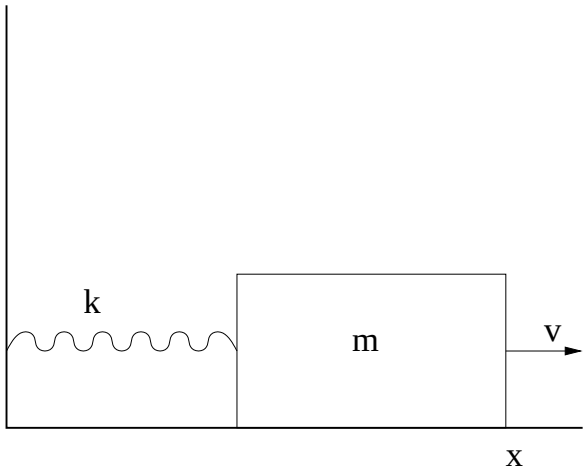


Figure 14.1: Block tied to a wall with a spring tension acting on it.

where A is the amplitude and ν the phase constant. This provides in turn an important test for the numerical solution and the development of a program for more complicated cases which cannot be solved analytically.

As mentioned earlier, in certain cases it is possible to rewrite a second-order differential equation as two coupled first-order differential equations. With the position $x(t)$ and the velocity $v(t) = dx/dt$ we can reformulate Newton's equation in the following way

$$\frac{dx(t)}{dt} = v(t), \tag{14.51}$$

and

$$\frac{dv(t)}{dt} = -\omega_0^2 x(t). \tag{14.52}$$

We are now going to solve these equations using the Runge-Kutta method to fourth order discussed previously. Before proceeding however, it is important to note that in addition to the exact solution, we have at least two further tests which can be used to check our solution.

Since functions like cos are periodic with a period 2π, then the solution $x(t)$ has also to be periodic. This means that

$$x(t+T) = x(t), \tag{14.53}$$

with T the period defined as

$$T = \frac{2\pi}{\omega_0} = \frac{2\pi}{\sqrt{k/m}}. \tag{14.54}$$

Observe that T depends only on k/m and not on the amplitude of the solution or the constant ν.

In addition to the periodicity test, the total energy has also to be conserved.

Suppose we choose the initial conditions

$$x(t=0) = 1\ \mathrm{m} \qquad v(t=0) = 0\ \mathrm{m/s}, \tag{14.55}$$

meaning that block is at rest at $t = 0$ but with a potential energy

$$E_0 = \frac{1}{2}kx(t = 0)^2 = \frac{1}{2}k. \tag{14.56}$$

The total energy at any time t has however to be conserved, meaning that our solution has to fulfil the condition

$$E_0 = \frac{1}{2}kx(t)^2 + \frac{1}{2}mv(t)^2. \tag{14.57}$$

An algorithm which implements these equations is included below.

1. Choose the initial position and speed, with the most common choice $v(t = 0) = 0$ and some fixed value for the position. Since we are going to test our results against the periodicity requirement, it is convenient to set the final time equal $t_f = 2\pi$, where we choose $k/m = 1$. The initial time is set equal to $t_i = 0$. You could alternatively read in the ratio k/m.

2. Choose the method you wish to employ in solving the problem. In the enclosed program we have chosen the fourth-order Runge-Kutta method. Subdivide the time interval $[t_i, t_f]$ into a grid with step size
$$h = \frac{t_f - t_i}{N},$$
where N is the number of mesh points.

3. Calculate now the total energy given by
$$E_0 = \frac{1}{2}kx(t = 0)^2 = \frac{1}{2}k.$$
and use this when checking the numerically calculated energy from the Runge-Kutta iterations.

4. The Runge-Kutta method is used to obtain x_{i+1} and v_{i+1} starting from the previous values x_i and v_i..

5. When we have computed $x(v)_{i+1}$ we upgrade $t_{i+1} = t_i + h$.

6. This iterative process continues till we reach the maximum time $t_f = 2\pi$.

7. The results are checked against the exact solution. Furthermore, one has to check the stability of the numerical solution against the chosen number of mesh points N.

