Computational Physics, an Introduction

Morten Hjorth-Jensen

February 13, 2017

Contents

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

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

This set of lecture notes serves the scope of presenting to you and train you in an algorithmic approach to problems in the sciences, represented here by the unity of three disciplines, physics, mathematics and informatics. This trinity outlines the emerging field of computational physics.

Our insight in a physical system, combined with numerical mathematics gives us the rules for setting up an algorithm, viz. a set of rules for solving a particular problem. Our understanding of the physical system under study is obviously gauged by the natural laws at play, the initial conditions, boundary conditions and other external constraints which influence the given system. Having spelled out the physics, for example in the form of a set of coupled partial differential equations, we need efficient numerical methods in order to set up the final algorithm. This algorithm is in turn coded into a computer program and executed on available computing facilities. To develop such an algorithmic approach, you will be exposed to several physics cases, spanning from the classical pendulum to quantum mechanical systems. We will also present some of the most popular algorithms from numerical mathematics used to solve a plethora of problems in the sciences. Finally we will codify these algorithms using some of the most widely used programming languages, presently C, C++ and Fortran and its most recent standard Fortran 2008[1]. However, a high-level and fully object-oriented language like Python is now emerging as a good alternative although C++ and Fortran still outperform Python when it comes to computational speed. In this text we offer an approach where one can write all programs in C/C++ or Fortran. We will also show you how to develop large programs in Python interfacing C++ and/or Fortran functions for those parts of the program which are CPU intensive. Such an approach allows you to structure the flow of data in a high-level language like Python while tasks of a mere repetitive and CPU intensive nature are left to low-level languages like C++ or Fortran. Python allows you also to smoothly interface your program with other software, such as plotting programs or operating

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

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

Computer simulations are nowadays an integral part of contemporary basic and applied research in the sciences. Computation is becoming as important as theory and experiment. In physics, computational physics, theoretical physics and experimental physics are all equally important in our daily research and studies of physical systems. Physics is the unity of theory, experiment and computation[2]. Moreover, the ability "to compute" forms part of the essential repertoire of research scientists. Several new fields within computational science have emerged and strengthened their positions in the last years, such as computational materials science, bioinformatics, computational mathematics and mechanics, computational chemistry and physics and so forth, just to mention a few. These fields underscore the importance of simulations as a means to gain novel insights into physical systems, especially for those cases where no analytical solutions can be found or an experiment is too complicated or expensive to carry out. To be able to simulate large quantal systems with many degrees of freedom such as strongly interacting electrons in a quantum dot will be of great importance for future directions in novel fields like nano-techonology. This ability often combines knowledge from many different subjects, in our case essentially from the physical sciences, numerical mathematics, computing languages, topics from high-performace computing and some knowledge of computers.

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

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

were using the trapezoidal rule to compute integrals. With Gaussian quadrature this dimensionality was easily reduced to matrix problems of the size of $10^2 \times 10^2$, with much better precision.

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

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

Computational speed is not the only reason for this choice of programming languages. Another important reason is that we feel that at a certain stage one needs to have some insights into the algorithm used, its stability conditions, possible pitfalls like loss of precision, ranges of applicability, the possibility to improve the algorithm and taylor it to special purposes etc etc. One of our major aims here is to present to you what we would dub 'the algorithmic approach', a set of rules for doing mathematics or a precise description of how to solve a problem. To device an algorithm and thereafter write a code for solving physics problems is a marvelous way of gaining insight into complicated physical systems. The algorithm you end up writing reflects in essentially all cases your own understanding of the physics and the mathematics (the way you express yourself) of the problem. We do therefore devote quite some space to the algorithms behind various functions presented in the text. Especially, insight into how errors propagate and how to avoid them is a topic we would like you to pay special attention to. Only then can you avoid problems like underflow, overflow and loss of precision. Such a control is not always achievable with interpreted languages and canned functions where the underlying algorithm and/or code is not easily accesible. Although we will at various stages recommend the use of library routines for say linear algebra[3], our belief is that one should understand what the given function does, at least to have a mere idea. With such a starting point, we strongly believe that it can be easier to develope more complicated programs on your own using Fortran, C++ or Python.

We have several other aims as well, namely:

- We would like to give you an opportunity to gain a deeper understanding of the physics you have learned in other courses. In most courses one is normally confronted with simple systems which provide exact solutions and mimic to a certain extent the realistic cases. Many are however the comments like 'why can't we do something else than the particle in a box potential?'. In several of the projects we hope to present some more 'realistic' cases to solve by various numerical methods. This also means that we wish to give examples of how physics can be applied in a much broader context than it is discussed in the traditional physics undergraduate curriculum.

- To encourage you to "discover" physics in a way similar to how researchers learn in the context of research.

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

- Hopefully also to introduce numerical methods and new areas of physics that can be studied with the methods discussed.
- To teach structured programming in the context of doing science.
- The projects we propose are meant to mimic to a certain extent the situation encountered during a thesis or project work. You will tipically have at your disposal 2-3 weeks to solve numerically a given project. In so doing you may need to do a literature study as well. Finally, we would like you to write a report for every project.

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

Needless to say, these lecture notes are upgraded continuously, from typos to new input. And we do always benefit from your comments, suggestions and ideas for making these notes better. It's through the scientific discourse and critics we advance. Moreover, I have benefitted immensely from many discussions with fellow colleagues and students. In particular I must mention Hans Petter Langtangen, Anders Malthe-Sørenssen, Knut Mørken and Øyvind Ryan, whose input during the last fifteen years has considerably improved these lecture notes. Furthermore, the time we have spent and keep spending together on the Computing in Science Education project at the University, is just marvelous. Thanks so much. Concerning the Computing in Science Education initiative, you can read more at http://www.mn.uio.no/english/about/collaboration/cse/.

Finally, I would like to add a petit note on referencing. These notes have evolved over many years and the idea is that they should end up in the format of a web-based learning environment for doing computational science. It will be fully free and hopefully represent a much more efficient way of conveying teaching material than traditional textbooks. I have not yet settled on a specific format, so any input is welcome. At present however, it is very easy for me to upgrade and improve the material on say a yearly basis, from simple typos to adding new material. When accessing the web page of the course, you will have noticed that you can obtain all source files for the programs discussed in the text. Many people have thus written to me about how they should properly reference this material and whether they can freely use it. My answer is rather simple. You are encouraged to use these codes, modify them, include them in publications, thesis work, your lectures etc. As long as your use is part of the dialectics of science you can use this material freely. However, since many weekends have elapsed in writing several of these programs, testing them, sweating over bugs, swearing in front of a f*@?%g code which didn't compile properly ten minutes before monday morning's eight o'clock lecture etc etc, I would dearly appreciate in case you find these codes of any use, to reference them properly. That can be done in a simple way, refer to M. Hjorth-Jensen, *Computational Physics*, University of Oslo (2013). The weblink to the course should also be included. Hope it is not too much to ask for. Enjoy!

About the Author

I am a theoretical physicist with a strong interest in computational physics and many-body theory in general, and the nuclear many-body problem and nuclear structure problems in particular. This means that I study various methods for solving either Schrödinger's equation or Dirac's equation for many interacting particles, spanning from algorithmic aspects to the mathematical properties of such methods. The latter also leads to a strong interest in computational physics as well as computational aspects of quantum mechanical methods. Since 2012, I share my time equally between Michigan State University in the US and the University of Oslo, Norway.

Part I

Introduction to Programming

Chapter 1

Introduction

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

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

The text gives a survey over some of the most used methods in computational physics and each chapter ends with one or more applications to realistic systems, from the structure of a neutron star to the description of quantum mechanical systems through Monte-Carlo methods. Among the algorithms we discuss, are some of the top algorithms in computational science. In recent surveys by Dongarra and Sullivan [3] and Cipra [2], the list over the ten top algorithms of the 20th century include

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

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

3. Krylov Subspace Iteration method for large eigenvalue problems in particular, developed by Magnus Hestenes, Eduard Stiefel, and Cornelius Lanczos, discussed in chapter 7.

4. The Householder matrix decomposition, developed by Alston Householder and discussed in chapter 7.

5. The Fortran compiler, developed by a team lead by John Backus, codes used throughout this text.

6. The QR algorithm for eigenvalue calculation, developed by Joe Francis, discussed in chapter 7

7. The Quicksort algorithm, developed by Anthony Hoare.

8. Fast Fourier Transform, developed by James Cooley and John Tukey.

9. The Integer Relation Detection Algorithm, developed by Helaman Ferguson and Rodney

10. The fast Multipole algorithm, developed by Leslie Greengard and Vladimir Rokhlin; (to calculate gravitational forces in an N-body problem normally requires N^2 calculations. The fast multipole method uses order N calculations, by approximating the effects of groups of distant particles using multipole expansions)

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

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

Monte Carlo methods and diffusion Monte Carlo methods. In chapter 7 we deal with eigensystems and applications to e.g., the Schrödinger equation rewritten as a matrix diagonalization problem. Problems from scattering theory are also discussed, together with the most used solution methods for systems of linear equations. Finally, we discuss various methods for solving differential equations and partial differential equations in chapters **??**-**??** with examples ranging from harmonic oscillations, equations for heat conduction and the time dependent Schrödinger equation. The emphasis is on various finite difference methods.

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

There are several texts on computational physics on the market, see for example Refs. [5, 6, 7, 9, 11, 12, 18, 22], ranging from introductory ones to more advanced ones. Most of these texts treat however in a rather cavalier way the mathematics behind the various numerical methods. We've also succumbed to this approach, mainly due to the following reasons: several of the methods discussed are rather involved, and would thus require at least a one-semester course for an introduction. In so doing, little time would be left for problems and computation. This course is a compromise between three disciplines, numerical methods, problems from the physical sciences and computation. To achieve such a synthesis, we will have to relax our presentation in order to avoid lengthy and gory mathematical expositions. You should also keep in mind that computational physics and science in more general terms consist of the combination of several fields and crafts with the aim of finding solution strategies for complicated problems. However, where we do indulge in presenting more formalism, we have borrowed heavily from several texts on mathematical analysis.

1.1 Choice of programming language

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

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

Fortran (FORmula TRANslation) was introduced in 1957 and remains in many scientific computing environments the language of choice. The latest standard, see Refs. [15, 16, 17, 19], includes extensions that are familiar to users of C++. Some of the most important features of Fortran include recursive subroutines, dynamic storage allocation and pointers, user defined data structures, modules, and the ability to manipulate entire arrays. However, there are several good reasons for choosing C++ as programming language for scientific and engineering problems. Here are some:

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

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

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

- In the last years there has been an enormous effort towards developing numerical libraries for C++. Numerous tools (numerical libraries such as MPI[8, 10, 20]) are written in C++ and interfacing them requires knowledge of C++. Most C++ and Fortran compilers compare fairly well when it comes to speed and numerical efficiency. Although Fortran 77 and C are regarded as slightly faster than C++ or Fortran, compiler improvements during the last few years have diminshed such differences. The Java numerics project has lost some of its steam recently, and Java is therefore normally slower than C++ or Fortran.