Program to solve the differential equations for a sliding block

The program which implements the above algorithm is presented here.

```
/*     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. 14.2 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.

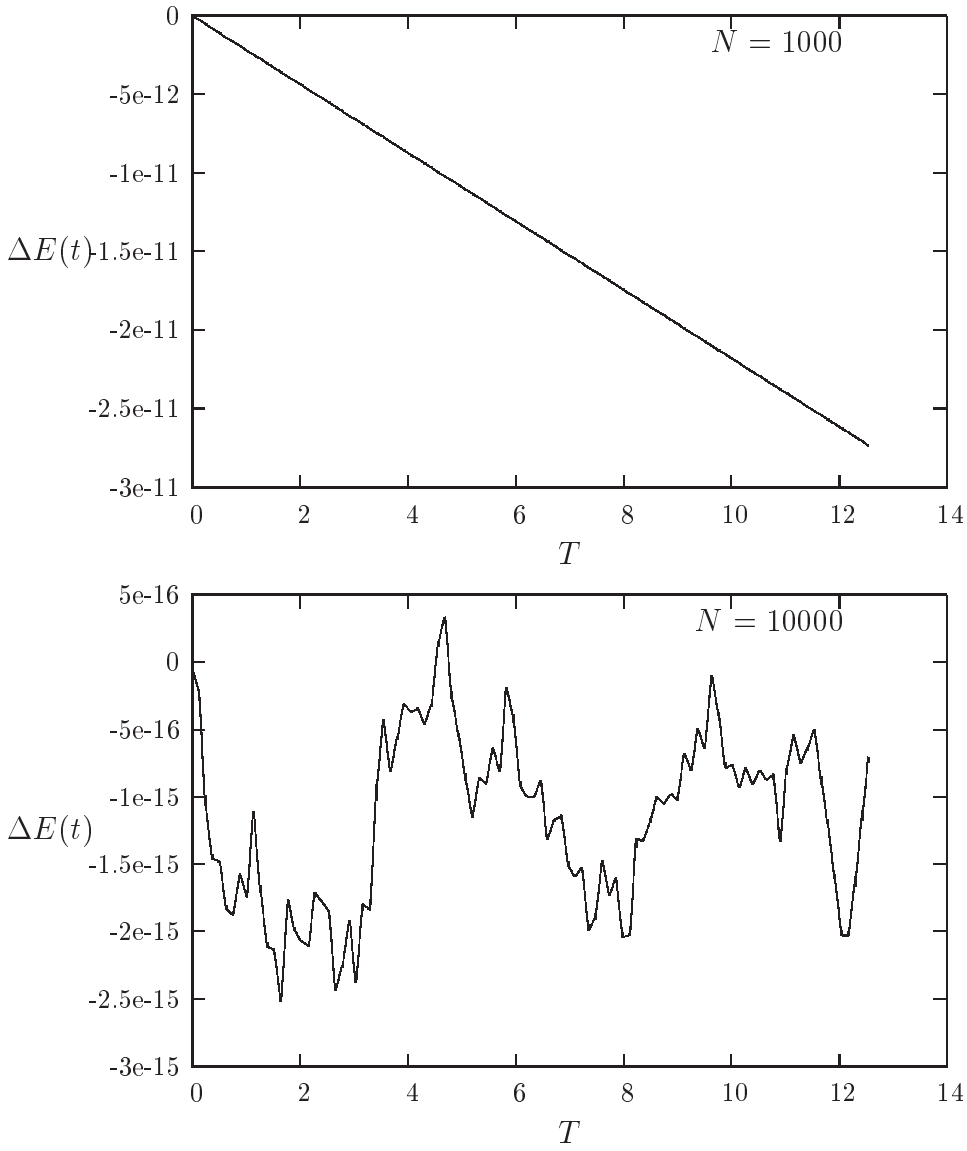


Figure 14.2: Plot of $\Delta E(t) = E_0 - E_{\text{computed}}$ for $N = 1000$ and $N = 10000$ time steps up to two periods. The initial position $x_0 = 1$ m and initial velocity $v_0 = 0$ m/s. The mass and spring tension are set to $k = m = 1$.

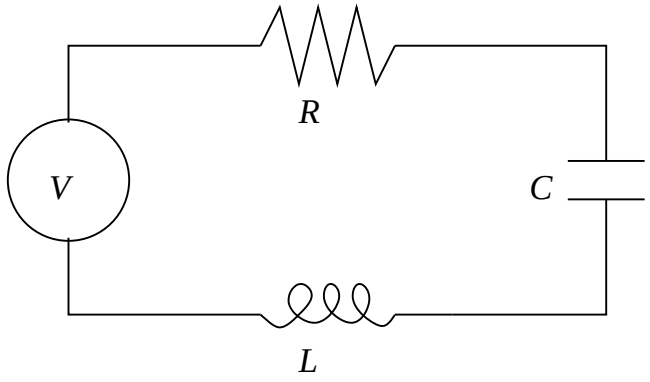


Figure 14.3: Simple RLC circuit with a voltage source V.

14.6.2 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. (14.50) to

$$\frac{d^2x}{dt^2} = -\omega_0^2 x - \nu\frac{dx}{dt}, \tag{14.58}$$

where ν is the damping coefficient, being a measure of the magnitude of the drag term.

We could however counteract the dissipative mechanism by applying e.g., a periodic external force

$$F(t) = Bcos(\omega t), \tag{14.59}$$

and we rewrite Eq. (14.58) as

$$\frac{d^2x}{dt^2} = -\omega_0^2 x - \nu\frac{dx}{dt} + F(t). \tag{14.60}$$

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. (14.60) as

$$L\frac{d^2Q}{dt^2} + \frac{Q}{C} + R\frac{dQ}{dt} = V(t). \tag{14.61}$$

The circuit is shown in Fig. 14.3.

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

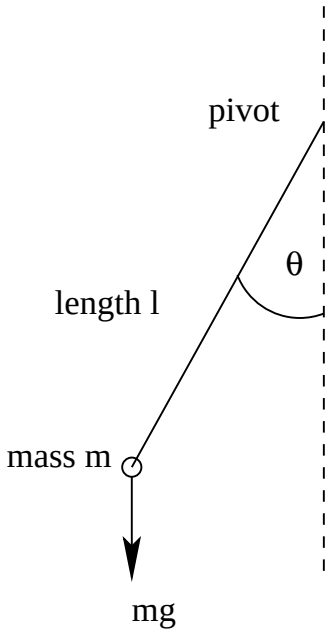


Figure 14.4: A simple pendulum.

drop LdI/dt. The circuit is powered by an alternating voltage source and using Kirchhoff's law, which is a consequence of energy conservation, we have

$$V(t) = IR + LdI/dt + Q/C, \tag{14.62}$$

and using

$$I = \frac{dQ}{dt}, \tag{14.63}$$

we arrive at Eq. (14.61).

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!

14.6.3 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. 14.4.

The angular equation of motion of the pendulum is again given by Newton's equation, but now as a nonlinear differential equation

$$ml\frac{d^2\theta}{dt^2} + mgsin(\theta) = 0, \tag{14.64}$$

with an angular velocity and acceleration given by

$$v = l\frac{d\theta}{dt}, \tag{14.65}$$

and

$$a = l\frac{d^2\theta}{dt^2}. \tag{14.66}$$

For small angles, we can use the approximation

$$sin(\theta) \approx \theta.$$

and rewrite the above differential equation as

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\theta, \tag{14.67}$$

which is exactly of the same form as Eq. (14.50). We can thus check our solutions for small values of θ against an analytical solution. The period is now

$$T = \frac{2\pi}{\sqrt{l/g}}. \tag{14.68}$$

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

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = 0, \tag{14.69}$$

where ν is now a positive constant parameterizing the viscosity of the medium in question. In order to maintain the motion against viscosity, it is necessary to add some external driving force. We choose here, in analogy with the discussion about the electric circuit, a periodic driving force. The last equation becomes then

$$ml\frac{d^2\theta}{dt^2} + \nu\frac{d\theta}{dt} + mgsin(\theta) = Acos(\omega t), \tag{14.70}$$

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

$$\omega_0 = \sqrt{g/l}, \tag{14.71}$$

the so-called natural frequency and the new dimensionless quantities

$$\hat{t} = \omega_0 t, \tag{14.72}$$

$$\hat{\omega} = \frac{\omega}{\omega_0}, \tag{14.73}$$

and introducing the quantity Q, called the *quality factor*,

$$Q = \frac{mg}{\omega_0 \nu}, \tag{14.74}$$

and the dimensionless amplitude

$$\hat{A} = \frac{A}{mg} \tag{14.75}$$

we can rewrite Eq. (14.70) as

$$\frac{d^2\theta}{d\hat{t}^2} + \frac{1}{Q}\frac{d\theta}{d\hat{t}} + sin(\theta) = \hat{A}cos(\hat{\omega}\hat{t}). \tag{14.76}$$

This equation can in turn be recast in terms of two coupled first-order differential equations as follows

$$\frac{d\theta}{d\hat{t}} = \hat{v}, \tag{14.77}$$

and

$$\frac{d\hat{v}}{d\hat{t}} = -\frac{\hat{v}}{Q} - sin(\theta) + \hat{A}cos(\hat{\omega}\hat{t}). \tag{14.78}$$

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.

14.6.4 Spinning magnet

Another simple example is that of e.g., a compass needle that is free to rotate in a periodically reversing magnetic field perpendicular to the axis of the needle. The equation is then

$$\frac{d^2\theta}{dt^2} = -\frac{\mu}{I}B_0 cos(\omega t)sin(\theta), \tag{14.79}$$

where θ is the angle of the needle with respect to a fixed axis along the field, μ is the magnetic moment of the needle, I its moment of inertia and B_0 and ω the amplitude and angular frequency of the magnetic field respectively.

14.7 Physics Project: the pendulum

14.7.1 Analytic results for the pendulum

Although the solution to the equations for the pendulum can only be obtained through numerical efforts, it is always useful to check our numerical code against analytic solutions. For small angles θ, we have $sin\theta \approx \theta$ and our equations become

$$\frac{d\theta}{d\hat{t}} = \hat{v}, \tag{14.80}$$

and

$$\frac{d\hat{v}}{d\hat{t}} = -\frac{\hat{v}}{Q} - \theta + \hat{A}cos(\hat{\omega}\hat{t}). \tag{14.81}$$

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

$$\begin{aligned}\theta(t) &= \left[\theta_0 - \frac{\hat{A}(1-\hat{\omega}^2)}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}\right] e^{-\tau/2Q}cos(\sqrt{1-\frac{1}{4Q^2}}\tau) \\ &+ \left[\hat{v}_0 + \frac{\theta_0}{2Q} - \frac{\hat{A}(1-3\hat{\omega}^2)/2Q}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}\right] e^{-\tau/2Q}sin(\sqrt{1-\frac{1}{4Q^2}}\tau) + \frac{\hat{A}(1-\hat{\omega}^2)cos(\hat{\omega}\tau)+\frac{\hat{\omega}}{Q}sin(\hat{\omega}\tau)}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2},\end{aligned} \tag{14.82}$$

and

$$\begin{aligned}\hat{v}(t) &= \left[\hat{v}_0 - \frac{\hat{A}\hat{\omega}^2/Q}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}\right] e^{-\tau/2Q}cos(\sqrt{1-\frac{1}{4Q^2}}\tau) \\ &- \left[\theta_0 + \frac{\hat{v}_0}{2Q} - \frac{\hat{A}[(1-\hat{\omega}^2)-\hat{\omega}^2/Q^2]}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}\right] e^{-\tau/2Q}sin(\sqrt{1-\frac{1}{4Q^2}}\tau) + \frac{\hat{\omega}\hat{A}[-(1-\hat{\omega}^2)sin(\hat{\omega}\tau)+\frac{\hat{\omega}}{Q}cos(\hat{\omega}\tau)]}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2},\end{aligned} \tag{14.83}$$

with $Q > 1/2$. The first two terms depend on the initial conditions and decay exponentially in time. If we wait long enough for these terms to vanish, the solutions become independent of the initial conditions and the motion of the pendulum settles down to the following simple orbit in phase space

$$\theta(t) = \frac{\hat{A}(1-\hat{\omega}^2)cos(\hat{\omega}\tau) + \frac{\hat{\omega}}{Q}sin(\hat{\omega}\tau)}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}, \tag{14.84}$$

and

$$\hat{v}(t) = \frac{\hat{\omega}\hat{A}[-(1-\hat{\omega}^2)sin(\hat{\omega}\tau) + \frac{\hat{\omega}}{Q}cos(\hat{\omega}\tau)]}{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}, \tag{14.85}$$

tracing the closed phase-space curve

$$\left(\frac{\theta}{\tilde{A}}\right)^2 + \left(\frac{\hat{v}}{\hat{\omega}\tilde{A}}\right)^2 = 1 \tag{14.86}$$

with

$$\tilde{A} = \frac{\hat{A}}{\sqrt{(1-\hat{\omega}^2)^2+\hat{\omega}^2/Q^2}}. \tag{14.87}$$

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. 14.5 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. (14.17), Euler-Richardson's method in Eqs. (14.32)-(14.33) and finally the fourth-order Runge-Kutta scheme RK4. We note that after few time steps, we obtain the classical harmonic

motion. We would have obtained a similar picture if we were to switch off the external force, $\hat{A} = 0$ and set the frictional damping to zero, i.e., $Q = 0$. Then, the qualitative picture is that of an idealized harmonic oscillation without damping. However, we see that Euler's method performs poorly and after a few steps its algorithmic simplicity leads to results which deviate considerably from the other methods. In the discussion hereafter we will thus limit ourselves to

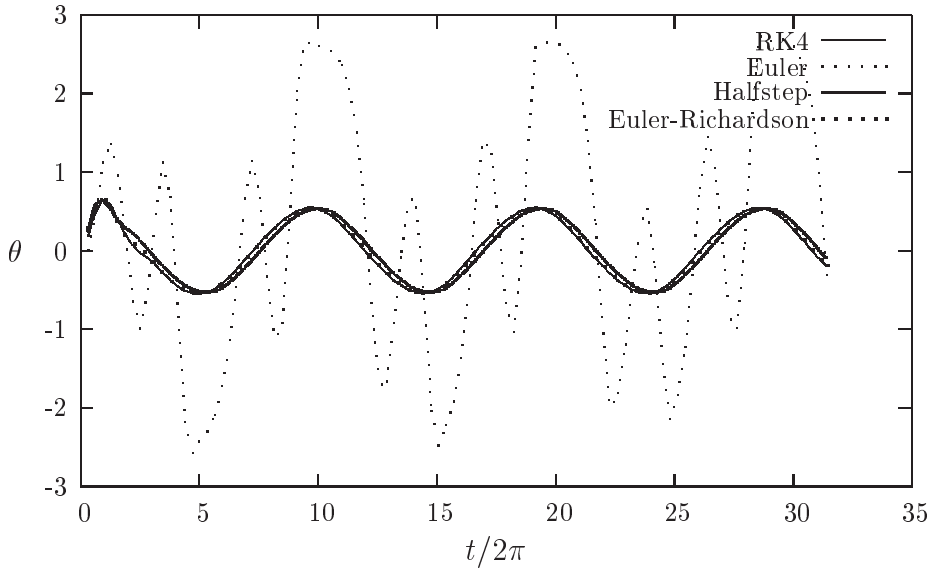


Figure 14.5: Plot of θ as function of time τ with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 0.5$. The mass and length of the pendulum are set equal to 1. The initial velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.01$. Four different methods have been used to solve the equations, Euler's method from Eq. (14.17), the half-step method, Euler-Richardson's method in Eqs. (14.32)-(14.33) and finally the fourth-order Runge-Kutta scheme RK4. Only $N = 100$ integration points have been used for a time interval $t \in [0, 10\pi]$.

present results obtained with the fourth-order Runge-Kutta method.

The corresponding phase space plot is shown in Fig. 14.6, for the same parameters as in Fig. **??**. 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. **??**. In addition, we should note that this periodic motion shows what we call resonant behavior since the the driving frequency of the force approaches the natural frequency of oscillation of the pendulum. This is essentially due to the fact that we are studying a linear system, yielding the well-known periodic motion. The non-linear system exhibits a much richer set of solutions and these can only be studied numerically.

In order to go beyond the well-known linear approximation we change the initial conditions to say $\theta_0 = 0.3x$ but keep the other parameters equal to the previous case. The curve for θ is

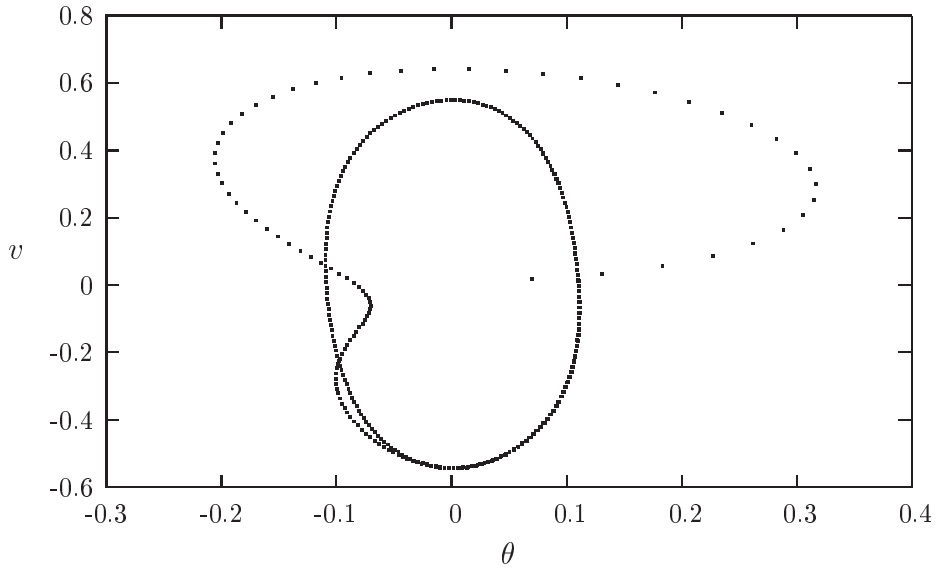


Figure 14.6: Phase-space curve of a linear damped pendulum with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 0.5$. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.01$.

shown in Fig. 14.7. 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. **??** that θ as function of time exhibits a rather different behavior from Fig. 14.6, even though the initial contiditions and all other parameters except $\hat{A}$ are the same.

If we then plot only the phase-space curve for the final orbit, we obtain the following figure We will explore these topics in more detail in Section 14.8 where we extende our discussion to the phenomena of period doubling and its link to chaotic motion.

14.7.2 The pendulum code

The program used to obtain the results discussed above is presented here. The program solves the pendulum equations for any angle θ with an external force $Acos(\omega t)$. It employes several methods for solving the two coupled differential equations, from Euler's method to adaptive size methods coupled with fourth-order Runge-Kutta. It is straightforward to apply this program to other systems which exhibit harmonic oscillations or change the functional form of the external force.