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

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

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

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

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

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

Future plans

Since our undergraduate curriculum has changed considerably from the beginning of the fall semester of 2007, with the introduction of Python as programming language, the content of this course will change accordingly from the fall semester 2009. C++ and Fortran will then coexist with Python and students can choose between these three programming languages. The emphasis in the text will be on C++ programming, but how to interface C++ or Fortran programs with Python codes will also be discussed. Tools like Cython (or SWIG) are highly recommended, see for example the Cython link at http://cython.org.

1.2 Designing programs

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

In writing a program for a specific algorithm (a set of rules for doing mathematics or a precise description of how to solve a problem), it is obvious that different programmers will apply different styles, ranging from barely readable [2] (even for the programmer) to well documented codes which can be used and extended upon

[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.

by others in e.g., a project. The lack of readability of a program leads in many cases to credibility problems, difficulty in letting others extend the codes or remembering oneself what a certain statement means, problems in spotting errors, not always easy to implement on other machines, and so forth. Although you should feel free to follow your own rules, we would like to focus certain suggestions which may improve a program. What follows here is a list of our recommendations (or biases/prejudices).

First about designing a program.

- Before writing a single line, have the algorithm clarified and understood. It is crucial to have a logical structure of e.g., the flow and organization of data before one starts writing.

- Always try to choose the simplest algorithm. Computational speed can be improved upon later.

- Try to write a as clear program as possible. Such programs are easier to debug, and although it may take more time, in the long run it may save you time. If you collaborate with other people, it reduces spending time on debugging and trying to understand what the codes do. A clear program will also allow you to remember better what the program really does!

- Implement a working code with emphasis on design for extensions, maintenance etc. Focus on the design of your code in the beginning and don't think too much about efficiency before you have a thoroughly debugged and verified program. A rule of thumb is the so-called $80-20$ rule, 80 % of the CPU time is spent in 20 % of the code and you will experience that typically only a small part of your code is responsible for most of the CPU expenditure. Therefore, spend most of your time in devising a good algorithm.

- The planning of the program should be from top down to bottom, trying to keep the flow as linear as possible. Avoid jumping back and forth in the program. First you need to arrange the major tasks to be achieved. Then try to break the major tasks into subtasks. These can be represented by functions or subprograms. They should accomplish limited tasks and as far as possible be independent of each other. That will allow you to use them in other programs as well.

- Try always to find some cases where an analytical solution exists or where simple test cases can be applied. If possible, devise different algorithms for solving the same problem. If you get the same answers, you may have coded things correctly or made the same error twice.

- When you have a working code, you should start thinking of the efficiency. Analyze the efficiency with a tool (profiler) to predict the CPU-intensive parts.

Attack then the CPU-intensive parts after the program reproduces benchmark results.

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

- Avoid lists, sets etc., when arrays can be used without too much waste of memory. Avoid also calls to functions in the innermost loop since that produces an overhead in the call.
- Heavy computation with small objects might be inefficient, e.g., vector of class complex objects
- Avoid small virtual functions (unless they end up in more than (say) 5 multiplications)
- Save object-oriented constructs for the top level of your code.
- Use taylored library functions for various operations, if possible.
- Reduce pointer-to-pointer-to....-pointer links inside loops.
- Avoid implicit type conversion, use rather the explicit keyword when declaring constructors in C++.
- Never return (copy) of an object from a function, since this normally implies a hidden allocation.

Finally, here are some of our favorite approaches to code writing.

- Use always the standard ANSI version of the programming language. Avoid local dialects if you wish to port your code to other machines.
- Add always comments to describe what a program or subprogram does. Comment lines help you remember what you did e.g., one month ago.
- Declare all variables. Avoid totally the `IMPLICIT` statement in Fortran. The program will be more readable and help you find errors when compiling.
- Do not use `GOTO` structures in Fortran. Although all varieties of spaghetti are great culinaric temptations, spaghetti-like Fortran with many `GOTO` statements is to be avoided. Extensive amounts of time may be wasted on decoding other authors' programs.

- When you name variables, use easily understandable names. Avoid `v1` when you can use `speed_of_light` . Associatives names make it easier to understand what a specific subprogram does.
- Use compiler options to test program details and if possible also different compilers. They make errors too.
- Writing codes in C++ and Fortran may often lead to segmentation faults. This means in most cases that we are trying to access elements of an array which are not available. When developing a code it is then useful to compile with debugging options. The use of debuggers and profiling tools is something we highly recommend during the development of a program.

For more detailed texts on C++ programming in engineering and science are the books by Flowers [4] and Barton and Nackman [1]. The classic text on C++ programming is the book of Bjarne Stoustrup [21]. The Fortran 95 standard is well documented in Refs. [15, 16, 19] while the new details of Fortran 2003 and 2008 can be found in Ref. [17**?**]. The reader should note that this is not a text on C++ or Fortran. It is therefore important than one tries to find additional literature on these programming languages. Good Python texts on scientific computing are [13, 14].

Bibliography

[1] J.J. Barton and L.R. Nackman. *Scientific and Engineering C++*. Addison Wesley, 1994.

[2] B. Cipra. *SIAM News*, 33:1, 2000.

[3] J. Dongarra and F. Sullivan. *Computing in Science and Engineering*, 2:22, 2000.

[4] B.H. Flowers. *An Introduction to Numerical Methods in C++*. Oxford University Press, 2000.

[5] J. Gibbs. *Computational Physics*. World Scientific, 1994.

[6] B. Giordano and H. Nakanishi. *Computational Physics*. Preston, 2005.

[7] H. Gould and J. Tobochnik. *An Introduction to Computer Simulation Methods: Applications to Physical Systems*. Addison-Wesley, 1996.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1999.

[9] R. Guardiola, E. Higon, and J. Ros. *Metodes Numèrics per a la Física*. Universitat de Valencia, 1997.

[10] G. E. Karniadakis and R. M. Kirby II. *Parallel scientific computing in C++ and MPI*. Cambridge, 2005.

[11] S.E. Koonin and D. Meredith. *Computational Physics*. Addison Wesley, 1990.

[12] R.H. Landau and M.J. Paez. *Computational Physics*. Wiley, 1997.

[13] H.P. Langtangen. *Python Scripting for Computational Science*. Springer, 2006.

[14] H.P. Langtangen. *Introduction to Computer Programming; A Python-based approach for Computational Science*. Springer Verlag, 2009.

[15] A.C. Marshall. *Fortran 90 Programming*. University of Liverpool, 1995.

[16] M. Metcalf and J. Reid. *The F90 Programming Language*. Oxford University Press, 1996.

[17] J. Reid. The new features of fortran 2003. Technical report, ISO working group on Fortran, 2007.

[18] E.W. Schmid, G. Spitz, and W. Lösch. *Theoretische Physik mit dem Personal Computer*. Springer Verlag, 1987.

[19] B.T. Smith, J.C. Adams, W.S. Brainerd, and J.L. Wagener. *Fortran 95 Handbook*. MIT press, 1997.

[20] M. Snir, S. Otto, S. Huss-Ledermann, D. Walker, and J. Dongarra. *MPI, the Complete Reference, Vols I and II*. The MIT Press, 1998.

[21] B. Stoustrup. *The C++ Programming Language*. Pearson, 1997.

[22] J.M. Thijssen. *Computational Physics*. Cambridge, 1999.

Chapter 2

Introduction to C++ and Fortran

2.1 Getting Started

In 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 for example `INTEGER (KIND=2)` for Fortran [1] or `short int/int` in C++, the programmer selects a particular date type with 2 bytes (16 bits) for every item of the class `INTEGER (KIND=2)` or `int`. Intrinsic types come in two classes, numerical (like integer, real or complex) and non-numeric (as logical and character). The general form for declaring variables is `data type name of variable` and Table 2.1 lists the standard variable declarations of C++ and Fortran (note well that there be may compiler and machine differences from the table below). An important aspect when declaring variables is their region of validity. Inside a function we define a a variable through the expression `int var` or `INTEGER :: var` . The question is whether this variable is available in other functions as well, moreover where is `var` initialized and finally, if we call the function where it is declared, is the value conserved from one call to the other?

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

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++ does.

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

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

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

In Table 2.1 we show a list of some of the most used language statements in Fortran and C++.

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

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

We will discuss the above declaration `complex<double> x,y;` in more detail in chapter 3.

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

Table 2.2: Elements of programming syntax.

2.1.1 Scientific hello world

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

Click here to view code

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

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

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

We call three functions `atof, sin, printf` and these are declared in three different header files. The main program is a function called main with a return value set to an integer, returning 0 if success. The operating system stores the return value, and other programs/utilities can check whether the execution was successful or not. The command-line arguments are transferred to the main function through the statement

```
int main (int argc, char* argv[])
```

The integer `argc` stands for the number of command-line arguments, set to one in our case, while `argv` is a vector of strings containing the command-line arguments with `argv[0]` containing the name of the program and `argv[1]`, `argv[2]`, ... are the command-line args, i.e., the number of lines of input to the program.

This means that we would run the programs as `mhjensen@compphys:./myprogram.exe 0.3`. The name of the program enters `argv[0]` while the text string 0.2 enters `argv[1]`. Here we define a floating point variable, see also below, through the keywords `float` for single precision real numbers and `double` for double precision. The function `atof` transforms a text `(argv[1])` to a float. The sine function is declared in math.h, a library which is not automatically included and needs to be linked when computing an executable file.

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

In C++ this program can be written as

Click here to view code

```
// A comment line begins like this in C++ programs
using namespace std;
#include <iostream>
#include <cstdlib>
#include <cmath>
int main (int argc, char* argv[])
{
// convert the text argv[1] to double using atof:
  double r = atof(argv[1]);
  double s = sin(r);
  cout << "Hello, World! sin(" << r << ")=" << s << endl;
// success
  return 0;
}
```

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

Click here to view code

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

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

Click here to view code

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

Here we test that we have more than one argument. If not, the program stops and writes to screen an error message. Observe also that we have included the mathematics library via the `#include <cmath>` declaration.