```
#include <stdio.h>
```

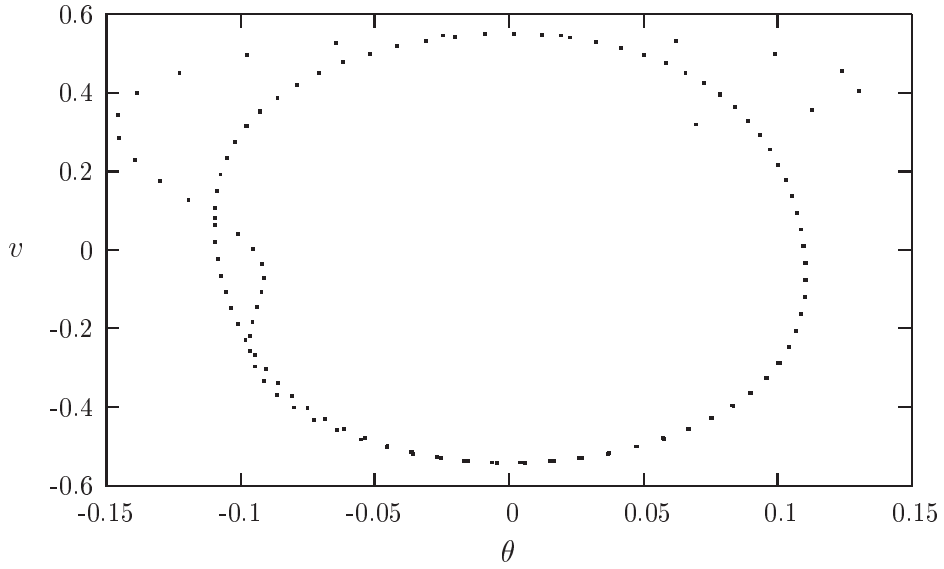


Figure 14.7: Plot of θ as function of time τ with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 0.5$. The mass of the pendulum is set equal to 1 kg and its length to 1 m. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.3$.

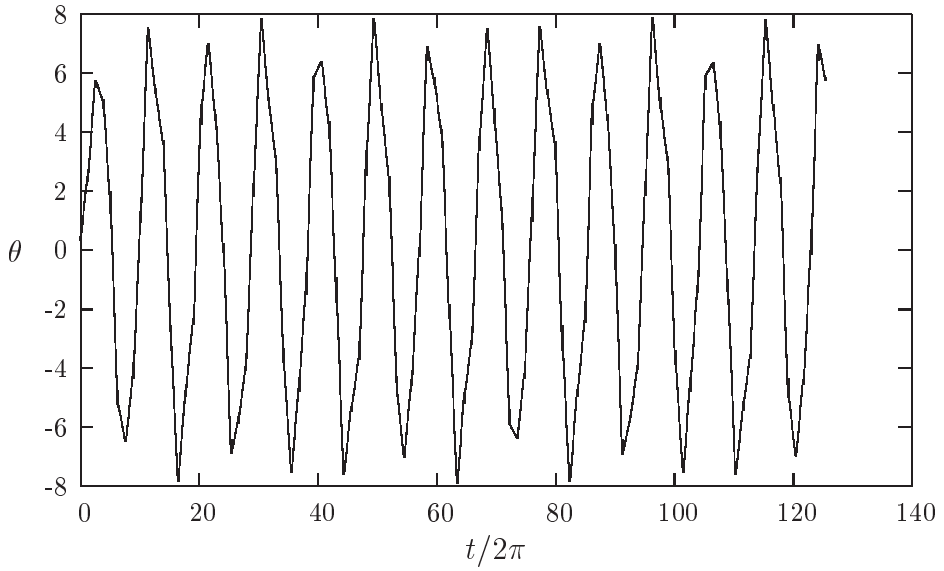


Figure 14.8: Phase-space curve with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 1.35$. The mass of the pendulum is set equal to 1 kg and its length $l = 1$ m.. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.3$.

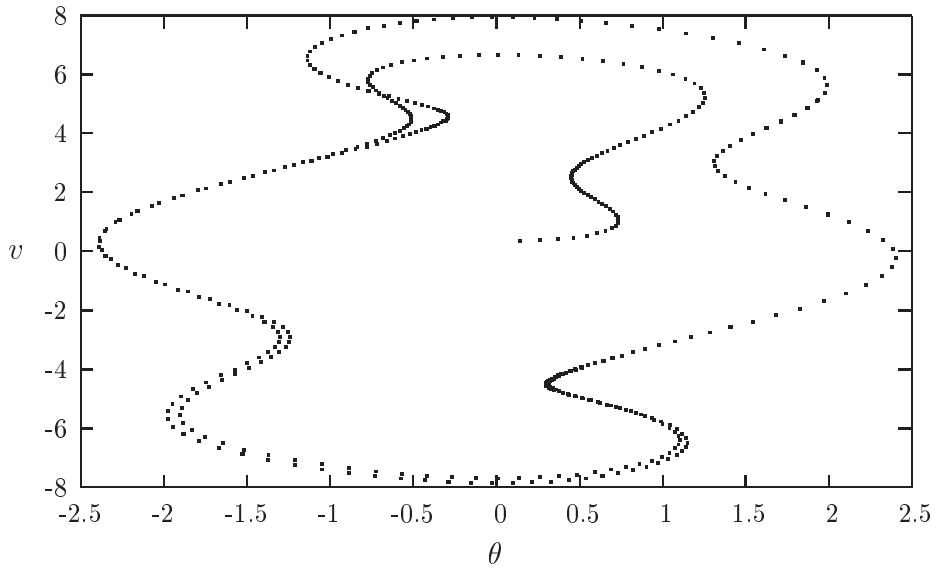


Figure 14.9: Phase-space curve for the attractor with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 1.35$. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.3$.

```
#include <iostream.h>
#include <math.h>
#include <fstream.h>
/*
Different methods for solving ODEs are presented
We are solving the following eqation:

m*l*(phi)'' + viscosity*(phi)' + m*g*sin(phi) = A*cos(omega*t)

If you want to solve similar equations with other values you have to
rewrite the methods 'derivatives' and 'initialise' and change the
   variables in the private
part of the class Pendulum

At first we rewrite the equation using the following definitions:

omega_0 = sqrt(g*l)
t_roof = omega_0*t
omega_roof = omega/omega_0
Q = (m*g)/(omega_0*reib)
A_roof = A/(m*g)

and we get a dimensionless equation
```

```
(phi)'' + 1/Q*(phi)' + sin(phi) = A_roof*cos(omega_roof*t_roof)

This equation can be written as two equations of first order:

(phi)' = v
(v)' = -v/Q - sin(phi) +A_roof*cos(omega_roof*t_roof)

All numerical methods are applied to the last two equations.
The algorithms are taken from the book "An introduction to computer
   simulation methods"
*/

class pendelum
 {
 private:
   double Q, A_roof, omega_0, omega_roof,g; //
   double y[2];          //for the initial-values of phi and v
   int n;                // how many steps
   double delta_t,delta_t_roof;

 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
 };

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)''
}


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);
}
```

```
void pendelum::euler()
{ //using simple euler-method
  int i;
  double yout[2],y_h[2];
  double t_h;

  y_h[0]=y[0];
  y_h[1]=y[1];
  t_h=0;
  ofstream fout("euler.out");
  fout.setf(ios::scientific);
  fout.precision(20);
  for(i=1;i<=n;i++){
    derivatives(t_h,y_h,yout);
    yout[1]=y_h[1]+yout[1]*delta_t_roof;
    yout[0]=y_h[0]+yout[0]*delta_t_roof;
    // Calculation with dimensionless values
    fout<<i*delta_t<<"\t\t"<<yout[0]<<"\t\t"<<yout[1]<<"\n";
    t_h+=delta_t_roof;
    y_h[1]=yout[1];
    y_h[0]=yout[0];
  }
  fout.close;
}

void pendelum::euler_cromer()
{
  int i;
  double t_h;
  double yout[2],y_h[2];

  t_h=0;
  y_h[0]=y[0];  //phi
  y_h[1]=y[1];  //v
  ofstream fout("ec.out");
  fout.setf(ios::scientific);
  fout.precision(20);
  for(i=1; i<=n; i++){
    derivatives(t_h,y_h,yout);
    yout[1]=y_h[1]+yout[1]*delta_t_roof;
    yout[0]=y_h[0]+yout[1]*delta_t_roof;
    // The new calculated value of v is used for calculating phi
    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;
}

void pendelum::midpoint()
{
  int i;
  double t_h;
  double yout[2],y_h[2];

  t_h=0;
  y_h[0]=y[0];   //phi
  y_h[1]=y[1];   //v
  ofstream fout("midpoint.out");
  fout.setf(ios::scientific);
  fout.precision(20);
  for(i=1; i<=n; i++){
    derivatives(t_h,y_h,yout);
    yout[1]=y_h[1]+yout[1]*delta_t_roof;
    yout[0]=y_h[0]+0.5*(yout[1]+y_h[1])*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;
}


void pendelum::euler_richardson()
{
  int i;
  double t_h,t_m;
  double yout[2],y_h[2],y_m[2];

  t_h=0;
  y_h[0]=y[0];   //phi
  y_h[1]=y[1];   //v
  ofstream fout("er.out");
  fout.setf(ios::scientific);
  fout.precision(20);
  for(i=1; i<=n; i++){
    derivatives(t_h,y_h,yout);
    y_m[1]=y_h[1]+0.5*yout[1]*delta_t_roof;
```

```
    y_m[0]=y_h[0]+0.5*y_h[1]*delta_t_roof;
    t_m=t_h+0.5*delta_t_roof;
    derivatives(t_m,y_m,yout);
    yout[1]=y_h[1]+yout[1]*delta_t_roof;
    yout[0]=y_h[0]+y_m[1]*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;
}

void pendelum::half_step()
{
  /*We are using the half_step_algorith.
    The algorithm is not self-starting, so we calculate
    v_1/2 by using the Euler algorithm. */

  int i;
  double t_h;
  double yout[2],y_h[2];

  t_h=0;
  y_h[0]=y[0];  //phi
  y_h[1]=y[1];  //v
  ofstream fout("half_step.out");
  fout.setf(ios::scientific);
  fout.precision(20);
  /*At first we have to calculate v_1/2
    For this we use Euler's method:
    v_'1/2 = v_0 + 1/2*a_0*delta_t_roof
    For calculating a_0 we have to start derivatives
  */
  derivatives(t_h,y_h,yout);
  yout[1]=y_h[1]+0.5*yout[1]*delta_t_roof;
  yout[0]=y_h[0]+yout[1]*delta_t_roof;
  fout<<delta_t<<"\t\t"<<yout[0]<<"\t\t"<<yout[1]<<"\n";
  y_h[0]=yout[0];
  y_h[1]=yout[1];
  for(i=2; i<=n; i++){
    derivatives(t_h,y_h,yout);
    yout[1]=y_h[1]+yout[1]*delta_t_roof;
    yout[0]=y_h[0]+yout[1]*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;
}

void pendelum::rk2()
{
  /*We are using the second-order-Runge-Kutta-algorithm
    We have to calculate the parameters k1 and k2 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],y_k[2];

  t_h=0;
  y_h[0]=y[0];  //phi
  y_h[1]=y[1];  //v
  ofstream fout("rk2.out");
  fout.setf(ios::scientific);
  fout.precision(20);
  for(i=1; i<=n; i++){
    /*Calculation of k1 */
    derivatives(t_h,y_h,yout);
    k1[1]=yout[1]*delta_t_roof;
    k1[0]=yout[0]*delta_t_roof;
    y_k[0]=y_h[0]+k1[0]*0.5;
    y_k[1]=y_h[1]+k2[1]*0.5;
    /*Calculation of k2 */
    derivatives(t_h+delta_t_roof*0.5,y_k,yout);
    k2[1]=yout[1]*delta_t_roof;
    k2[0]=yout[0]*delta_t_roof;
    yout[1]=y_h[1]+k2[1];
    yout[0]=y_h[0]+k2[0];
    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;
```

```
}

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;
}



void pendelum::asc()
{
  /*
    We are using the Runge-Kutta-algorithm with adaptive stepsize
       control
    according to "Numerical Recipes in C", S. 574 ff.

    At first we calculate y(x+h) using rk4-method  => y1
    Then we calculate y(x+h) using two times rk4-method at x+h/2 and x
       +h  => y2

    The difference between these values is called "delta" If it is
       smaller than a given value,
```

```
    we calculate y(x+h) by  y2 + (delta)/15 (page 575, Numerical R.)

    If delta is not smaller than ... we calculate a new stepsize using
    h_new=(Safety)*h_old*(.../delta)^(0.25) where "Safety" is constant
        (page 577 N.R.)
    and start again with calculating y(x+h)...
   */
  int i;
  double t_h,h_alt,h_neu,hh,errmax;
  double yout[2],y_h[2],y_m[2],y1[2],y2[2], delta[2], yscal[2];

  const double eps=1.0e-6;
  const double safety=0.9;
  const double errcon=6.0e-4;
  const double tiny=1.0e-30;

  t_h=0;
  y_h[0]=y[0];   //phi
  y_h[1]=y[1];   //v
  h_neu=delta_t_roof;
  ofstream fout("asc.out");
  fout.setf(ios::scientific);
  fout.precision(20);
  for(i=0;i<=n;i++){
    /* The error is scaled against yscal
       We use a yscal of the form yscal = fabs(y[i]) + fabs(h*
          derivatives[i])
       (N.R. page 567)
    */
    derivatives(t_h,y_h,yout);
    yscal[0]=fabs(y[0])+fabs(h_neu*yout[0])+tiny;
    yscal[1]=fabs(y[1])+fabs(h_neu*yout[1])+tiny;
    /* the do-while-loop is used until the */
    do{
      /* Calculating y2 by two half steps */
      h_alt=h_neu;
      hh=h_alt*0.5;
      rk4_step(t_h, y_h, y_m, hh);
      rk4_step(t_h+hh,y_m,y2,hh);
      /* Calculating y1 by one normal step */
      rk4_step(t_h,y_h,y1,h_alt);
      /* Now we have two values for phi and v at the time t_h + h  in
         y2 and y1
         We can now calculate the delta for phi and v
      */
```

```
      delta[0]=fabs(y1[0]-y2[0]);
      delta[1]=fabs(y1[1]-y2[1]);
      errmax=(delta[0]/yscal[0] > delta[1]/yscal[1] ? delta[0]/yscal
         [0] : delta[1]/yscal[1]);