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

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

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

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

The corresponding Fortran code is

Click here to view code

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

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

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

We call the Fortran compiler (using free format) through

```
gfortran -c -free myprogram.f90
gfortran -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= gfortran
PROG= myprogram

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

# whereas here we create the object file

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

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

Click here to view code

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

import sys, math
```

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

where we have used a formatted printout with scientific notation. In Python we do not need to declare variables. Mathematical functions like the $\sin$ function are imported from the *math* module. For further references to Python and its syntax, we recommend the text of Hans Petter Langtangen [2]. The corresponding codes in Python are available at the webpage of the course. All programs are listed as a directory tree beginning with programs/chapterxx. Each chapter has in turn three directories, one for C++, one for Fortran and finally one for Python codes. The Fortran codes in this chapter can be found in the directory programs/chapter02/Fortran.

2.2 Representation of Integer Numbers

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

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

An example of a program declaration is

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

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

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

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

Again one bit is used to store the sign of the variable age_of_fossile and the other 31 bits are used to store the number which then may range from zero to $2^{31} - 1 =$

2.147.483.647. In order to give you a feeling how integer numbers are represented in the computer, think first of the decimal representation of the number 417

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

which in binary representation becomes

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

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

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

In binary notation we have thus

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

since we have

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

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

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

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

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

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

Click here to view code

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

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

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

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

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

For this number 2 significant bits would be enough.

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

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

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

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

http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program3.cpp

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

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

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

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

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

2.2.1 Fortran codes

The corresponding Fortran code is

http://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program2.f90

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

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

END PROGRAM binary_integer
```

and

http://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program3.f90

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

END PROGRAM integer_exp
```

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

2.3 Real Numbers and Numerical Precision

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

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

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

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

Suppose we wish to evaluate the function

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

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

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

If we now choose $x = 0.006$ (in radians) our choice of precision results in

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

and

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

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

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

while the second expression results in

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

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

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

2.3.1 Representation of real numbers

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

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

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

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

and a real non-zero number could be generalized as

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

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

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

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

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

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

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

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

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

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

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

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

As an example, consider the 32 bits binary number

$$(10111110111101000000000000000000)_2,$$

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

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

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

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

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

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

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

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

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

2.3.2 Machine numbers

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

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

or in a more compact form

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

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

In most cases there will not be an exact machine representation of the number x. Our number will be placed between two exact 32 bits machine numbers x_- and x_+. Following the discussion of Kincaid and Cheney [1] these numbers are given by

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

and

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

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

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

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

If we compare the relative errors

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

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

We define then the relative error for x as

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

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

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

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

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

we can write the previous inequality

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

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

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

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

or

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

meaning that

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

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

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

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

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

whereas the relative error is

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

The above subraction is thus

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

yielding

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

An interesting question is then how many significant binary bits are lost in a subtraction $a = b - c$ when we have $b \approx c$. The loss of precision theorem for a subtraction $a = b - c$ states that [1]: *if b and c are positive normalized floating-point binary machine numbers with $b > c$ and*

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

then at most r and at least s significant binary bits are lost in the subtraction $b - c$. For a proof of this statement, see for example Ref. [1].

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

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

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

2.4.1 Algorithms for e^{-x}

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

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

1. by simply coding
$$e^{-x} = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!}$$

2. or to employ a recursion relation for
$$e^{-x} = \sum_{n=0}^{\infty} s_n = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!}$$
using
$$s_n = -s_{n-1} \frac{x}{n},$$

3. or to first calculate
$$\exp x = \sum_{n=0}^{\infty} s_n$$

and thereafter taking the inverse

$$e^{-x} = \frac{1}{\exp x}$$

Below we have included a small program which calculates

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

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

http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program4.cpp

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

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


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

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

There are several features to be noted[2]. 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}.

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

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

The overflow problem can be dealt with via a recurrence formula[3] for the terms

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

[3]Recurrence formulae, in various disguises, either as ways to represent series or continued frac-

in the sum, so that we avoid calculating factorials. A simple recurrence formula for our equation

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

is to note that

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

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

http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program5.cpp

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

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

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

In this case, we do not get the overflow problem, as can be seen from the large number of terms. Our results do however not make much sense for larger values of x. Decreasing the truncation test will not help! (try it). This is a much more serious problem.

In order better to understand this problem, let us consider the case of $x = 20$, which already differs largely from the exact result. Writing out each term in the summation, we obtain the largest term in the sum appears at $n = 19$, with a value

tions, are among the most commonly used forms for function approximation. Examples are Bessel functions, Hermite and Laguerre polynomials, discussed for example in chapter 5.

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

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

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

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

2.4.2 Fortran codes

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

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

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

http://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program4.f90

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

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

MODULE functions

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

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

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

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

END PROGRAM exp_prog
```

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

http://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program5.f90

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

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

  ! loop over x-values, no floats as loop variables
  DO loop_over_x=0, 100, 10
     x=loop_over_x
     ! initialize the EXP sum
     final_sum=1.0 ; term=1.0 ; n = 1
     DO WHILE ( ABS(term) > truncation)
        term = -term*x/FLOAT(n)
        final_sum=final_sum+term
        n=n+1
     ENDDO
     ! write the argument x, the exact value, the computed value and n
     WRITE(*,*) x ,EXP(-x), final_sum, n
  ENDDO

END PROGRAM improved_exp
```

2.4.3 Further examples

Summing $1/n$

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

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

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

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

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

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

The standard algorithm for the standard deviation

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

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

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

while

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

Let us now assume that

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

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

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

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

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

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

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

while the exact answer is

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

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 $i = 64$

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

and we have no potential for loss of precision.

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

http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/program6.cpp

```
// program to calculate the mean and standard deviation of
// a user created data set stored in array x[]
using namespace std;
#include <iostream>
int main()
{
   int    i;
   float  sum, sumsq2, xbar, sigma1, sigma2;
   // array declaration with fixed dimension
   float x[127];
   // initialise the data set
   for ( i=0; i < 127 ; i++){
      x[i] = i + 100000.;
   }
   // The variable sum is just the sum over all elements
   // The variable sumsq2 is the sum over x^2
   sum=0.;
   sumsq2=0.;
   // Now we use the text book algorithm
   for ( i=0; i < 127; i++){
      sum += x[i];
      sumsq2 += pow((double) x[i],2.);
   }
   // calculate the average and sigma
   xbar=sum/127.;
   sigma1=sqrt((sumsq2-sum*xbar)/126.);
   /*
   ** Here comes the second algorithm where we evaluate
   ** separately first the average and thereafter the
   ** sum which defines the standard deviation. The average
   ** has already been evaluated through xbar
   */
   sumsq2=0.;
   for ( i=0; i < 127; i++){
     sumsq2 += pow( (double) (x[i]-xbar),2.);
   }
   sigma2=sqrt(sumsq2/126.);
   cout << "xbar = `` << xbar << ``sigma1 = `` << sigma1 << ``sigma2 = `` << sigma2;
   cout << endl;
   return 0;
}// End: function main()
```

The corresponding Fortran program is given below.

http://folk.uio.no/mhjensen/compphys/programs/chapter02/Fortran/program6.f90

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

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

    sumsq2 = sumsq2+x(i)**2
  ENDDO
  !    average
  xbar=sum/127.
  sigma1=SQRT((sumsq2-sum*xbar)/126.)
  !    second algorithm to evaluate the standard deviation
  sumsq2=0.
  DO i=1, 127
    sumsq2=sumsq2+(x(i)-xbar)**2
  ENDDO
  sigma2=SQRT(sumsq2/126.)
  WRITE(*,*) xbar, sigma1, sigma2

END PROGRAM standard_deviation
```

2.5 Additional Features of C++ and Fortran

2.5.1 Operators in C++

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

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

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

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

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

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

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

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

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

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

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

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

Table 2.7: List of bitwise operations.

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

Table 2.8: C++ specific expressions.

2.5.2 Pointers and arrays in C++.

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

`int name`	defines an integer variable called `name`. It is given an address in memory where we can store an integer number.
`&name`	is the address of a specific place in memory where the integer `name` is stored. Placing the operator & in front of a variable yields its address in memory.
`int *pointer`	defines an integer pointer and reserves a location in memory for this specific variable The content of this location is viewed as the address of another place in memory where we have stored an integer.

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

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

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

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

```
using namespace std;
main()
  {
```

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

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

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

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

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

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

http://folk.uio.no/mhjensen/compphys/programs/chapter02/cpp/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
```

2.5.3 Macros in C++

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

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

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

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

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

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

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

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

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

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

In C++ we would replace such function definition by employing so-called `inline` functions. The above functions could then read

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

where we have defined the transferred variables to be of type `double`. The functions also return a `double` type. These functions could easily be generalized through the use of classes and templates, see chapter 6, to return whather types of real, complex or integer variables.

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

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

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

2.5.4 Structures in C++ and TYPE in Fortran

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

However, C++ and Fortran offer other ways as well by which we can organize our data in a more transparent and reusable way. One of these options is through the `struct` declaration of C++, or the correspondingly similar `TYPE` in Fortran. The latter data type will also be discussed in chapter 6.

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

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

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

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

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

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

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

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

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

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

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

2.6 Reading and writing to file

Furthermore, we will use this section to introduce three important C++-programming features, namely reading and writing to a file, call by reference and call by value, and dynamic memory allocation. We are also going to split the tasks performed by the program into subtasks. We define one function which reads in the input data, one which calculates the second derivative and a final function which writes the results to file.

Let us look at a simple case first, the use of `printf` and `scanf`. If we wish to print a variable defined as `double speed_of_sound;` we could for example write

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

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

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

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

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

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

Consider now the following program

```
1  using namespace std;
2  # include <iostream>
3  // begin main function
4  int main(int argc, char argv[])
   {
5    int a;
6    int *b;
7    a = 10;
8    b = new int[10];
9    for( int i = 0; i < 10; i++){
10     b[i] = i;
11   }
12   func(a,b);
13   return 0;
14 } // end of main function
15 // definition of the function func
16 void func(int x, int *y)
17 {
18   x += 7;
19   *y += 10;
20   y[6] += 10;
21   return;
22 } // end function func
```

There are several features to be noted.

- Lines 5 and 6: Declaration of two variables a and b. The compiler reserves two locations in memory. The size of the location depends on the type of variable. Two properties are important for these locations – the address in memory and the content in the

- Line 7: The value of a is now 10.

- Line 8: Memory to store 10 integers is reserved. The address to the first location is stored in b. The address of element number 6 is given by the expression (b + 6).
- Line 10: All 10 elements of b are given values: b[0] = 0, b[1] = 1, ..., b[9] = 9;
- Line 12: The main() function calls the function func() and the program counter transfers to the first statement in func(). With respect to data the following happens. The content of a (= 10) and the content of b (a memory address) are copied to a stack (new memory location) associated with the function func()
- Line 16: The variable x and y are local variables in func(). They have the values – x = 10, y = address of the first element in b in the main() program.
- Line 18: The local variable x stored in the stack memory is changed to 17. Nothing happens with the value a in main().
- Line 19: The value of y is an address and the symbol *y stands for the position in memory which has this address. The value in this location is now increased by 10. This means that the value of b[0] in the main program is equal to 10. Thus func() has modified a value in main().
- Line 20: This statement has the same effect as line 9 except that it modifies element b[6] in main() by adding a value of 10 to what was there originally, namely 6.
- Line 21: The program counter returns to main(), the next expression after *func(a,b);*. All data on the stack associated with func() are destroyed.
- The value of a is transferred to func() and stored in a new memory location called x. Any modification of x in func() does not affect in any way the value of a in main(). This is called **transfer of data by value**. On the other hand the next argument in func() is an address which is transferred to func(). This address can be used to modify the corresponding value in main(). In the programming language C it is expressed as a modification of the value which y points to, namely the first element of b. This is called **transfer of data by reference** and is a method to transfer data back to the calling function, in this case main().

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

```
int n; n =8;
func(&n); /* &n is a pointer to n */
....
```

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

whereas in C++ we would write

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

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

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

Initializations and main program

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

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

http://folk.uio.no/mhjensen/compphys/programs/chapter03/cpp/program1.cpp

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

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

int main()
{
      // declarations of variables
      int number_of_steps;
      double x, initial_step;
 double *h_step, *computed_derivative;
      // read in input data from screen
      initialize (&initial_step, &x, &number_of_steps);
 // allocate space in memory for the one-dimensional arrays
 // h_step and computed_derivative
      h_step = new double[number_of_steps];
      computed_derivative = new double[number_of_steps];
 // compute the second derivative of exp(x)
      second_derivative( number_of_steps, x, initial_step, h_step,
                     computed_derivative);
      // Then we print the results to file
 output(h_step, computed_derivative, x, number_of_steps );
      // free memory
      delete [] h_step;
      delete [] computed_derivative;
      return 0;
}  // end main program
```

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

These arrays are defined as pointers through the statement

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

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

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

while the called function is declared in the following way

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

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

Another aspect to observe is the possibility of dynamical allocation of memory through the `new` function. In the included program we reserve space in memory for these three arrays in the following way

```
  h_step = new double[number_of_steps];
  computed_derivative = new double[number_of_steps];
```

When we no longer need the space occupied by these arrays, we free memory through the declarations

```
  delete [] h_step;
  delete [] computed_derivative;
```

The function initialize

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

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

This function receives the addresses of the three variables

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

and returns updated values by reading from screen.

The function second_derivative

```
// This function computes the second derivative

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

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

The output function

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

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

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

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

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

```
4
5 int main(int argc, char *argv[])
6 {
7   FILE *inn, *out;
8   int c;
9   if( argc < 3) {
10  printf("You have to read in :\n");
11  printf("in_file and out_file \n");
12  exit(1);
13  inn = fopen( argv[1], "r");} // returns pointer to the in_file
14  if( inn == NULL ) {   // can't find in_file
15    printf("Can't find the input file %s\n", argv[1]);
16    exit(1);
17  }
18  out = fopen( argv[2], "w"); // returns a pointer to the out_file
19  if( out == NULL ) {   // can't find out_file
20    printf("Can't find the output file %s\n", argv[2]);
21    exit(1);
22  }
   ... program statements

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

This program has several interesting features.

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

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

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

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

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

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

http://folk.uio.no/mhjensen/compphys/programs/chapter03/Fortran/program1.f90

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

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

MODULE functions
USE constants
IMPLICIT NONE
CONTAINS
  SUBROUTINE derivative(number_of_steps, x, initial_step, h_step, &
       computed_derivative)
    USE constants
    INTEGER, INTENT(IN) :: number_of_steps
    INTEGER :: loop
    REAL(DP), DIMENSION(number_of_steps), INTENT(INOUT) :: &
         computed_derivative, h_step
    REAL(DP), INTENT(IN) :: initial_step, x
    REAL(DP) :: h
    !    calculate the step size
    !    initialize the derivative, y and x (in minutes)
    !    and iteration counter
    h = initial_step
    ! start computing for different step sizes
    DO loop=1, number_of_steps
       ! setup arrays with derivatives and step sizes
       h_step(loop) = h
       computed_derivative(loop) = (EXP(x+h)-2.*EXP(x)+EXP(x-h))/(h*h)
       h = h*0.5
    ENDDO
  END SUBROUTINE derivative

END MODULE functions

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

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

END PROGRAM second_derivative
```

The MODULE declaration in Fortran allows one to place functions like the one which calculates second derivatives in a module. Since this is a general method, one could extend its functionality by simply transfering the name of the function to differentiate. In our case we use explicitely the exponential function, but there is nothing which hinders us from defining other functions. Note the usage of the module **constants** where we define double and complex variables. If one wishes to switch to another precision, one needs to change the declaration in one part of the program only. This hinders possible errors which arise if one has to change variable declarations in every function and subroutine. Finally, dynamic memory allocation and deallocation is in Fortran done with the keywords ALLOCATE(array(size)) and DEALLOCATE(array). Although most compilers deallocate and thereby free space in memory when leaving a function, you should always deallocate an array when it is no longer needed. In case your arrays are very large, this may block unnecessarily large fractions of the memory. Furthermore, you should always initialize arrays. In the example above, we note that Fortran allows us to simply write h_step = 0.0_dp; computed_derivative = 0.0_dp, which means that all elements of these two arrays are set to zero. Coding arrays in this manner brings us much closer to the way we deal with mathematics. In Fortran it is irrelevant whether this is a one-dimensional or multi-dimensional array. In chapter 6, where we deal with allocation of matrices, we will introduce the numerical libraries Armadillo and Blitz++ which allow for similar treatments of arrays in C++. By default however, these features are not included in the ANSI C++ standard.

2.7 Exercises

Exercise 2.1: Converting from decimal to binary representation

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

Exercise 2.1: Summing series

Make a program which sums

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

   and

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

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

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

prob 2.3: Finding alternative expressions

Write a program which computes

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

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

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

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

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

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

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

prob 2.4: Computing e^{-x}

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

prob 2.5: Computing the quadratic equation

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

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

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

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

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

prob 2.7: Nearest machine number

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

prob 2.8: Recurrence relations

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

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

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

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

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

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

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

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

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

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

Using Eq. (2.4) we obtain the final result

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

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

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

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

prob 2.9: Continued fractions

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

$$x^2-4x-1=0, equation which we rewrite as x=\frac{1}{4+x},$$

which in turn could be represented through an iterative substitution process

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

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

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

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

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

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

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

or as

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

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

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

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

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

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

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

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

and with the analytic (on closed form) expressions

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

and

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

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

1. Make first a function which computes the associated Legendre functions for different values of L and M. Compare with the closed-form results listed in chapter 5.

2. Make thereafter a program which calculates the real part of the spherical harmonics

3. Make plots for various $L = M$ as functions of θ (set $\phi = 0$) and study the behavior as L is increased. Try to explain why the functions become more and more narrow as L increases. In order to make these plots you can use for example gnuplot, as discussed in appendix 3.6.

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

Project 2.2: Special functions, Laguerre and Hermite polynomials

Other well-known polynomials are the Laguerre and the Hermite polynomials, both being solutions to famous differential equations. The Laguerre polynomials arise from the solution of the differential equation

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

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

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

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

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

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

and

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

They fulfil the orthogonality relation

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

and the recursion relation

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

Similalry, the Hermite polynomials are solutions of the differential equation

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

which arises for example by solving Schrödinger's equation for a particle confined to move in a harmonic oscillator potential. The first few polynomials are

$$H_0(x) = 1,$$

$$H_1(x) = 2x,$$

$$H_2(x) = 4x^2 - 2,$$

$$H_3(x) = 8x^3 - 12,$$

and

$$H_4(x) = 16x^4 - 48x^2 + 12.$$

They fulfil the orthogonality relation

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

and the recursion relation

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

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

Bibliography

[1] D. Kincaid and W. Cheney. *Numerical Analysis*. Brooks/Gole Publishing Company, 1996.

[2] H.P. Langtangen. *Introduction to Computer Programming; A Python-based approach for Computational Science*. Springer Verlag, 2009.

Chapter 3

Numerical differentiation and interpolation

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 the derivative f' or an integral $\int f(x)dx$. The aim of this chapter is to introduce some of these methods with a critical eye on numerical accuracy, following the discussion in the previous chapter. The next section deals essentially with topics from numerical differentiation. There we present also the most commonly used formulae for computing first and second derivatives, formulae which in turn find their most important applications in the numerical solution of ordinary and partial differential equations. We discuss also selected methods for numerical interpolation. This chapter serves also the scope of introducing some more advanced C++ programming concepts, such as call by reference and value, reading and writing to a file and the use of dynamic memory allocation. We will also discuss several object-oriented features of C++, ending the chapter with an analogous discussion of Fortran features.