      /*We scale delta against the constant yscal
        Then we take the biggest one and call it errmax */
      errmax=(double)errmax/eps;
      /*We divide errmax by eps and have only    */
      h_neu=safety*h_alt*exp(-0.25*log(errmax));
    }while(errmax>1.0);
    /*Now we are outside the do-while-loop and have a delta which is
       small enough
      So we can calculate the new values of phi and v
    */
    yout[0]=y_h[0]+delta[0]/15.0;
    yout[1]=y_h[1]+delta[1]/15.0;
    fout<<(double)(t_h+h_alt)/omega_0<<"\t\t"<<yout[0]<<"\t\t"<<yout
       [1]<<"\n";
    // Calculating of the new stepsize
    h_neu=(errmax > errcon ? safety*h_alt*exp(-0.20*log(errmax))
        : 4.0*h_alt);
    y_h[0]=yout[0];
    y_h[1]=yout[1];
    t_h+=h_neu;
  }
}


int main()
{
  pendelum testcase;
  testcase.initialise();
  testcase.euler();
  testcase.euler_cromer();
  testcase.midpoint();
  testcase.euler_richardson();
  testcase.half_step();
  testcase.rk2();
  testcase.rk4();
  return 0;
}  // end of main function
```

14.8 Physics project: Period doubling and chaos

in preparation

In Fig. **??** we have kept the same constants as in the previous section except for $\hat{A}$ which we

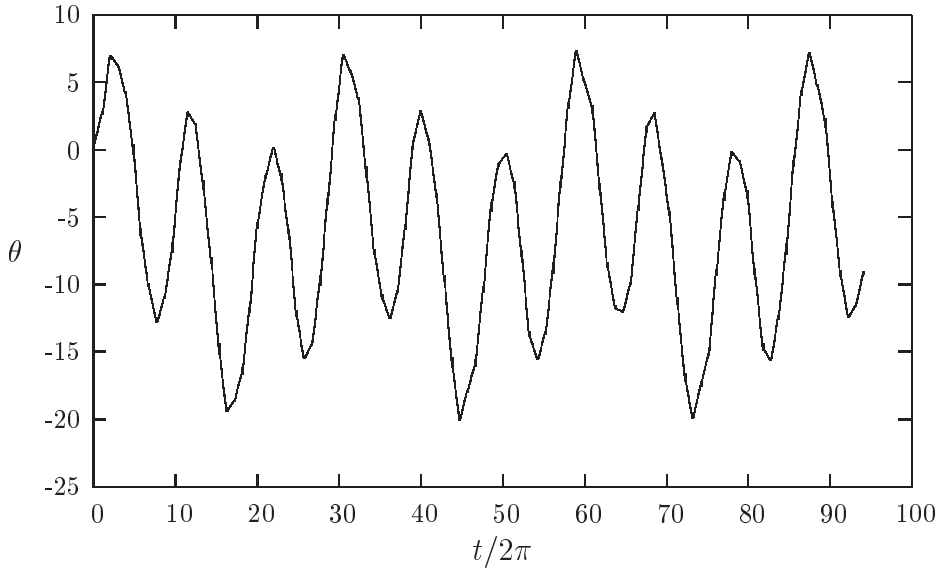


Figure 14.10: Phase-space curve with $Q = 2$, $\hat{\omega} = 2/3$ and $\hat{A} = 1.52$. The mass of the pendulum is set equal to 1 kg and its length $l = 1$ m. The inital velocity is $\hat{v}_0 = 0$ and $\theta_0 = 0.3$.

now set to $\hat{A} = 1.52$.

14.9 Physics Project: studies of neutron stars

In the pendulum example we rewrote the equations as two differential equations in terms of so-called dimensionless variables. One should always do that. There are at least two good reasons for doing this.

- By rewriting the equations as dimensionless ones, the program will most likely be easier to read, with hopefully a better possibility of spotting eventual errors. In addtion, the various constants which are pulled out of the equations in the process of rendering the equations dimensionless, are reintroduced at the end of the calculation. If one of these constants is not correctly defined, it is easier to spot an eventual error.

- In many physics applications, variables which enter a differential equation, may differ by orders of magnitude. If we were to insist on not using dimensionless quantities, such differences can cause serious problems with respect to loss of numerical precision.

An example which demonstrates these features is the set of equations for gravitational equilibrium of a neutron star. We will not solve these equations numerically here, rather, we will limit ourselves to merely rewriting these equations in a dimensionless form.

14.9.1 The equations for a neutron star

The discovery of the neutron by Chadwick in 1932 prompted Landau to predict the existence of neutron stars. The birth of such stars in supernovae explosions was suggested by Baade and Zwicky 1934. First theoretical neutron star calculations were performed by Tolman, Oppenheimer and Volkoff in 1939 and Wheeler around 1960. Bell and Hewish were the first to discover a neutron star in 1967 as a *radio pulsar*. The discovery of the rapidly rotating Crab pulsar (rapidly rotating neutron star) in the remnant of the Crab supernova observed by the chinese in 1054 A.D. confirmed the link to supernovae. Radio pulsars are rapidly rotating with periods in the range 0.033 s $\leq P \leq 4.0$ s. They are believed to be powered by rotational energy loss and are rapidly spinning down with period derivatives of order $\dot{P} \sim 10^{-12} - 10^{-16}$. Their high magnetic field B leads to dipole magnetic braking radiation proportional to the magnetic field squared. One estimates magnetic fields of the order of $B \sim 10^{11} - 10^{13}$ G. The total number of pulsars discovered so far has just exceeded 1000 before the turn of the millenium and the number is increasing rapidly.

The physics of compact objects like neutron stars offers an intriguing interplay between nuclear processes and astrophysical observables. Neutron stars exhibit conditions far from those encountered on earth; typically, expected densities ρ of a neutron star interior are of the order of 10^3 or more times the density $\rho_d \approx 4 \cdot 10^{11}$ g/cm^3 at 'neutron drip', the density at which nuclei begin to dissolve and merge together. Thus, the determination of an equation of state (EoS) for dense matter is essential to calculations of neutron star properties. The EoS determines properties such as the mass range, the mass-radius relationship, the crust thickness and the cooling rate. The same EoS is also crucial in calculating the energy released in a supernova explosion. Clearly, the relevant degrees of freedom will not be the same in the crust region of a neutron star, where the density is much smaller than the saturation density of nuclear matter, and in the center of the star, where density is so high that models based solely on interacting nucleons are questionable. Neutron star models including various so-called realistic equations of state result in the following general picture of the interior of a neutron star. The surface region, with typical densities $\rho < 10^6$ g/cm^3, is a region in which temperatures and magnetic fields may affect the equation of state. The outer crust for 10^6 g/cm^3 $< \rho < 4 \cdot 10^{11}$g/cm^3 is a solid region where a Coulomb lattice of heavy nuclei coexist in β-equilibrium with a relativistic degenerate electron gas. The inner crust for $4 \cdot 10^{11}$ g/cm^3 $< \rho < 2 \cdot 10^{14}$g/cm^3 consists of a lattice of neutron-rich nuclei together with a superfluid neutron gas and an electron gas. The neutron liquid for $2 \cdot 10^{14}$ g/cm^3 $< \rho < \cdot 10^{15}$g/cm^3 contains mainly superfluid neutrons with a smaller concentration of superconducting protons and normal electrons. At higher densities, typically $2 - 3$ times nuclear matter saturation density, interesting phase transitions from a phase with just nucleonic degrees of freedom to quark matter may take place. Furthermore, one may have a mixed phase of quark and nuclear matter, kaon or pion condensates, hyperonic matter, strong magnetic fields in young stars etc.

14.9.2 Equilibrium equations

If the star is in thermal equilibrium, the gravitational force on every element of volume will be balanced by a force due to the spacial variation of the pressure P. The pressure is defined by the equation of state (EoS), recall e.g., the ideal gas $P = Nk_BT$. The gravitational force which acts on an element of volume at a distance r is given by

$$F_{Grav} = -\frac{Gm}{r^2}\rho/c^2, \tag{14.88}$$

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

$$m(r) = \frac{4\pi}{c^2}\int_0^r \rho(r')r'^2dr' \tag{14.89}$$

which gives rise to a differential equation for mass and density

$$\frac{dm}{dr} = 4\pi r^2\rho(r)/c^2. \tag{14.90}$$

When the star is in equilibrium we have

$$\frac{dP}{dr} = -\frac{Gm(r)}{r^2}\rho(r)/c^2. \tag{14.91}$$

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.

14.9.3 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

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

list here the units of the various quantities and in case of physical constants, also their values. A bracketed symbol like $[P]$ stands for the unit of the quantity inside the brackets.

We introduce therefore dimensionless quantities for the radius $\overline{r} = r/R_0$, mass-energy density $\overline{\rho} = \rho/\rho_s$, pressure $\overline{P} = P/\rho_s$ and mass $\overline{m} = m/M_0$.

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

$$\frac{dM_0\overline{m}}{dR_0\overline{r}} = 4\pi R_0^2\overline{r}^2\rho_s\overline{\rho}, \tag{14.92}$$

yielding

$$\frac{d\overline{m}}{d\overline{r}} = 4\pi R_0^3\overline{r}^2\rho_s\overline{\rho}/M_0. \tag{14.93}$$

If these equations should be dimensionless we must demand that

$$4\pi R_0^3\rho_s/M_0 = 1. \tag{14.94}$$

Correspondingly, we have for the pressure equation

$$\frac{d\rho_s\overline{P}}{dR_0\overline{r}} = -GM_0\frac{\overline{m}\rho_s\overline{\rho}}{R_0^2\overline{r}^2} \tag{14.95}$$

and since this equation should also be dimensionless, we will have

$$GM_0/R_0 = 1. \tag{14.96}$$

This means that the constants R_0 and M_0 which will render the equations dimensionless are given by

$$R_0 = \frac{1}{\sqrt{\rho_s G 4\pi}}, \tag{14.97}$$

and

$$M_0 = \frac{4\pi\rho_s}{(\sqrt{\rho_s G 4\pi})^3}. \tag{14.98}$$

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

$$\frac{d\overline{P}}{d\overline{r}} = -\frac{\overline{m\rho}}{\overline{r}^2}, \qquad \frac{d\overline{m}}{d\overline{r}} = \overline{r}^2\overline{\rho}. \tag{14.99}$$

14.9.4 Program and selected results

in preparation

14.10 Physics project: Systems of linear differential equations

in preparation

Chapter 15

Two point boundary value problems.

15.1 Introduction

This chapter serves as an intermediate step to the next chapter on partial differential equations. Partial differential equations involve both boundary conditions and differential equations with functions depending on more than one variable. Here we focus on the problem of boundary conditions with just one variable. When diffential equations are required to satify boundary conditions at more than one value of the independent variable, the resulting problem is called a *two point boundary value problem.* As the terminology indicates, the most common case by far is when boundary conditions are supposed to be satified 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 restricted to be finite everywhere (in particular at $r = 0$) and for bound states the functions must go to zero at infinity. In this chapter we will discuss the solution of the one-particle Schödinger equation and apply the method to the hydrogen atom.

15.2 Schrödinger equation

We discuss the numerical solution of the Schrödinger equation for the case of a particle with mass m moving in a spherical symmetric potential.

The initial eigenvalue equation reads

$$\widehat{\mathbf{H}}\psi(\vec{r}) = (\widehat{\mathbf{T}} + \widehat{\mathbf{V}})\psi(\vec{r}) = E\psi(\vec{r}). \tag{15.1}$$

In detail this gives

$$\left(-\frac{\hbar^2}{2m}\nabla^2 + V(r)\right)\psi(\vec{r}) = E\psi(\vec{r}). \tag{15.2}$$

The eigenfunction in spherical coordinates takes the form

$$\psi(\vec{r}) = R(r)Y_l^m(\theta,\phi), \tag{15.3}$$

and the radial part $R(r)$ is a solution to

$$-\frac{\hbar^2}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr}-\frac{l(l+1)}{r^2}\right)+V(r)R(r)=ER(r). \tag{15.4}$$

Then we substitute $R(r)=(1/r)u(r)$ and obtain

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r)+\left(V(r)+\frac{l(l+1)}{r^2}\frac{\hbar^2}{2m}\right)u(r)=Eu(r). \tag{15.5}$$

We introduce a dimensionless variable $\rho=(1/\alpha)r$ where α is a constant with dimension length and get

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(r)+\left(V(\rho)+\frac{l(l+1)}{\rho^2}\frac{\hbar^2}{2m\alpha^2}\right)u(\rho)=Eu(\rho). \tag{15.6}$$

In our case we are interested in attractive potentials

$$V(r)=-V_0f(r), \tag{15.7}$$

where $V_0>0$ and analyze bound states where $E<0$. The final equation can be written as

$$\frac{d^2}{d\rho^2}u(\rho)+k(\rho)u(\rho)=0, \tag{15.8}$$

where

$$\begin{aligned} k(\rho) &= \gamma\left(f(\rho)-\frac{1}{\gamma}\frac{l(l+1)}{\rho^2}-\epsilon\right)\\ \gamma &= \frac{2m\alpha^2V_0}{\hbar^2}\\ \epsilon &= \frac{|E|}{V_0} \end{aligned} \tag{15.9}$$

15.3 Numerov's method

Eq. (15.8) is a second order differential equation without any first order derivatives. Numerov's method is designed to solve such an equation numerically, achieving an extra order of precision.

Let us start with the Taylor expansion of the wave function

$$u(\rho+h)=u(\rho)+hu^{(1)}(\rho)+\frac{h^2}{2!}u^{(2)}(\rho)+\frac{h^3}{3!}u^{(3)}(\rho)+\frac{h^4}{4!}u^{(4)}(\rho)+\cdots \tag{15.10}$$

where $u^{(n)}(\rho)$ is a shorthand notation for the nth derivative $d^n/d\rho^n$. Because the corresponding Taylor expansion of $u(\rho-h)$ has odd powers of h appearing with negative signs, all odd powers cancel when we add $u(\rho+h)$ and $u(\rho-h)$

$$u(\rho+h)+u(\rho-h)\approx 2u(\rho)+h^2u^{(2)}(\rho)+\frac{h^4}{12}u^{(4)}(\rho)+O(h^6). \tag{15.11}$$

Then we obtain

$$u^{(2)}(\rho) \approx \frac{u(\rho+h)+u(\rho-h)-2u(\rho)}{h^2} - \frac{h^2}{12}u^{(4)}(\rho) + O(h^4). \tag{15.12}$$

To eliminate the fourth-derivative term we apply the operator $(1+\frac{h^2}{12}\frac{d^2}{d\rho^2})$ to Eq. (15.8) and obtain a modified equation

$$h^2u^{(2)}(\rho) + \frac{h^2}{12}u^{(4)}(\rho) + k(\rho)u(\rho) + \frac{h^2}{12}\frac{d^2}{d\rho^2}\left(k(\rho)u(\rho)\right) \approx 0. \tag{15.13}$$

In this expression the $u^{(4)}$ terms cancel. To treat the general ρ dependence of $k(\rho)$ we approximate the second derivative of $k(\rho)u(\rho)$ by

$$\frac{d^2(k(\rho)u(\rho))}{d\rho^2} \approx \frac{(k(\rho+h)u(\rho+h)+k(\rho)u(\rho)) + (k(\rho-h)u(\rho-h)+k(\rho)u(\rho))}{h^2}, \tag{15.14}$$

and the following numerical algorithm is obtained

$$u_{(i+2)} \approx \frac{2\left(1-\frac{5}{12}h^2k_{(i+1)}u_{(i+1)}\right) - \left(1+\frac{5}{12}h^2k_iu_i\right)}{1+\frac{h^2}{12}k_{(i+2)}} \tag{15.15}$$

where $\rho = ih$, $k_i = k(ih)$ and $u_i = u(ih)$ etc.

15.4 Schrödinger equation for a spherical box potential

Let us now specify the spherical symmetric potential to

$$f(r) = \begin{cases} 1 & \text{for} \quad r \le a \\ -0 & \quad\quad r > a \end{cases} \tag{15.16}$$

and choose $\alpha = a$. Then

$$k(\rho) = \gamma \begin{cases} 1-\epsilon-\frac{1}{\gamma}\frac{l(l+1)}{\rho^2} & \text{for} \quad r \le a \\ -\epsilon--\frac{1}{\gamma}\frac{l(l+1)}{\rho^2} & \quad\quad r > a \end{cases} \tag{15.17}$$

The eigenfunctions in Eq. (15.2) are subject to conditions which limit the possible solutions. Of importance for the present example is that $u(\vec{r})$ must be finite everywhere and $\int |u(\vec{r})|^2 d\tau$ must be finite. The last condition means that $rR(r) \longrightarrow 0$ for $r \longrightarrow \infty$. These conditions imply that $u(r)$ must be finite at $r = 0$ and $u(r) \longrightarrow 0$ for $r \longrightarrow \infty$.

15.4.1 Analysis of $u(\rho)$ at $\rho = 0$

For small ρ Eq. (15.8) reduces to

$$\frac{d^2}{d\rho^2}u(\rho) - \frac{l(l+1)}{\rho^2}u(\rho) = 0, \tag{15.18}$$

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

$$u(\rho) = \rho^{l+1} \quad \text{for small } \rho \tag{15.19}$$

15.4.2 Analysis of $u(\rho)$ for $\rho \longrightarrow \infty$

For large ρ Eq. (15.8) reduces to

$$\frac{d^2}{d\rho^2}u(\rho) - \gamma\epsilon u(\rho) = 0 \quad \gamma > 0, \tag{15.20}$$

with solutions $u(\rho) = \exp(\pm\gamma\epsilon\rho)$ and the condition for large ρ means that our numerical solution must satisfy

$$u(\rho) = e^{-\gamma\epsilon\rho} \quad \text{for large } \rho \tag{15.21}$$

15.5 Numerical procedure

The eigenvalue problem in Eq. (15.8) 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 exponetially as $u(\rho) = e^{-\gamma\epsilon\rho}$. However, due to numerical inaccuracy the solution will contain small admixtures of the undesireable exponential growing function $u(\rho) = e^{+\gamma\epsilon\rho}$. The final solution will then become unstable. Therefore, it is better to generate two solutions, with one starting from small values of ρ and integrate outwards to some matching point $\rho = \rho_m$. We call that function $u^<(\rho)$. The next solution $u^>(\rho)$ is then obtained by integrating from some large value ρ where the potential is of no importance, and inwards to the same matching point ρ_m. Due to the quantum mechanical requirements the logarithmic derivative at the matching point ρ_m should be well defined. We obtain the following condition

$$\frac{\frac{d}{d\rho}u^<(\rho)}{u^<(\rho)} = \frac{\frac{d}{d\rho}u^>(\rho)}{u^>(\rho)} \quad \text{at} \quad \rho = \rho_m. \tag{15.22}$$

We can modify this expression by normalizing the function $u^<u^<(\rho_m) = Cu^>u^<(\rho_m)$. Then Eq. (15.22) becomes

$$\frac{d}{d\rho}u^<(\rho) = \frac{d}{d\rho}u^>(\rho) \quad \text{at} \quad \rho = \rho_m \tag{15.23}$$

For an arbitary value of the eigenvalue Eq. (15.22) will not be satisfied. Thus the numerical procedure will be to iterate for different eigenvalues until Eq. (15.23) is satisfied.

We can calculate the first order derivatives by

$$\frac{d}{d\rho}u^<(\rho_m) \approx \frac{u^<(\rho_m) - u^<(\rho_m - h)}{h}$$
$$\frac{d}{d\rho}u^>(\rho_m) \approx \frac{u^>(\rho_m + h) - u^>(\rho_m)}{h} \tag{15.24}$$

Thus the criterium for a proper eigenfunction will be

$$f = u^<(\rho_m - h) - u^>(\rho_m + h) \tag{15.25}$$

15.6 Algorithm for solving Schrödinger's equation

of the solution. 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 8.

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 $\max\_\text{iter}$ and the desired numerical precision.
- Search then for the roots of the function $f(E)$, where the root(s) is(are) in the interval $E \in [E_{\min}, E_{\max}]$ using e.g., the bisection method. 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. (15.25) we have a solution. The matching code is given below.

The function $f(E)$ above receives as input a guess for the energy. In the version implemented below, we use the standard three-point formula for the second derivative, namely

$$f_0'' \approx \frac{f_h - 2f_0 + f_{-h}}{h^2}.$$

We leave it as an exercise to the reader to implement Numerov's algorithm.

```
//
// The function
//         f()
// calculates the wave function at fixed energy eigenvalue.
//

void f(double step, int max_step, double energy, double *w, double *wf
    )
{
    int       loop, loop_1,match;
    double    const sqrt_pi = 1.77245385091;
    double    fac, wwf, norm;
// adding the energy guess to the array containing the potential
    for(loop = 0; loop <= max_step; loop ++) {
       w[loop] = (w[loop] - energy) * step * step + 2;
    }
// integrating from large r-values
    wf[max_step]     = 0.0;
    wf[max_step - 1] = 0.5 * step * step;
// search for matching point
    for(loop = max_step - 2; loop > 0; loop--) {
       wf[loop] = wf[loop + 1] * w[loop + 1] - wf[loop + 2];
       if(wf[loop] <= wf[loop + 1]) break;
    }
    match = loop + 1;
    wwf   = wf[match];
// start integrating up to matching point from r =0
    wf[0] = 0.0;
    wf[1] = 0.5 * step * step;
    for(loop = 2; loop <= match; loop++) {
       wf[loop] = wf[loop -1] * w[loop - 1] - wf[loop - 2];
       if(fabs(wf[loop]) > INFINITY) {
          for(loop_1 = 0; loop_1 <= loop; loop_1++) {
             wf[loop_1] /= INFINITY;
          }
       }
    }
// now implement the test of Eq. (10.25)
    return fabs(wf[match-1]-wf[match+1]);
```