3.2 Numerical Differentiation

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

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

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

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

We can then obtain an expression for the first derivative as

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

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

This means that we can represent the derivative with

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

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

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

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

$$f_2'(x) = 2bx + bh,$$

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

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

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

which we rewrite as

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

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

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

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

For our expression with a quadratic function $f(x) = a + bx^2$ we see that the three-point formula f_3' for the derivative gives the exact answer $2bx$. Thus, if our function has a quadratic behavior in x in a certain region of space, the three-point formula will result in reliable first derivatives in the interval $[-h,h]$. Using the relation

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

we can define the second derivative as

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

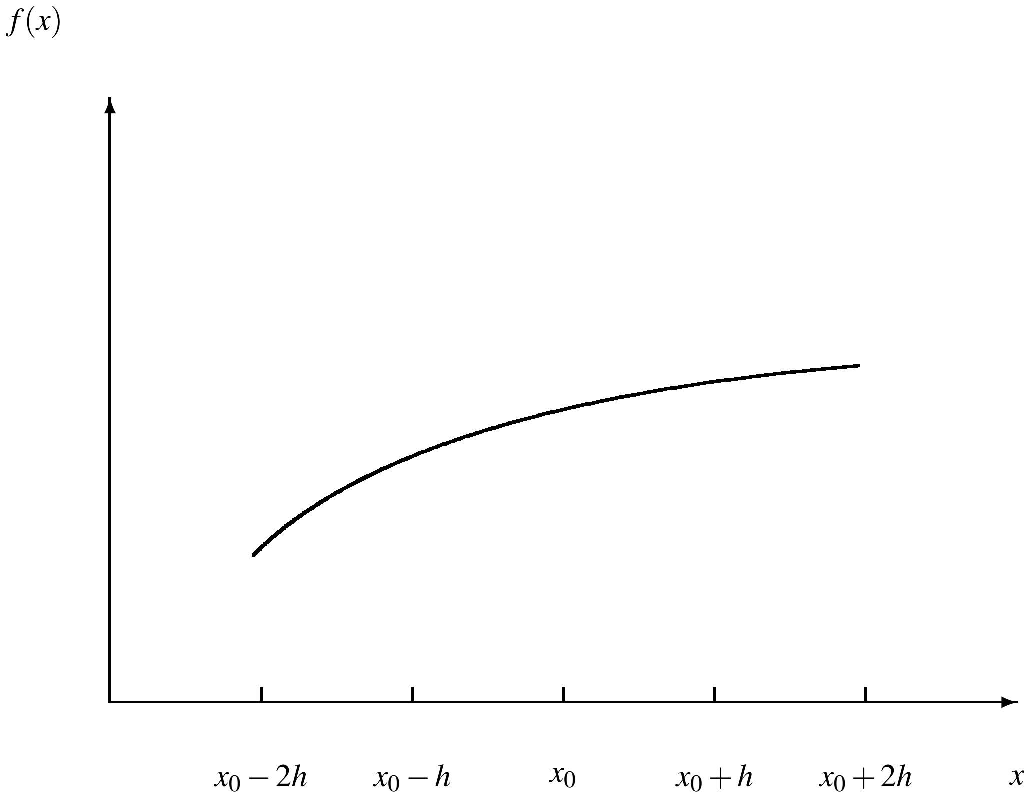

Figure 3.1: Demonstration of the subdivision of the x-axis into small steps h. Each point corresponds to a set of values $x, f(x)$. The value of x is incremented by the step length h. If we use the points x_0 and $x_0 + h$ we can draw a straight line and use the slope at this point to determine an approximation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

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

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

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

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

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

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

$$\frac{f_h - f_{-h}}{2h} = f'_0 + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j}, \tag{3.3}$$

and

$$\frac{f_h - 2f_0 + f_{-h}}{h^2} = f''_0 + 2\sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j}, \tag{3.4}$$

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

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

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

where a_{-h}, a_0 and a_h are unknown constants to be chosen so that $a_{-h} f_{-h} + a_0 f_0 + a_h f_h$ is the best possible approximation for f'_0 and f''_0. Eq. (3.5) can be rewritten as

$$a_{-h} f_{-h} + a_0 f_0 + a_h f_h = [a_{-h} + a_0 + a_h] f_0$$
$$+ [a_h - a_{-h}] h f'_0 + [a_{-h} + a_h] \frac{h^2 f''_0}{2} + \sum_{j=3}^{\infty} \frac{f_0^{(j)}}{j!} (h)^j \left[(-1)^j a_{-h} + a_h\right].$$

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

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

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

and

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

These equations have the solution

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

and

$$a_0 = 0,$$

yielding

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

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

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

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

and

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

These equations have the solution

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

and

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

yielding

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

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

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

Let us look at a simple case first, the use of `printf` and `scanf`. If we wish to print a variable defined as `double speed_of_sound;` we could for example write

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

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

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

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

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

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

Consider now the following program

```
using namespace std;
# include <iostream>
// begin main function
int main(int argc, char argv[])
{
  int a;
  int *b;
  a = 10;
8    b = new int[10];
9    for( int i = 0; i < 10; i++){
10     b[i] = i;
11   }
12   func(a,b);
13   return 0;
14 } // end of main function
15 // definition of the function func
16 void func(int x, int *y)
17 {
18   x += 7;
19   *y += 10;
20   y[6] += 10;
21   return;
22 } // end function func
```

There are several features to be noted.

- Lines 5 and 6: Declaration of two variables a and b. The compiler reserves two locations in memory. The size of the location depends on the type of variable. Two properties are important for these locations – the address in memory and the content in the

- Line 7: The value of a is now 10.

- Line 8: Memory to store 10 integers is reserved. The address to the first location is stored in b. The address of element number 6 is given by the expression (b + 6).

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

- Line 12: The main() function calls the function func() and the program counter transfers to the first statement in func(). With respect to data the following happens. The content of a (= 10) and the content of b (a memory address) are copied to a stack (new memory location) associated with the function func()

- Line 16: The variable x and y are local variables in func(). They have the values – x = 10, y = address of the first element in b in the main() program.

- Line 18: The local variable x stored in the stack memory is changed to 17. Nothing happens with the value a in main().

- Line 19: The value of y is an address and the symbol *y stands for the position in memory which has this address. The value in this location is now increased by 10. This means that the value of b[0] in the main program is equal to 10. Thus func() has modified a value in main().

- Line 20: This statement has the same effect as line 9 except that it modifies element b[6] in main() by adding a value of 10 to what was there originally, namely 6.
- Line 21: The program counter returns to main(), the next expression after *func(a,b);*. All data on the stack associated with func() are destroyed.
- The value of a is transferred to func() and stored in a new memory location called x. Any modification of x in func() does not affect in any way the value of a in main(). This is called **transfer of data by value**. On the other hand the next argument in func() is an address which is transferred to func(). This address can be used to modify the corresponding value in main(). In the programming language C it is expressed as a modification of the value which y points to, namely the first element of b. This is called **transfer of data by reference** and is a method to transfer data back to the calling function, in this case main().

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

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

whereas in C++ we would write

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

Note well that the way we have defined the input to the function `func(int& i)` or `func(int *i)` decides how we transfer variables to a specific function. The reason why we emphasize the difference between call by value and call by reference is that it allows the programmer to avoid pitfalls like unwanted changes of variables. However, many people feel that this reduces the readability of the code. It is more or less common in C++ to use call by reference, since it gives a much cleaner code. Recall also that behind the curtain references are usually implemented as pointers.

When we transfer large objects such a matrices and vectors one should always use call by reference. Copying such objects to a called function slows down considerably the execution. If you need to keep the value of a call by reference object, you should use the `const` declaration.

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

Initializations and main program

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

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

http://folk.uio.no/mhjensen/compphys/programs/chapter03/cpp/program1.cpp

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

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

int main()
{
      // declarations of variables
```

```
     int number_of_steps;
     double x, initial_step;
 double *h_step, *computed_derivative;
     // read in input data from screen
     initialize (&initial_step, &x, &number_of_steps);
 // allocate space in memory for the one-dimensional arrays
 // h_step and computed_derivative
     h_step = new double[number_of_steps];
     computed_derivative = new double[number_of_steps];
 // compute the second derivative of exp(x)
     second_derivative( number_of_steps, x, initial_step, h_step,
                    computed_derivative);
     // Then we print the results to file
 output(h_step, computed_derivative, x, number_of_steps );
     // free memory
     delete [] h_step;
     delete [] computed_derivative;
     return 0;
}  // end main program
```

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

These arrays are defined as pointers through the statement

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

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

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

while the called function is declared in the following way

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

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

Another aspect to observe is the possibility of dynamical allocation of memory through the `new` function. In the included program we reserve space in memory for these three arrays in the following way

```
 h_step = new double[number_of_steps];
 computed_derivative = new double[number_of_steps];
```

When we no longer need the space occupied by these arrays, we free memory through the declarations

```
 delete [] h_step;
 delete [] computed_derivative;
```

The function initialize

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

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

This function receives the addresses of the three variables

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

and returns updated values by reading from screen.

The function second_derivative

```
// This function computes the second derivative

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

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

The output function

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

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

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

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

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

int main(int argc, char *argv[])
{
  FILE *inn, *out;
  int c;
  if( argc < 3) {
  printf("You have to read in :\n");
  printf("in_file and out_file \n");
  exit(1);
  inn = fopen( argv[1], "r");} // returns pointer to the in_file
  if( inn == NULL ) {   // can't find in_file
    printf("Can't find the input file %s\n", argv[1]);
    exit(1);
  }
  out = fopen( argv[2], "w"); // returns a pointer to the out_file
  if( out == NULL ) {   // can't find out_file
    printf("Can't find the output file %s\n", argv[2]);
    exit(1);
  }
 ... program statements

  fclose(inn);
  fclose(out);
  return 0;
}
```

This program has several interesting features.

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

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

http://folk.uio.no/mhjensen/compphys/programs/chapter03/cpp/program3.cpp

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

ofstream ofile;

int main(int argc, char* argv[])
{
   // declarations of variables
```

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

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

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

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

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

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

http://folk.uio.no/mhjensen/compphys/programs/chapter03/Fortran/program1.f90

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

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

MODULE functions
USE constants
IMPLICIT NONE
CONTAINS
  SUBROUTINE derivative(number_of_steps, x, initial_step, h_step, &
       computed_derivative)
    USE constants
    INTEGER, INTENT(IN) :: number_of_steps
    INTEGER :: loop
    REAL(DP), DIMENSION(number_of_steps), INTENT(INOUT) :: &
```

```
         computed_derivative, h_step
    REAL(DP), INTENT(IN) :: initial_step, x
    REAL(DP) :: h
    !    calculate the step size
    !    initialize the derivative, y and x (in minutes)
    !    and iteration counter
    h = initial_step
    ! start computing for different step sizes
    DO loop=1, number_of_steps
       ! setup arrays with derivatives and step sizes
       h_step(loop) = h
       computed_derivative(loop) = (EXP(x+h)-2.*EXP(x)+EXP(x-h))/(h*h)
       h = h*0.5
    ENDDO
  END SUBROUTINE derivative

END MODULE functions

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

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

```
END PROGRAM second_derivative
```

The `MODULE` declaration in Fortran allows one to place functions like the one which calculates second derivatives in a module. Since this is a general method, one could extend its functionality by simply transfering the name of the function to differentiate. In our case we use explicitely the exponential function, but there is nothing which hinders us from defining other functions. Note the usage of the module **constants** where we define double and complex variables. If one wishes to switch to another precision, one needs to change the declaration in one part of the program only. This hinders possible errors which arise if one has to change variable declarations in every function and subroutine. Finally, dynamic memory allocation and deallocation is in Fortran done with the keywords `ALLOCATE( array(size))` and `DEALLOCATE(array)`. Although most compilers deallocate and thereby free space in memory when leaving a function, you should always deallocate an array when it is no longer needed. In case your arrays are very large, this may block unnecessarily large fractions of the memory. Furthermore, you should always initialize arrays. In the example above, we note that Fortran allows us to simply write `h_step = 0.0_dp; computed_derivative = 0.0_dp`, which means that all elements of these two arrays are set to zero. Coding arrays in this manner brings us much closer to the way we deal with mathematics. In Fortran it is irrelevant whether this is a one-dimensional or multi-dimensional array. In chapter 6, where we deal with allocation of matrices, we will introduce the numerical libraries Armadillo and Blitz++ which allow for similar treatments of arrays in C++. By default however, these features are not included in the ANSI C++ standard.