```
} // End: funtion plot()
```

Chapter 16

Partial differential equations

16.1 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

$$\frac{\partial^2 U}{\partial x^2} = A\frac{\partial^2 U}{\partial t^2}, \tag{16.1}$$

where A is a constant. 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 the constant $A = c^2$, with c the speed of light. It is rather straightforward to extend this equation to two or three dimension. In two dimensions we have

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = A\frac{\partial^2 U}{\partial t^2}, \tag{16.2}$$

In Chapter 10 we saw another case of a partial differential equation widely used in the Natural Sciences, namely the diffusion equation whose one-dimensional version we derived from a Markovian random walk. It reads

$$\frac{\partial^2 U}{\partial x^2} = A\frac{\partial U}{\partial t}, \tag{16.3}$$

and A is in this case called the diffusion constant. It can be used to model a wide selection of diffusion processes, from molecules to the diffusion of heat in a given material.

Another familiar equation from electrostatics is Laplace's equation, which looks similar to the wave equation in Eq. (16.1) except that we have set $A = 0$

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0, \tag{16.4}$$

or if we have a finite electric charge represented by a charge density $\rho(\mathbf{x})$ we have the familiar Poisson equation

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = -4\pi\rho(\mathbf{x}). \tag{16.5}$$

However, although parts of these equation look similar, we will see below that different solution strategies apply. In this chapter we focus essentially on so-called finite difference schemes and explicit and implicit methods. The more advanced topic of finite element methods is relegated to the part on advanced topics.

A general partial differential equation in $2+1$-dimensions (with 2 standing for the spatial coordinates x and y and 1 for time) reads

$$A(x,y)\frac{\partial^2 U}{\partial x^2} + B(x,y)\frac{\partial^2 U}{\partial x \partial y} + C(x,y)\frac{\partial^2 U}{\partial y^2} = F(x,y,U,\frac{\partial U}{\partial x},\frac{\partial U}{\partial y}), \tag{16.6}$$

and if we set

$$B = C = 0, \tag{16.7}$$

we recover the $1+1$-dimensional diffusion equation which is an example of a so-called parabolic partial differential equation. With

$$B = 0, \qquad AC < 0 \tag{16.8}$$

we get the $2+1$-dim wave equation which is an example of a so-called hyperolic PDE, where more generally we have $B^2 > AC$. For $B^2 < AC$ we obtain a so-called ellyptic PDE, with the Laplace equation in Eq. (16.4) as one of the classical examples. These equations can all be easily extended to non-linear partial differential equations and $3+1$ dimensional cases.

The aim of this chapter is to present some of the most familiar difference methods and their eventual implementations.

16.2 Diffusion equation

The let us assume that the diffusion of heat through some material is proportional with the temperature gradient $T(\mathbf{x},t)$ and using conservation of energy we arrive at the diffusion equation

$$\frac{\kappa}{C\rho}\nabla^2 T(\mathbf{x},t) = \frac{\partial T(\mathbf{x},t)}{\partial t} \tag{16.9}$$

where C is the specific heat and ρ the density of the material. Here we let the density be represented by a constant, but there is no problem introducing an explicit spatial dependence, viz.,

$$\frac{\kappa}{C\rho(\mathbf{x},t)}\nabla^2 T(\mathbf{x},t) = \frac{\partial T(\mathbf{x},t)}{\partial t}. \tag{16.10}$$

Setting all constants equal to the diffusion constant D, i.e.,

$$D = \frac{C\rho}{\kappa}, \tag{16.11}$$

we arrive at

$$\nabla^2 T(\mathbf{x},t) = D\frac{\partial T(\mathbf{x},t)}{\partial t}. \tag{16.12}$$

Specializing to the $1+1$-dimensional case we have

$$\frac{\partial^2 T(x,t)}{\partial x^2} = D\frac{\partial T(x,t)}{\partial t}. \tag{16.13}$$

We note that the dimension of D is time/length^2. Introducing the dimensional variables $\alpha\hat{x} = x$ we get

$$\frac{\partial^2 T(x,t)}{\alpha^2\partial \hat{x}^2} = D\frac{\partial T(x,t)}{\partial t}, \tag{16.14}$$

and since α is just a constant we could define $\alpha^2 D = 1$ or use the last expression to define a dimensionless time-variable $\hat{t}$. This yields a simplified diffusion equation

$$\frac{\partial^2 T(\hat{x},\hat{t})}{\partial \hat{x}^2} = \frac{\partial T(\hat{x},\hat{t})}{\partial \hat{t}}. \tag{16.15}$$

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 $1+1$-dimensional partial differential equation is replaced by $T(\hat{x},\hat{t}) \to u(x,t)$.

16.2.1 Explicit scheme

In one dimension we have thus the following equation

$$\nabla^2 u(x,t) = \frac{\partial u(x,t)}{\partial t}, \tag{16.16}$$

or

$$u_{xx} = u_t, \tag{16.17}$$

with initial conditions, i.e., the conditions at $t = 0$,

$$u(x,0) = g(x) \quad 0 \leq x \leq L \tag{16.18}$$

with $L = 1$ the length of the x-region of interest. The boundary conditions are

$$u(0,t) = a(t) \quad t \geq 0, \tag{16.19}$$

and

$$u(L,t) = b(t) \quad t \geq 0, \tag{16.20}$$

where $a(t)$ and $b(t)$ are two functions which depend on time only, while $g(x)$ depends only on the position x. Our next step is to find a numerical algorithm for solving this equation. Here we recur to our familiar equal-step methods discussed in Chapter 3 and introduce different step lengths for the space-variable x and time t through the step length for x

$$\Delta x = \frac{1}{n+1} \tag{16.21}$$

and the time step length Δt. The position after i steps and time at time-step j are now given by

$$\begin{cases} t_j = j\Delta t & j \geq 0 \\ x_i = i\Delta x & 1 \leq i \leq n+1 \end{cases} \tag{16.22}$$

If we then use standard approximations for the derivatives we obtain

$$u_t \approx \frac{u(x,t+\Delta t)-u(x,t)}{\Delta t} = \frac{u(x_i,t_j+\Delta t)-u(x_i,t_j)}{\Delta t} \tag{16.23}$$

with a local approximation error $O(\Delta t)$ and

$$u_{xx} \approx \frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)}{\Delta x^2}, \tag{16.24}$$

or

$$u_{xx} \approx \frac{u(x_i+\Delta x,t_j)-2u(x_i,t_j)+u(x_i-\Delta x,t_j)}{\Delta x^2}, \tag{16.25}$$

with a local approximation error $O(\Delta x^2)$. Our approximation is to higher order in the coordinate space. This can be justified since in most cases it is the spatial dependence which causes numerical problems. These equations can be further simplified as

$$u_t \approx \frac{u_{i,j+1}-u_{i,j}}{\Delta t}, \tag{16.26}$$

and

$$u_{xx} \approx \frac{u_{i+i,j}-2u_{i,j}+u_{i-1,j}}{\Delta x^2}. \tag{16.27}$$

The one-dimensional diffusion equation can then be rewritten in its discretized version as

$$\frac{u_{i,j+1}-u_{i,j}}{\Delta t} = \frac{u_{i+i,j}-2u_{i,j}+u_{i-1,j}}{\Delta x^2}. \tag{16.28}$$

Defining $\alpha = \Delta t/\Delta x^2$ results in the explicit scheme

$$u_{i,j+1} = \alpha u_{i-1,j} + (1-2\alpha)u_{i,j} + \alpha u_{i+1,j}. \tag{16.29}$$

Since all the discretized initial values

$$u_{i,0} = g(x_i), \tag{16.30}$$

are known, then after one time-step the only unknown quantity is $u_{i,1}$ which is given by

$$u_{i,1} = \alpha u_{i-1,0} + (1-2\alpha)u_{i,0} + \alpha u_{i+1,0} = \alpha g(x_{i-1}) + (1-2\alpha)g(x_i) + \alpha g(x_{i+1}). \tag{16.31}$$

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 $u_{i,j}$ is

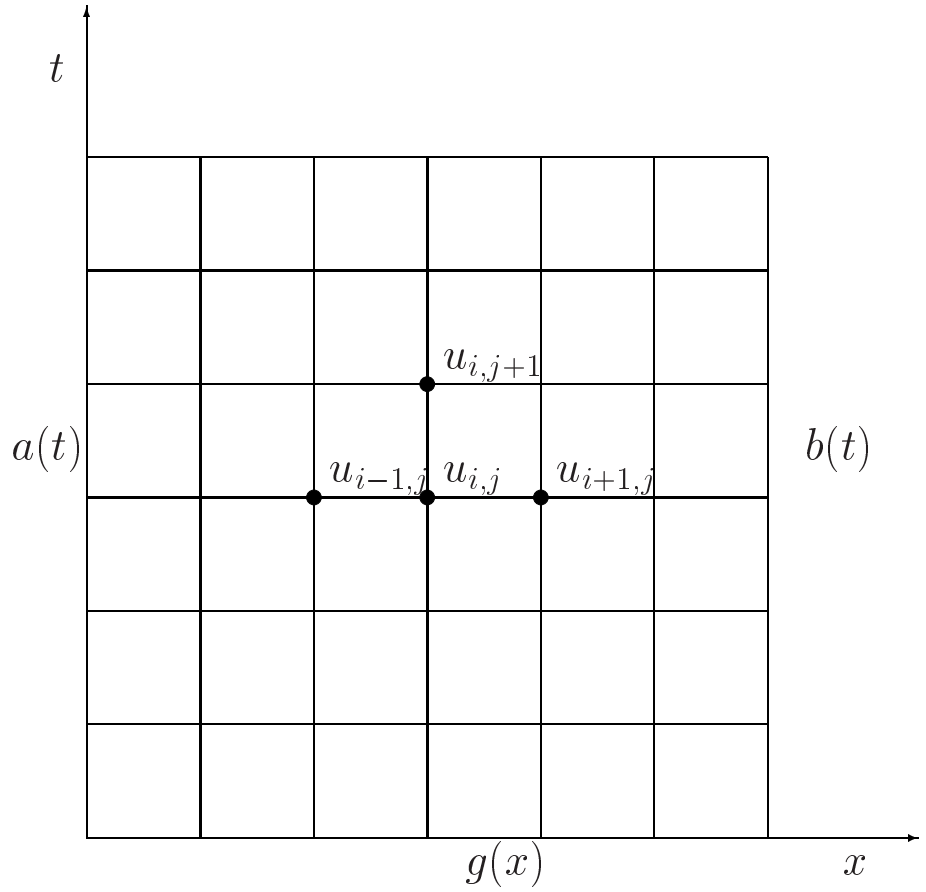


Figure 16.1: Discretization of the integration area used in the solution of the $1+1$-dimensional diffusion equation.

explicitely given by Eq. (16.29). The procedure is depicted in Fig. 16.2.1. The explicit scheme, although being rather simple to implement has a very weak stability condition given by

$$\Delta t/\Delta x^2 \leq 1/2 \tag{16.32}$$

We will now specialize to the case $a(t) = b(t) = 0$ which results in $u_{0,j} = u_{n+1,j} = 0$. We can then reformulate our partial differential equation through the vector V_j at the time $t_j = j\Delta t$

$$V_j = \begin{pmatrix} u_{1,j} \\ u_{2,j} \\ \dots \\ u_{n,j} \end{pmatrix}. \tag{16.33}$$

This results in a matrix-vector multiplication

$$V_{j+1} = \hat{A}V_j \tag{16.34}$$

with the matrix $\hat{A}$ given by

$$\hat{A} = \begin{pmatrix} 1-2\alpha & \alpha & 0 & 0\dots \\ \alpha & 1-2\alpha & \alpha & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & \alpha & 1-2\alpha \end{pmatrix} \tag{16.35}$$

which means we can rewrite the original partial differential equation as a set of matrix-vector multiplications

$$V_{j+1} = \hat{A}V_j = \cdots = \hat{A}^j V_0, \tag{16.36}$$

where V_0 is the initial vector at time $t = 0$ defined by the initial value $g(x)$.

16.2.2 Implicit scheme

In deriving the equations for the explicit scheme we started with the so-called forward formula for the first derivative, i.e., we used the discrete approximation

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t}. \tag{16.37}$$

However, there is nothing which hinders us from using the backward formula

$$u_t \approx \frac{u(x_i, t_j) - u(x_i, t_j - \Delta t)}{\Delta t}, \tag{16.38}$$

still with a truncation error which goes like $O(\Delta t)$. We could also have used a midpoint approximation for the first derivative, resulting in

$$u_t \approx \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j - \Delta t)}{2\Delta t}, \tag{16.39}$$

with a truncation error $O(\Delta t^2)$. Here we will stick to the backward formula and come back to the later below. For the second derivative we use however

$$u_{xx} \approx \frac{u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j)}{\Delta x^2}, \tag{16.40}$$

and define again $\alpha = \Delta t/\Delta x^2$. We obtain now

$$u_{i,j-1} = -\alpha u_{i-1,j} + (1 - 2\alpha)u_{i,j} - \alpha u_{i+1,j}. \tag{16.41}$$

Here $u_{i,j-1}$ is the only unknown quantity. Defining the matrix $\hat{A}$

$$\hat{A} = \begin{pmatrix} 1-2\alpha & -\alpha & 0 & 0\dots \\ -\alpha & 1-2\alpha & -\alpha & 0\dots \\ \dots & \dots & \dots & \dots \\ 0\dots & 0\dots & -\alpha & 1-2\alpha \end{pmatrix}, \tag{16.42}$$

we can reformulate again the problem as a matrix-vector multiplication

$$\hat{A}V_j = V_{j-1} \tag{16.43}$$

meaning that we can rewrite the problem as

$$V_j = \hat{A}^{-1}V_{j-1} = \hat{A}^{-1}\left(\hat{A}^{-1}V_{j-2}\right) = \cdots = \hat{A}^{-j}V_0. \tag{16.44}$$

If α does not depend on time t, we need to invert a matrix only once. This is an implicit scheme since it relies on determining the vector $u_{i,j-1}$ instead of $u_{i,j+1}$

16.2.3 Program example

Here we present a simple Fortran90 code which solves the following $1+1$-dimensional diffusion problem with $L = 1$

$$\begin{cases} u_{xx} = u_t \\ u(x,0) = sin(\pi x) \\ u(0,t) = u(1,t) = 0 \end{cases}, \tag{16.45}$$

with the exact solution $u(x,t) = e^{-\pi^2 t}sin(\pi x)$.

programs/chap16/program1.f90

```
!  Program to solve the 1-dim heat equation using
!  matrix inversion. The initial conditions are given by
!  u(xmin,t)=u(xmax,t)=0 ang u(x,0) = f(x) (user provided function)
!  Initial conditions are read in by the function initialise
!  such as number of steps in the x-direction, t-direction,
!  xmin and xmax. For xmin = 0 and xmax = 1, the exact solution
!  is u(x,t)= exp(-pi**2*x)sin(pi*x) with f(x) = sin(pi*x)
```

```
!   Note the structure of this module, it contains various
!   subroutines for initialisation of the problem and solution
!   of the PDE with a given initial function for u(x,t)

    MODULE one_dim_heat_equation
       DOUBLE PRECISION, PRIVATE :: xmin, xmax, k
       INTEGER, PRIVATE :: m , ndim

       CONTAINS

       SUBROUTINE initialise
       IMPLICIT NONE
       WRITE(*,*) 'read in number of mesh points in x'
       READ(*,*) ndim
       WRITE(*,*) 'read in xmin and xmax'
       READ(*,*) xmin, xmax
       WRITE(*,*) 'read in number of time steps'
       READ(*,*) m
       WRITE(*,*) 'read in stepsize in t'
       READ(*,*) k

       END SUBROUTINE initialise

       SUBROUTINE solve_1dim_equation(func)
       DOUBLE PRECISION :: h, factor, det, t, pi
       INTEGER :: i, j, l
       DOUBLE PRECISION, ALLOCATABLE, DIMENSION(:,:) :: a
       DOUBLE PRECISION, ALLOCATABLE, DIMENSION(:) :: u, v
       INTERFACE
            DOUBLE PRECISION FUNCTION  func(x)
            IMPLICIT NONE
            DOUBLE PRECISION, INTENT(IN) :: x

            END FUNCTION func
       END INTERFACE
!  define the step size
       h = (xmax-xmin)/FLOAT(ndim+1)
       factor = k/h/h
!  allocate space for the vectors u and v and the matrix a
       ALLOCATE ( a( ndim, ndim) )
       ALLOCATE (u (ndim) , v(ndim) )
       pi = ACOS(-1.)
       DO i=1, ndim
          v(i) = func(pi*i*h)
```

```
        ENDDO
!   write out for t = 0
        t = 0.
        DO  i=1, ndim
            WRITE(6,*) t, i*h, v(i)
        ENDDO
!   setup the matrix to be inverted
        a = 0.  ; u=0.
        DO i=1, ndim - 1
            a(i,i)=1.+2.*factor
            a(i,i+1)=-factor
            a(i+1,i)=-factor
        ENDDO
        a(ndim, ndim) = 1.+2.*factor
!   now invert the matrix
        CALL matinv( a, ndim, det)
        DO i = 1, m
            DO l=1, ndim
                u(l) = DOT_PRODUCT(a(l,:),v(:))
            ENDDO
            v = u
            t = i*k
            DO  j=1, ndim
                WRITE(6,*) t, j*h, v(j)
            ENDDO
        ENDDO
        DEALLOCATE ( a); DEALLOCATE (u, v)

        END SUBROUTINE solve_1dim_equation

    END MODULE one_dim_heat_equation

    PROGRAM heat_eq_1dm
    USE one_dim_heat_equation
    IMPLICIT NONE
        INTERFACE
            DOUBLE PRECISION FUNCTION  function_initial(x)
            IMPLICIT NONE
            DOUBLE PRECISION, INTENT(IN) :: x

            END FUNCTION function_initial
        END INTERFACE
    CALL initialise
    OPEN(UNIT=6, FILE='heat.dat')
    CALL solve_1dim_equation(function_initial)
```

```
    CLOSE(6)

    END PROGRAM heat_eq_1dm


    DOUBLE PRECISION FUNCTION  function_initial(x)
    IMPLICIT NONE
    DOUBLE PRECISION, INTENT(IN) :: x

    function_initial = SIN (x)

    END FUNCTION function_initial
```

16.2.4 Crank-Nicolson scheme

It is possible to combine the implicit and explicit methods in a slightly more general approach. Introducing a parameter θ (the so-called θ-rule) we can set up an equation

$$\frac{\theta}{\Delta x^2}\left(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}\right) + \frac{1-\theta}{\Delta x^2}\left(u_{i+1,j-1} - 2u_{i,j-1} + u_{i-1,j-1}\right) = \frac{1}{\Delta t}\left(u_{i,j} - u_{i,j-1}\right), \tag{16.46}$$

which for $\theta = 0$ yields the forward formula for the first derivative and the explicit scheme, while $\theta = 1$ yields the backward formula and the implicit scheme. These two schemes are called the backward and forward Euler schemes, respectively. For $\theta = 1/2$ we obtain a new scheme after its inventors, Crank and Nicolson. This scheme yields a truncation in time which goes like $O(\Delta t^2)$ and it is stable for all possible combinations of Δt and Δx.

Using our previous definition of $\alpha = \Delta t/\Delta x^2$ we can rewrite the latter equation as

$$-\alpha u_{i-1,j} + (2+2\alpha)\,u_{i,j} - \alpha u_{i+1,j} = \alpha u_{i-1,j-1} + (2-2\alpha)\,u_{i,j-1} + \alpha u_{i+1,j-1}, \tag{16.47}$$

or in matrix-vector form as

$$\left(2\hat{I} + 2\alpha\hat{B}\right)V_j = \left(2\hat{I} - 2\alpha\hat{B}\right)V_{j-1}, \tag{16.48}$$

where the vector V_j is the same as defined in the implicit case while the matrix $\hat{B}$ is

$$\hat{B} = \begin{pmatrix} 2 & -1 & 0 & 0\ldots \\ -1 & 2 & -1 & 0\ldots \\ \ldots & \ldots & \ldots & \ldots \\ 0\ldots & 0\ldots & & 2 \end{pmatrix} \tag{16.49}$$

16.2.5 Non-linear terms and implementation of the Crank-Nicoloson scheme

16.3 Laplace's and Poisson's equations

Laplace's equation reads

$$\nabla^2 u(\mathbf{x}) = u_{xx} + u_{yy} = 0. \tag{16.50}$$

with possible boundary conditions $u(x,y) = g(x,y)$ on the border. There is no time-dependence. Choosing equally many steps in both directions we have a quadratic or rectangular grid, depending on whether we choose equal steps lengths or not in the x and the y directions. Here we set $\Delta x = \Delta y = h$ and obtain a discretized version

$$u_{xx} \approx \frac{u(x+h,y) - 2u(x,y) + u(x-h,y)}{h^2}, \tag{16.51}$$

and

$$u_{yy} \approx \frac{u(x,y+h) - 2u(x,y) + u(x,y-h)}{h^2}, \tag{16.52}$$

which we rewrite as

$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \tag{16.53}$$

and

$$u_{yy} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}, \tag{16.54}$$

which gives when inserted in Laplace's equation

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right]. \tag{16.55}$$

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

$$u_{xx} + u_{yy} = -\rho(\mathbf{x}),$$

and we need only to add a discretized version of $\rho(\mathbf{x})$ resulting in

$$u_{i,j} = \frac{1}{4}\left[u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j}\right] + \rho_{i,j}. \tag{16.56}$$

It is fairly straightforward to extend this equation to the three-dimensional case. Whether we solve Eq. (16.55) or Eq. (16.56), 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. (16.55) requires the knowledge of u at all of the neighbouring points in order to calculate u at any given point.

The way we solve these equations is based on an iterative scheme called the relaxation method. Its steps are rather simple. We start with an initial guess for $u_{i,j}^{(0)}$ where all values are known. To obtain a new solution we solve Eq. (16.55) or Eq. (16.56) in order to obtain a new solution $u_{i,j}^{(1)}$. Most likely this solution will not be a solution to Eq. (16.55). 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.

A simple example may help in visualizing this method. We consider a condensator with parallel plates separated at a distance L resulting in e.g., the voltage differences $u(x,0) = 100sin(2\pi x/L)$ and $u(x,1) = -100sin(2\pi x/L)$. These are our boundary conditions and we ask what is the voltage u between the plates? To solve this problem numerically we provide below a Fortran 90/95 program which solves iteratively Eq. (16.55).

programs/chap16/program2.f90

```
!  Program to solve the 2-dim Laplace equation using iteration.
!  No time-dependence.
!  Initial conditions are read in by the function initialise
!  such as number of steps in the x-direction, y-direction,
!  xmin and xmax, ymin and ymax. Here we employ a square lattice
!  with equal number of steps in x and y directions

!   Note the structure of this module, it contains various
!   subroutines for initialisation of the problem and solution
!   of the PDE with a given initial function for u(x,y)

    MODULE two_dim_laplace_equation
       DOUBLE PRECISION, PRIVATE :: xmin, xmax, ymin, ymax
       INTEGER, PRIVATE :: ndim, iterations
       DOUBLE PRECISION, ALLOCATABLE, DIMENSION(:,:), PRIVATE :: u,
           u_temp
       CONTAINS

!      this function reads in the size of lattice, xmin, xmax,
!      ymin and ymax and the number of iterations

       SUBROUTINE initialise
       IMPLICIT NONE

       WRITE(*,*) 'read in number of mesh points in x and y direction
           '
       READ(*,*) ndim
       WRITE(*,*) 'read in xmin and xmax'
       READ(*,*) xmin, xmax
       WRITE(*,*) 'read in ymin and ymax'
       READ(*,*) ymin, ymax
       WRITE(*,*) 'read in max number of iterations'
       READ(*,*) iterations

       END SUBROUTINE initialise

       SUBROUTINE solve_2dimlaplace_equation(func)
       DOUBLE PRECISION :: h, x, y, pi, length, diff
       INTEGER :: i, j, l

       INTERFACE
            DOUBLE PRECISION FUNCTION  func(argument)
            IMPLICIT NONE
            DOUBLE PRECISION, INTENT(IN) :: argument
```

```
            END FUNCTION func
        END INTERFACE
!   define the step size
        h = (xmax-xmin)/FLOAT(ndim+1)
        length = xmax-xmin
!   allocate space for the vector u and the temporary vector to
!   be upgraded in every iteration
        ALLOCATE ( u( ndim, ndim) )
        ALLOCATE ( u_temp( ndim, ndim) )
        pi = ACOS(-1.)
!   set up of initial conditions at t = 0 and boundary conditions
        u = 0.
        DO i=1, ndim
           x = i*h*pi/length
           u(i,1) = func(x)
           u(i,ndim) = -func(x)
        ENDDO
!   iteration algorithm starts here
        iterations = 0
        DO WHILE ( (iterations <= 20) .OR. ( diff > 0.00001) )
           u_temp = u; diff = 0.
           DO j = 2, ndim - 1
              DO l = 2, ndim -1
                 u(j,l) = 0.25*(u_temp(j+1,l)+u_temp(j-1,l)+ &
                                u_temp(j,l+1)+u_temp(j,l-1))
                 diff = diff + ABS(u_temp(i,j)-u(i,j))
              ENDDO
           ENDDO
           iterations = iterations + 1
           diff = diff/(ndim+1)**2
        ENDDO
!    write out results
        DO j = 1, ndim
           DO l = 1, ndim
              WRITE(6,*) j*h, l*h, u(j,l)
           ENDDO
        ENDDO
        DEALLOCATE (u, u_temp)
        END SUBROUTINE solve_2dimlaplace_equation

    END MODULE two_dim_laplace_equation

    PROGRAM laplace_eq_2dim
    USE two_dim_laplace_equation
```

```
IMPLICIT NONE
    INTERFACE
        DOUBLE PRECISION FUNCTION  function_initial(x)
        IMPLICIT NONE
        DOUBLE PRECISION, INTENT(IN) :: x

        END FUNCTION function_initial
    END INTERFACE

CALL initialise

OPEN(UNIT=6, FILE='laplace.dat')
CALL solve_2dimlaplace_equation(function_initial)
CLOSE(6)

END PROGRAM laplace_eq_2dim


DOUBLE PRECISION FUNCTION  function_initial(x)
IMPLICIT NONE
DOUBLE PRECISION, INTENT(IN) :: x

function_initial = 100*SIN(x)