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

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

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

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

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

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

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

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

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

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

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

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

and the truncation or approximation error goes like

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

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

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

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

We have then

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

Our total error becomes

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

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

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

With double precision and $x = 10$ we obtain

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

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

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

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

as

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

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

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

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

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

$-4 < log_{10}(h) < -2$, we could extract a slope close to 2, in agreement with the mathematical expression for the truncation error.

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

3.3 Numerical Interpolation and Extrapolation

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

We may then need to evaluate the function f at some point x within the data set $x_1 \dots x_n$, but where x differs from the tabulated values. In this case we are dealing with interpolation. If x is outside we are left with the more troublesome problem of numerical extrapolation. Below we will concentrate on two methods for interpolation and extrapolation, namely polynomial interpolation and extrapolation. The cubic spline interpolation approach is discussed in chapter 6.

3.3.1 Interpolation

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

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

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

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

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

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

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

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

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

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

and with three points (a parabolic approximation) we have

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

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

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

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

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

and $\xi = \xi(x)$ is a point in the smallest interval containing all interpolation points x_j and x. The program we provide below is however based on divided differences. The recipe is quite simple. If we take $x = x_0$ in Eq. (3.8), we then have obviously that $a_0 = f(x_0) = y_0$. Moving a_0 over to the left-hand side and dividing by $x - x_0$ we have

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

where we hereafter omit the rest term

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

The quantity

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

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

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

and the quantity

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

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

$$a_1 = f_{01},$$

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

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

meaning that the corresponding coefficient a_k is given by

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

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

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

If we replace $x_0, x_1, \dots, x_k$ in Eq. (3.11) with $x_{i+1}, x_{i+2}, \dots, x_k$, that is we count from $i+1$ to k instead of counting from 0 to k and replace x with x_i, we can then construct the following recursive algorithm for the calculation of divided differences

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

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

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

The coefficients we are searching for will then be the elements along the main diagonal. We can understand this algorithm by considering the following. First we construct the unique polynomial of order zero which passes through the point x_0, y_0. This is just a_0 discussed above. Therafter we construct the unique polynomial of order one which passes through both x_0y_0 and x_1y_1. This corresponds to the coefficient a_1 and the tabulated value $f_{x_0x_1}$ and together with a_0 results in the polynomial for a straight line. Likewise we define polynomial coefficients for all other couples of points such as $f_{x_1x_2}$ and $f_{x_2x_3}$. Furthermore, a coefficient like $a_2 = f_{x_0x_1x_2}$ spans now three points, and adding together $f_{x_0x_1}$ we obtain a polynomial which represents three points, a parabola. In this fashion we can continue till we have all coefficients. The function we provide below included is based on an extension of this algorithm, knowns as Neville's algorithm. The error provided by Neville's algorithm is based on the truncation error in Eq. (3.10).

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

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

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

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

3.3.2 Richardson's deferred extrapolation method

Here we present an elegant method to improve the precision of our mathematical truncation, without too many additional function evaluations. We will again study the evaluation of the first and second derivatives of $\exp(x)$ at a given point $x = \xi$. In Eqs. (3.3) and (3.4) for the first and second derivatives, we noted that the truncation error goes like $O(h^{2j})$.

Employing the mid-point approximation to the derivative, the various derivatives D of a given function $f(x)$ can then be written as

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

where $D(h)$ is the calculated derivative, $D(0)$ the exact value in the limit $h \to 0$ and a_i are independent of h. By choosing smaller and smaller values for h, we should in principle be able to approach the exact value. However, since the derivatives involve differences, we may easily loose numerical precision as shown in the previous sections. A possible cure is to apply Richardson's deferred approach, i.e., we perform calculations with several values of the step h and extrapolate to $h = 0$. The philososphy is to combine different values of h so that the terms in the above equation involve only large exponents for h. To see this, assume that we mount a calculation for two values of the step h, one with h and the other with $h/2$. Then we have

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

and

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

and we can eliminate the term with a_1 by combining

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

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

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

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

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

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

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

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

where the elements in the first column represent the given approximations

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

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

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

with $m > 0$.

In Table 3.1 we presented the results for various step sizes for the second derivative of $\exp(x)$ using $f''_0 = \frac{f_h - 2f_0 + f_{-h}}{h^2}$. The results were compared with the exact ones for various x values. Note well that as the step is decreased we get closer to the exact value. However, if it is further increased, we run into problems of loss of precision. This is clearly seen for $h = 0.000001$. This means that even though we could let the computer run with smaller and smaller values of the step, there is a limit for how small the step can be made before we loose precision. Consider now the results in Table 3.3 where we choose to employ Richardson's extrapolation scheme. In this calculation we have computed our function with only three possible values for the step size, namely h, $h/2$ and $h/4$ with $h = 0.1$. The agreement with the exact value is amazing! The extrapolated result is based upon the use of Eq. (3.13). An

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.

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

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

3.4 Classes in C++

In Fortran a vector (this applies to matrices as well) starts with 1, but it is easy to change the declaration of vector so that it starts with zero or even a negative number. If we have a double precision Fortran vector which starts at -10 and ends at 10, we could declare it as `REAL(KIND=8) :: vector(-10:10)`. Similarly, if we want to start at zero and end at 10 we could write `REAL(KIND=8) :: vector(0:10)`.

Fortran allows us to write a vector addition $\mathbf{a} = \mathbf{b} + \mathbf{c}$ as `a = b + c`. This means that we have overloaded the addition operator in order to translate this operation into two loops and an addition of two vector elements $a_i = b_i + c_i$.

The way the vector addition is written is very close to the way we express this relation mathematically. The benefit for the programmer is that our code is easier to read. Furthermore, such a way of coding makes it more likely to spot eventual errors as well.

In Ansi C and C++ arrays start by default from $i = 0$. Moreover, if we wish to add two vectors we need to explicitely write out a loop as

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

However, the strength of C++ over programming languages like C and Fortran 77 is the possibility to define new data types, tailored to some particular problem. Via new data types and overloading of operations such as addition and subtraction, we can easily define sets of operations and data types which allow us to write a vector or matrix addition in exactly the same way as we would do in Fortran. We could also change the way we declare a C++ vector (or matrix) element a_i, from $a[i]$ to say $a(i)$, as we would do in Fortran. Similarly, we could also change the default range from $0 : n-1$ to $1 : n$.

To achieve this we need to introduce two important entities in C++ programming, classes and templates.

The function and class declarations are fundamental concepts within C++. Functions are abstractions which encapsulate an algorithm or parts of it and perform specific tasks in a program. We have already met several examples on how to use functions. Classes can be defined as abstractions which encapsulate data and operations on these data. The data can be very complex data structures and the class can contain particular functions which operate on these data. Classes allow therefore for a higher level of abstraction in computing. The elements (or components) of the data type are the class data members, and the procedures are the class member functions.

Classes are user-defined tools used to create multi-purpose software which can be reused by other classes or functions. These user-defined data types contain data (variables) and functions operating on the data.

A simple example is that of a point in two dimensions. The data could be the x and y coordinates of a given point. The functions we define could be simple read and write functions or the possibility to compute the distance between two points.

The two examples we elaborate on below demonstrate most of the features of classes. We develop first a class called `Complex` which allows us to perform various operations on complex variables. We extend thereafter our discussion of classes to define a class `Vector` which allows us to perform various operations on a user-specified one-dimesional array, from declarations of a vector to mathematical operations such as additions of vectors. Later, in our discussion on linear algebra, we will also present our final matrix and vector class.

The classes we define are easy to use in other codes and/or other classes and many of the details which would be present in C (or Fortran 77) codes are hidden inside the class. The reuse of a well-written and functional class is normally rather simple. However, to write a given class is often complicated, especially if we deal with complicated matrix operations. In this text we will rely on ready-made classes in C++ for dealing with matrix operations. We have chosen to use the libraries like Armadillo or Blitz++, discussed in our linear algebra chapter. These libraries hide many low-level operations with matrices and vectors, such as matrix-vector multiplications or allocation and deallocation of memory. Such libraries make it then easier to build our own high-level classes out of well-tested lower-level classes.

The way we use classes in this text is close to the `MODULE` data type in Fortran and we provide some simple demonstrations at the end of this section.

3.4.1 The Complex class

As remarked in chapter 2, C++ has a class complex in its standard template library (STL). The standard usage in a given function could then look like

```
// Program to calculate addition and multiplication of two complex numbers
using namespace std;
#include <iostream>
#include <cmath>
#include <complex>
int main()
{
 complex<double> x(6.1,8.2), y(0.5,1.3);
 // write out x+y
 cout << x + y << x*y << endl;
 return 0;
}
```

where we add and multiply two complex numbers $x = 6.1 + \imath 8.2$ and $y = 0.5 + \imath 1.3$ with the obvious results $z = x + y = 6.6 + \imath 9.5$ and $z = x \cdot y = -7.61 + \imath 12.03$. In Fortran we would declare the above variables as `COMPLEX(DPC) :: x(6.1,8.2), y(0.5,1.3)`.

The libraries Armadillo and Blitz++ include an extension of the complex class to operations on vectors, matrices and higher-dimensional arrays. We recommend the usage of such libraries when you develop your own codes. However, writing a complex class yourself is a good pedagogical exercise.

We proceed by splitting our task in three files.

- We define first a header file complex.h which contains the declarations of the class. The header file contains the class declaration (data and functions), declaration of stand-alone functions, and all inlined functions, starting as follows

```
#ifndef Complex_H
#define Complex_H
//  various include statements and definitions
#include <iostream>   // Standard ANSI-C++ include files
#include <new>
#include ....

class Complex
{...
definition of variables and their character
};
//  declarations of various functions used by the class
...
#endif
```

- Next we provide a file complex.cpp where the code and algorithms of different functions (except inlined functions) declared within the class are written. The files complex.h and complex.cpp are normally placed in a directory with other classes and libraries we have defined.

- Finally,we discuss here an example of a main program which uses this particular class. An example of a program which uses our complex class is given below. In particular we would like our class to perform tasks like declaring complex variables, writing out the real and imaginary part and performing algebraic operations such as adding or multiplying two complex numbers.

```
#include "Complex.h"
... other include and declarations
int main ()
{
  Complex a(0.1,1.3); // we declare a complex variable a
  Complex b(3.0), c(5.0,-2.3); // we declare complex variables b and c
  Complex d = b;    // we declare a new complex variable d
  cout << "d=" << d << ", a=" << a << ", b=" << b << endl;
  d = a*c + b/a; // we add, multiply and divide two complex numbers
  cout << "Re(d)=" << d.Re() << ", Im(d)=" << d.Im() << endl; // write out of
       the real and imaginary parts
}
```

We include the header file complex.h and define four different complex variables. These are $a = 0.1 + \imath 1.3$, $b = 3.0 + \imath 0$ (note that if you don't define a value for the imaginary part this is set to zero), $c = 5.0 - \imath 2.3$ and $d = b$. Thereafter we have defined standard algebraic operations and the member functions of the class which allows us to print out the real and imaginary part of a given variable.

To achieve these features, let us see how we define the complex class. In C++ we could define a complex class as follows

```
class Complex
{
private:
  double re, im; // real and imaginary part
public:
  Complex ();                      // Complex c;
  Complex (double re, double im = 0.0); // Definition of a complex variable;
  Complex (const Complex& c);    // Usage: Complex c(a); // equate two complex
      variables
  Complex& operator= (const Complex& c); // c = a; // equate two complex variables,
      same as previous
 ~Complex () {}                 // destructor
  double Re () const;   // double real_part = a.Re();
  double Im () const;   // double imag_part = a.Im();
  double abs () const;  // double m = a.abs(); // modulus
  friend Complex operator+ (const Complex& a, const Complex& b);
  friend Complex operator- (const Complex& a, const Complex& b);
  friend Complex operator* (const Complex& a, const Complex& b);
  friend Complex operator/ (const Complex& a, const Complex& b);
};
```

The class is defined via the statement `class Complex`. We must first use the key word `class`, which in turn is followed by the user-defined variable name `Complex`. The body of the class, data and functions, is encapsulated within the parentheses {...};.

Data and specific functions can be private, which means that they cannot be accessed from outside the class. This means also that access cannot be inherited by other functions outside the class. If we use `protected` instead of `private`, then data and functions can be inherited outside the class. The key word `public` means that data and functions can be accessed from outside the class. Here we have defined several functions which can be accessed by functions outside the class. The declaration `friend` means that stand-alone functions can work on privately declared variables of the type `(re, im)`. Data members of a class should be declared as private variables.

The first public function we encounter is a so-called constructor, which tells how we declare a variable of type `Complex` and how this variable is initialized. We have chosen three possibilities in the example above:

1. A declaration like `Complex c;` calls the member function `Complex()` which can have the following implementation

```
Complex:: Complex () { re = im = 0.0; }
```

meaning that it sets the real and imaginary parts to zero. Note the way a member function is defined. The constructor is the first function that is called when an object is instantiated.

2. Another possibility is

```
Complex:: Complex () {}
```

which means that there is no initialization of the real and imaginary parts. The drawback is that a given compiler can then assign random values to a given variable.

3. A call like `Complex a(0.1,1.3);` means that we could call the member function as

```
Complex:: Complex (double re_a, double im_a)
{ re = re_a; im = im_a; }
```

The simplest member function are those we defined to extract the real and imaginary part of a variable. Here you have to recall that these are private data, that is they are invisible for users of the class. We obtain a copy of these variables by defining the functions

```
double Complex:: Re () const { return re; }} // getting the real part
double Complex:: Im () const { return im; } // and the imaginary part
\end{lstlistingline}
Note that we have introduced the declaration \verb?const}. What does it mean?
This declaration means that a variable cannot be changed within a called function.
If we define a variable as
\verb?const double p = 3;? and then try to change its value, we will get an error
    when we
compile our program. This means that constant arguments in functions cannot be
    changed.
\begin{lstlisting}
// const arguments (in functions) cannot be changed:
void myfunc (const Complex& c)
{ c.re = 0.2; /* ILLEGAL!! compiler error... */ }
```

If we declare the function and try to change the value to 0.2, the compiler will complain by sending an error message. If we define a function to compute the absolute value of complex variable like

```
double Complex:: abs () { return sqrt(re*re + im*im);}
```

without the constant declaration and define thereafter a function `myabs` as

```
double myabs (const Complex& c)
{ return c.abs(); } // Not ok because c.abs() is not a const func.
```

the compiler would not allow the c.abs() call in myabs since `Complex::abs` is not a constant member function. Constant functions cannot change the object's state. To avoid this we declare the function `abs` as

```
double Complex:: abs () const { return sqrt(re*re + im*im); }
```

Overloading operators

C++ (and Fortran) allows for overloading of operators. That means we can define algebraic operations on for example vectors or any arbitrary object. As an example, a vector addition of the type $\mathbf{c} = \mathbf{a} + \mathbf{b}$ means that we need to write a small part of code with a for-loop over the dimension of the array. We would rather like to write this statement as `c = a+b;` as this makes the code much more readable and close to eventual equations we want to code. To achieve this we need to extend the definition of operators.

Let us study the declarations in our complex class. In our main function we have a statement like `d = b;`, which means that we call `d.operator= (b)` and we have defined a so-called assignment operator as a part of the class defined as

```
Complex& Complex:: operator= (const Complex& c)
{
   re = c.re;
   im = c.im;
   return *this;
}
```

With this function, statements like `Complex d = b;` or `Complex d(b);` make a new object d, which becomes a copy of b. We can make simple implementations in terms of the assignment

```
Complex:: Complex (const Complex& c)
{ *this = c; }
```

which is a pointer to "this object", `*this` is the present object, so `*this = c;` means setting the present object equal to c, that is `this->operator= (c);`.

The meaning of the addition operator $+$ for complex objects is defined in the function

```
Complex operator+ (const Complex& a, const Complex& b);
```

The compiler translates `c = a + b;` into `c = operator+ (a, b);`. Since this implies the call to a function, it brings in an additional overhead. If speed is crucial and this function call is performed inside a loop, then it is more difficult for a given compiler to perform optimizations of a loop. The solution to this is to inline functions. We discussed inlining in chapter 2. Inlining means that the function body is copied directly into the calling code, thus avoiding calling the function. Inlining is enabled by the inline keyword

```
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex (a.re + b.re, a.im + b.im); }
```

Inline functions, with complete bodies must be written in the header file complex.h. Consider the case `c = a + b`; that is, `c.operator= (operator+ (a,b));` If `operator+`, `operator=` and the constructor `Complex(r,i)` all are inline functions, this transforms to

```
c.re = a.re + b.re;
c.im = a.im + b.im;
```

by the compiler, i.e., no function calls

The stand-alone function `operator+` is a friend of the Complex class

```
class Complex
{
  ...
  friend Complex operator+ (const Complex& a, const Complex& b);
  ...
};
```

so it can read (and manipulate) the private data parts *re* and *im* via

```
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex (a.re + b.re, a.im + b.im); }
```

Since we do not need to alter the re and im variables, we can get the values by Re() and Im(), and there is no need to be a friend function

```
inline Complex operator+ (const Complex& a, const Complex& b)
{ return Complex (a.Re() + b.Re(), a.Im() + b.Im()); }
```

The multiplication functionality can now be extended to imaginary numbers by the following code

```
inline Complex operator* (const Complex& a, const Complex& b)
{
 return Complex(a.re*b.re - a.im*b.im, a.im*b.re + a.re*b.im);
}
```

It will be convenient to inline all functions used by this operator. To inline the complete expression `a*b;`, the constructors and `operator=` must also be inlined. This can be achieved via the following piece of code

```
inline Complex:: Complex () { re = im = 0.0; }
inline Complex:: Complex (double re_, double im_)
{ ... }
inline Complex:: Complex (const Complex& c)
{ ... }
inline Complex:: operator= (const Complex& c)
{ ... }
// e, c, d are complex
e = c*d;
// first compiler translation:
e.operator= (operator* (c,d));
// result of nested inline functions
// operator=, operator*, Complex(double,double=0):
e.re = c.re*d.re - c.im*d.im;
e.im = c.im*d.re + c.re*d.im;
```

The definitions `operator-` and `operator/` follow the same setup.

Finally, if we wish to write to file or another device a complex number using the simple syntax `cout << c;`, we obtain this by defining the effect of $<<$ for a Complex object as

```
ostream& operator<< (ostream& o, const Complex& c)
{ o << "(" << c.Re() << "," << c.Im() << ") "; return o;}
```

Templates

The reader may have noted that all variables and some of the functions defined in our class are declared as doubles. What if we wanted to make a class which takes integers or floating point numbers with single precision? A simple way to achieve this is copy and paste our class and replace `double` with for example `int`.

C++ allows us to do this automatically via the usage of templates, which are the C++ constructs for parameterizing parts of classes. Class templates is a template for producing classes. The declaration consists of the keyword `template` followed by a list of template arguments enclosed in brackets. We can therefore make a more general class by rewriting our original example as

```
template<class T>
class Complex
{
private:
  T re, im; // real and imaginary part
public:
  Complex ();                      // Complex c;
  Complex (T re, T im = 0); // Definition of a complex variable;
  Complex (const Complex& c);    // Usage: Complex c(a); // equate two complex
      variables
```

```
   Complex& operator= (const Complex& c); // c = a; // equate two complex variables,
       same as previous
  ~Complex () {}                // destructor
   T  Re () const;   // T real_part = a.Re();
   T  Im () const;   // T imag_part = a.Im();
   T  abs () const;  // T m = a.abs(); // modulus
   friend Complex operator+ (const Complex& a, const Complex& b);
   friend Complex operator- (const Complex& a, const Complex& b);
   friend Complex operator* (const Complex& a, const Complex& b);
   friend Complex operator/ (const Complex& a, const Complex& b);
};
```

What it says is that `Complex` is a parameterized type with T as a parameter and T has to be a type such as double or float. The class complex is now a class template and we would define variables in a code as

```
Complex<double> a(10.0,5.1);
Complex<int> b(1,0);
```

Member functions of our class are defined by preceding the name of the function with the `template` keyword. Consider the function we defined as

```
Complex:: Complex (double re_a, double im_a)
```

We could rewrite this function as

```
template<class T>
Complex<T>:: Complex (T re_a, T im_a)
{ re = re_a; im = im_a; }
```

The member functions are otherwise defined following ordinary member function definitions.