END FUNCTION function_initial
```

The important part of the algorithm is applied in the function which sets up the two-dimensional Laplace equation. There we have a do-while statement which tests the difference between the temporary vector and the solution $u_{i,j}$. Moreover, we have fixed the number of iterations to be at most 20. This is sufficient for the above problem, but for more general applications you need to test the convergence of the algorithm.

16.4 Wave equation in two dimensions

The $1+1$-dimensional wave equation reads

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial t^2}, \tag{16.57}$$

with $u = u(x,t)$ and we have assumed that we operate with dimensionless variables. Possible boundary and initial conditions with $L = 1$ are

$$\begin{cases} u_{xx} = u_{tt} & x \in [0,1], t > 0 \\ u(x,0) = g(x) & x \in [0,1] \\ u(0,t) = u(1,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = 0 & x \in [0,1] \end{cases}. \tag{16.58}$$

We discretize again time and position,

$$u_{xx} \approx \frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)}{\Delta x^2}, \tag{16.59}$$

and

$$u_{tt} \approx \frac{u(x,t+\Delta t)-2u(x,t)+u(x,t-\Delta t)}{\Delta t^2}, \tag{16.60}$$

which we rewrite as

$$u_{xx} \approx \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{\Delta x^2}, \tag{16.61}$$

and

$$u_{tt} \approx \frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{\Delta t^2}, \tag{16.62}$$

resulting in

$$u_{i,j+1} = 2u_{i,j} - u_{i,j-1} + \frac{\Delta t}{\Delta x^2}\left(u_{i+1,j}-2u_{i,j}+u_{i-1,j}\right). \tag{16.63}$$

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. Wehave thus an explicit finite difference scheme for computing the wave function u. The only additional complication in our case is the initial condition given by the first derivative in time, namely $\partial u/\partial t|_{t=0} = 0$. The discretized version of this first derivative is given by

$$u_t \approx \frac{u(x_i,t_j+\Delta t)-u(x_i,t_j-\Delta t)}{2\Delta t}, \tag{16.64}$$

and at $t =$ it reduces to

$$u_t \approx \frac{u_{i,+1}-u_{i,-1}}{2\Delta t} = 0, \tag{16.65}$$

implying that $u_{i,+1} = u_{i,-1}$. If we insert this condition in Eq. (16.63) we arrive at a special formula for the first time step

$$u_{i,1} = u_{i,0} + \frac{\Delta t}{2\Delta x^2}\left(u_{i+1,0}-2u_{i,0}+u_{i-1,0}\right). \tag{16.66}$$

We need seemingly two different equations, one for the first time step given by Eq. (16.66) and one for all other time-steps given by Eq. (16.63). However, it suffices to use Eq. (16.63) for all times as long as we provide $u(i,-1)$ using

$$u_{i,-1} = u_{i,0} + \frac{\Delta t}{2\Delta x^2}\left(u_{i+1,0}-2u_{i,0}+u_{i-1,0}\right), \tag{16.67}$$

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

$$\begin{cases} t_l = l\Delta t & l \geq 0 \\ x_i = i\Delta x & 1 \leq i \leq n_x+1 \\ y_j = j\Delta y & 1 \leq j \leq n_y+1 \end{cases}, \tag{16.68}$$

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

$$\begin{cases} u_{xx} + u_{yy} = u_{tt} & x, y \in [0,1], t > 0 \\ u(x,y,0) = g(x,y) & x, y \in [0,1] \\ u(0,0,t) = u(1,1,t) = 0 & t > 0 \\ \partial u/\partial t|_{t=0} = 0 & x, y \in [0,1] \end{cases} . \tag{16.69}$$

We have now the following discretized partial derivatives

$$u_{xx} \approx \frac{u^l_{i+1,j} - 2u^l_{i,j} + u^l_{i-1,j}}{h^2}, \tag{16.70}$$

and

$$u_{yy} \approx \frac{u^l_{i,j+1} - 2u^l_{i,j} + u^l_{i,j-1}}{h^2}, \tag{16.71}$$

and

$$u_{tt} \approx \frac{u^{l+1}_{i,j} - 2u^l_{i,j} + u^{l-1}_{i,j}}{\Delta t^2}, \tag{16.72}$$

which we merge into the discretized $2+1$-dimensional wave equation as

$$u^{l+1}_{i,j} = 2u^l_{i,j} - u^{l-1}_{i,j} + \frac{\Delta t}{h^2}\left(u^l_{i+1,j} - 4u^l_{i,j} + u^l_{i-1,j} + u^l_{i,j+1} + u^l_{i,j-1}\right), \tag{16.73}$$

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

$$u^{-1}_{i,j} = u^0_{i,j} + \frac{\Delta t}{2h^2}\left(u^0_{i+1,j} - 4u^0_{i,0} + u^0_{i-1,j} + u^0_{i,j+1} + u^0_{i,j-1}\right), \tag{16.74}$$

in our setup of the initial conditions.

16.4.1 Program for the $2+1$ wave equation and applications

16.5 Inclusion of non-linear terms in the wave equation

Part II

Advanced topics

Chapter 17

Modelling phase transitions

17.1 Methods to classify phase transition

17.1.1 The histogram method

17.1.2 Multi-histogram method

17.2 Renormalization group approach

Chapter 18

Hydrodynamic models

Chapter 19

Diffusion Monte Carlo methods

We discuss implementations and the underlying theory for diffusion Monte Carlo methods.

19.1 Diffusion Monte Carlo

The DMC method is based on rewriting the Schrödinger equation in imaginary time, by defining $\tau = it$. The imaginary time Schrödinger equation is then

$$\frac{\partial \psi}{\partial \tau} = -\widehat{\mathbf{H}}\psi, \tag{19.1}$$

where we have omitted the dependence on τ and the spatial variables in ψ. The wave function ψ is again expanded in eigenstates of the Hamiltonian

$$\psi = \sum_i^{\infty} c_i \phi_i, \tag{19.2}$$

where

$$\widehat{\mathbf{H}}\phi_i = \epsilon_i \phi_i, \tag{19.3}$$

ϵ_i being an eigenstate of $\widehat{\mathbf{H}}$. A formal solution of the imaginary time Schrödinger equation is

$$\psi(\tau_1 + \delta\tau) = e^{-\widehat{\mathbf{H}}\delta\tau}\psi(\tau_1) \tag{19.4}$$

where the state $\psi(\tau_1)$ evolves from an imaginary time τ_1 to a later time $\tau_1 + \delta$. If the initial state $\psi(\tau_1)$ is expanded in energy ordered eigenstates, following Eq. (19.2), then we obtain

$$\psi(\delta\tau) = \sum_i^{\infty} c_i e^{-\epsilon_i \delta\tau}\phi_i. \tag{19.5}$$

Hence any initial state, ψ, that is not orthogonal to the ground state ϕ_0 will evolve to the ground state in the long time limit, that is

$$\lim_{\tau\to\infty} \psi(\delta\tau) = c_0 e^{-\epsilon_0 \tau}\phi_0. \tag{19.6}$$

This derivation shares many formal similarities with that given for the variational principle discussed in the previous sections. However in the DMC method the imaginary time evolution results in excited states decaying exponentially fast, whereas in the VMC method any excited state contributions remain and contribute to the VMC energy.

The DMC method is a realisation of the above derivation in position space. Including the spatial variables as well, the above equation reads

$$\lim_{\tau\to\infty} \psi(\mathbf{R}, \delta\tau) = c_0 e^{-\epsilon_0 \tau} \phi_0(\mathbf{R}). \tag{19.7}$$

By introducing a constant offset to the energy, $E_T = \epsilon_0$, the long-time limit of Eq. (19.7) can be kept finite. If the Hamiltonian is separated into the kinetic energy and potential terms, the imaginary time Schrödinger equation, takes on a form similar to a diffusion equation, namely

$$-\frac{\partial\psi(\mathbf{R},\tau)}{\partial\tau} = \left[\sum_i^N -\frac{1}{2}\nabla_i^2\psi(\mathbf{R},\tau)\right] + (V(\mathbf{R}) - E_T)\psi(\mathbf{R},\tau). \tag{19.8}$$

This equation is a diffusion equation where the wave function ψ may be interpreted as the density of diffusing particles (or "walkers"), and the term $V(\mathbf{R})-E_T$ is a rate term describing a potential-dependent increase or decrease in the particle density. The above equation may be transformed into a form suitable for Monte Carlo methods, but this leads to a very inefficient algorithm. The potential $V(\mathbf{R})$ is unbounded in coulombic systems and hence the rate term $V(\mathbf{R}) - E_T$ can diverge. Large fluctuations in the particle density then result and give impractically large statistical errors.

These fluctuations may be substantially reduced by the incorporation of importance sampling in the algorithm. Importance sampling is essential for DMC methods, if the simulation is to be efficient. A trial or guiding wave function $\psi_T(\mathbf{R})$, which closely approximates the ground state wave function is introduced. This is where typically the VMC result would enter, see also discussion below A new distribution is defined as

$$f(\mathbf{R},\tau) = \psi_T(\mathbf{R})\psi(\mathbf{R},\tau), \tag{19.9}$$

which is also a solution of the Schrödinger equation when $\psi(\mathbf{R},\tau)$ is a solution. Eq. (19.8) consequently modified to

$$\frac{\partial f(\mathbf{R},\tau)}{\partial\tau} = \frac{1}{2}\nabla\left[\nabla - F(\mathbf{R})\right] f(\mathbf{R},\tau) + (E_L(\mathbf{R}) - E_T) f(\mathbf{R},\tau). \tag{19.10}$$

In this equation we have introduced the so-called force-term F, given by

$$F(\mathbf{R}) = \frac{2\nabla\psi_T(\mathbf{R})}{\psi_T(\mathbf{R})}, \tag{19.11}$$

and is commonly referred to as the "quantum force". The local energy E_L is defined as previously

$$E_L\mathbf{R}) = -\frac{1}{\psi_T(\mathbf{R})}\frac{\nabla^2\psi_T(\mathbf{R})}{2} + V(\mathbf{R})\psi_T(\mathbf{R}), \tag{19.12}$$

and is computed, as in the VMC method, with respect to the trial wave function.

We can give the following interpretation to Eq. (19.10). The right hand side of the importance sampled DMC equation consists, from left to right, of diffusion, drift and rate terms. The problematic potential dependent rate term of the non-importance sampled method is replaced by a term dependent on the difference between the local energy of the guiding wave function and the trial energy. The trial energy is initially chosen to be the VMC energy of the trial wave function, and is updated as the simulation progresses. Use of an optimised trial function minimises the difference between the local and trial energies, and hence minimises fluctuations in the distribution f . A wave function optimised using VMC is ideal for this purpose, and in practice VMC provides the best method for obtaining wave functions that accurately approximate ground state wave functions locally. The trial wave function may be also constructed to minimise the number of divergences in , unlike the non-importance sampled method where divergences in the coulomb interactions are always present.

To be of use however, the importance sampled DMC method of Eq. (19.10) must be transformed into a form suitable for Monte Carlo integration. The transformation is more complex than for VMC, which simply required the insertion of the factor

$$P(\mathbf{R}) = \frac{|\psi_T(\mathbf{R})|^2}{\int |\psi_T(\mathbf{R})|^2 \, d\mathbf{R}}.$$

into the conventional formulas for quantum mechanical expectation values.

A Green's function $G(\mathbf{R}', \mathbf{R}, \tau)$ that is a solution of Eq. (19.10) is desired, i.e., a spatial representation of the imaginary time propagator, $e^{-\tau(\hat{\mathbf{H}}-E_T)}$. One can show that the Green's function of the diffusion equation, by factorising the propagator into branching and diffusion parts, can be written as

$$G(\mathbf{R}', \mathbf{R}, \tau) \sim e^{-\left((\mathbf{R}-\mathbf{R}'-\tau F(\mathbf{R}')^2/2\tau\right)}. \tag{19.13}$$

19.2 Other Quantum Monte Carlo techniques and systems

In our discussion, the emphasis has been on variational methods, since they are rather intuitive and one can simulate physical systems with rather simple trial wave functions. We have also not dealt with problems arising in many-fermion systems, where both the sign of the wave function in the diffusion Monte Carlo is crucial and the evaluation of the Slater determinant is computationally involved. Furthermore, techniques to improve the variance have also not been discussed. We defer these topics, together with a discussion of other Monte Carlo methods such as Green's function Monte Carlo, path integral Monte Carlo and Lattice methods to a more advanced course on computational Physics.

Chapter 20

Finite element method

Chapter 21

Stochastic methods in Finance

Chapter 22

Quantum information theory and quantum algorithms

Bibliography

[1] W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, Numerical Recipes in C/Fortran, The art of scientific computing, (Cambridge, 1992).

[2] W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, Numerical Recipes in Fortran 90, The art of scientific computing, (Cambridge, 1996).

[3] J. Stoer and R. Bulirsch, Introduction to Numerical Analysis, (Springer, 1993).

[4] S.E. Koonin and D. Meredith, Computational Physics, (Addison Wesley, 1990).

[5] F.S. Acton, Numerical Methods that work, (Harper and Row, 1970).

[6] R.H. Landau and M.J. Paez, Computational Physics (Wiley, 1997).

[7] E.W. Schmid, G. Spitz and W. Lösch, Theoretische Physik mit dem Personal Computer, (Springer, 1987).

[8] J.M. Thijssen, Computational Physics, (Springer, 1999).

[9] R. Guardiola, E. Higon and J. Ros, Metodes Numèrics per a la Física, (Universitat de Valencia, 1997).

[10] H. Gould and J. Tobochnik, An Introduction to Computer Simulation Methods: Applications to Physical Systems, (Addison-Wesley, 1996).

[11] M. Metcalf and J. Reid, The F90 Programming Language, (Oxford University Press, 1996).