To write a class like the above is rather straightforward. The class for handling one-dimensional arrays, presented in the next subsection shows some of the additional possibilities which C++ offers. However, it can be rather difficult to write good classes for handling matrices or more complex objects. For such applications we recommend therefore the usage of ready-made libraries like Blitz++ or Armadillo.

Blitz++ http://www.oonumerics.org/blitz/ is a C++ library whose two main goals are to improve the numerical efficiency of C++ and to extend the conventional dense array model to incorporate new and useful features. Some examples of such extensions are flexible storage formats, tensor notation and index placeholders. It allows you also to write several operations involving vectors and matrices in a simple and clear (from a mathematical point of view) way. The way you would code the addition of two matrices looks very similar to the way it is done in Fortran. From a computational point of view, a library like Armadillo http://arma.sourceforge.net/, which contains much of the array functionality included in Blitz++, is preferred. Armadillo is a C++ linear algebra library that aims towards a good balance between speed and ease of use. It includes optional integration possibilities with popular linear algebra packages like LAPACK and BLAS, see chapter 6 for further discussions.

3.4.2 The vector class

Our next next example is a very simple class to handle one-dimensional arrays. It demonstrates again many aspects of C++ programming. However, most likely you will end up using a ready-made array class from libraries like Blitz++ or Armadillo discussed above. Furthermore, as was the case for the complex class, C++ contains also its own class for one-dimensional arrays, that is a vector class. At the end however, we recommend that you use libraries like Armadillo.

Our class `Vector` has as data a plain one-dimensional array. We define several functions which operate on these data, from subscripting, change of the length of the array, assignment to another vector, inner product with another vector etc etc. To be more specific, we define the following usage of our class,that is the way the class is used in another part of the program:

- Create vectors of a specified length defining a vector as `Vector\ v(n);` Via this statement we allocate space in memory for a vector with n elements.
- Create a vector with zero length by writing the statement `Vector v;`
- Change the dimension of a vector v to a given length n by declaring `v.redim(n);`. Note here the way we use a function defined within a class. The function here is `redim`.
- Create a vector as a copy of another vector by simply writing `Vector v(w);`
- To extract the length of the vector by writing `const int n = v.size();`
- To find particular value of the vector `double e = v(i);`
- or assign a number to an entry via `v(j) = e;`
- We would also like to set two vectors equal to each other by simply writing `w = v;`
- or take the inner product of two vectors as `double a = w.inner(v);` or alternatively `a = inner(w,v);`
- To write out the content of a vector could be done by via `v.print(cout);`

This list can be made longer by adding features like vector algebra, operator overloading etc.

We present now the declaration of the class, with our comments on the various declarations.

```
class Vector
{
private:
  double* A;              // vector entries
  int    length;          // the length ofthe vector
  void   allocate (int n); // allocate memory, length=n
  void   deallocate();    // free memory
public:
  Vector ();             // Constructor, use as Vector v;
  Vector (int n);        // use as Vector v(n);
  Vector (const Vector& w); // us as Vector v(w);
 ~Vector ();             // destructor to clean up dynamic memory

  bool redim (int n);            // change length, us as v.redim(m);
  Vector& operator= (const Vector& w);// set two vectors equal v = w;
  double operator() (int i) const; // a = v(i);
  double& operator() (int i);    // v(i) = a;

  void print (std::ostream& o) const; // v.print(cout);
  double inner (const Vector& w) const; // a = v.inner(w);
  int size () const { return length; } // n = v.size();
};
```

The class is defined via the statement `class Vector`. We must first use the key word `class`, which in turn is followed by the user-defined variable name. The body of the class, data and functions, is encapsulated within the parentheses ...;.

Data and specific functions can be private, which means that they cannot be accessed from outside the class. This means also that access cannot be inherited by other functions outside the class. If we use `protected` instead of `private`, then data and functions can be inherited outside the class. The key word `public` means that data and functions can be accessed from outside the class. Here we have defined several functions which can be accessed by functions outside the class.

The first public function we encounter is a so-called constructor, which tells how we declare a variable of type `Vector` and how this variable is initialized

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

    // this actually means calling the function

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

The constructor is the first function that is called when an object is instantiated. The variable A is the vector entry which defined as a private entity. Here the length is set to zero. Note also the way we define a method within the class by writing `Vector::Vector ()`. The general form is `< return type> name of class :: name of metho`

To give our vector v a dimensionality n we would write

```
Vector v(n); // declare a vector of length n
// means calling the function
Vector::Vector (int n)
{ allocate(n); }
void Vector::allocate (int n)
{
  length = n;
  A = new double[n]; // create n doubles in memory
}
```

Note that we defined a Fortran-like function for allocating memory. This is one of nice features of C++ for Fortran programmers, one can always define a Fortran-like world if one wishes. Moreover,the private function `allocate` operates on the private variables `length` and `A`. A `Vector` object is created (dynamically) at run time, but must also be destroyed when it is no longer in use. The destructor specifies how to destroy the object via the tilde symbol shown here

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

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

Again we have define a deallocation statement which mimicks the Fortran way of removing an object from memory. The observant reader may also have discovered that we have sneaked in the word 'object'. What do we mean by that? A clarification is needed. We will always refer to a class as user defined and declared variable which encapsulates various data (of a given type) and operations on these data. An object on the other hand is an instance of a variable of a given type. We refer to every variable we create and use as an object of a given type. The variable A above is an object of type `int`.

The function where we set two vectors to have the same length and have the same values can be written as

```
// v and w are Vector objects
v = w;
```

```
    // means calling
    Vector& Vector::operator= (const Vector& w)
    // for setting v = w;
    {
      redim (w.size()); // make v as long as w
      int i;
      for (i = 0; i < length; i++) { // (C++ arrays start at 0)
        A[i] = w.A[i]; // fill in teh vector w
      }
      return *this;
    }
    // return of *this, i.e. a Vector&, allows nested operations
    u = v = u_vec = v_vec;
```

where we have used the `redim` function

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

    bool Vector::redim (int n)
    {
      if (length == n)
        return false; // no need to allocate anything
      else {
        if (A != NULL) {
          // "this" object has already allocated memory
          deallocate();
        }
        allocate(n);
        return true; // the length was changed
      }
    }
```

and the copy action is defined as

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

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

Here we have defined `this` to be a pointer to the current (“this”) object, in other words `this` is the object itself.

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

```
double a = v.inner(w);

double Vector::inner (const Vector& w) const
{
  int i; double sum = 0;
  for (i = 0; i < length; i++)
    sum += A[i]*w.A[i];
  // alternative:
  // for (i = 1; i <= length; i++) sum += (*this)(i)*w(i);
  return sum;
}
```

```
// Vector v
cout << v;

ostream& operator<< (ostream& o, const Vector& v)
{ v.print(o); return o; }

// must return ostream& for nested output operators:
cout << "some text..." << w;

// this is realized by these calls:
operator<< (cout, "some text...");
operator<< (cout, w);
```

We can redefine the multiplication operator to mean the inner product of two vectors:

```
      double a = v*w; // example on attractive syntax

      class Vector
      {
        ...
        // compute (*this) * w
        double operator* (const Vector& w) const;
        ...
      };

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

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

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

Classes and templates in C++

We can again use templates to generalize our class to accept other types than just doubles. To achieve that we use templates, which are the native C++ constructs for parameterizing parts of classes, using statements like

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

In a code which uses this class we could declare various vectors as Declarations in user code:

```
Vector<double> a(10);
Vector<int> i(5);
```

where the first variable is double vector with ten elements while the second is an integer vector with five elements.

Summarizing, it is easy to use the class `Vector` and we can hide in the class many details which are visible in C and Fortran 77 codes. However, as you may have noted it is not easy to write class `Vector`. One ends often up with using ready-made classes in C++ libraries such as Blitz++ or Armadillo unless you really need to develop your own code. Furthermore, our vector class has served mainly a pedagogical scope, since C++ has a Standard Template Library (STL) with vector types, including a vector for doing numerics that can be declared as

```
std::valarray<double> x(n); // vector with n entries
```

However, there is no STL for a matrix type. We end therefore with recommending the use of ready-made libraries like Blitz++ or Armadillo or the matrix class discussed in the linear algebra chapter, see chapter 6.

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

```
Vector v1;

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

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

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

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

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

// Assign the entries in a vector:
v5(0) = 3.0; // or alternatively v5[0] = 3.0;
v5(1) = 2.5; // or alternatively v5[1] = 2.5;
v5(2) = 1.0; // or alternatively v5[2] = 1.0;

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

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

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

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

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

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

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

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

We list here the header file first.

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

```
#ifndef VECTOR_H
#define VECTOR_H

#include <cmath>
#include <iostream>

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

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

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

```
public:

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



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


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

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

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

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

  // Return pointers to the data: Useful for sending data
  // to Fortran and C
```

```
const double* getPtr() const;
double* getPtr();

double inner(const Vector&) const;

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

void print(std::ostream&) const;

/**
* Change the length of a vector
**/
bool redim(int n1);

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

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

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

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

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

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

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

const double& operator[](int i) const;
double& operator[](int i);
const double& operator()(int i) const;
double& operator()(int i);
bool indexOk(int i) const;
// Get the euclidian norm (l2norm)
double l2norm() const;
// Unary operator +
friend Vector operator+(const Vector&);   // u = + v
// Unary operator -
friend Vector operator-(const Vector&);   // u = - v
```

```
   /**
   * Addition of two vectors:
   **/
   friend Vector operator+(const Vector&, const Vector&); // u = v + w
   /**
   * Substraction of two vectors:
   **/
   friend Vector operator-(const Vector&, const Vector&); // u = v - w
   /**
   * Product between two vectors:
   **/
   friend Vector operator*(const Vector&, const Vector&); // u = v * w
    /**
   * Premultiplication by a floating point number:
   **/
   friend Vector operator*(double, const Vector&); // u = a*v
   /**
   * Postmultiplication by a floating point number:
   **/
   friend Vector operator*(const Vector&, double); // u = v*a

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

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

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

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

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

// Get the number of entries in a vector
```

```
inline int Vector::getLength() const {return length;}

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

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

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

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


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

// Subscript (DANGEROUS). If v is an object of type Vector, the ith
// component of v can be accessed as v(i) closer to the
// ordinary mathematical notation instead of v.vec(i).
```

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

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

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

#endif
```

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

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

```
#include "Vector.h"

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

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

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

// Declare the array to be constant because it is passed
// as a pointer. Hence, it could be modified by the calling code.
Vector::Vector(int _length, // length of the array
           const double *array){ // one-dimensioal array
  length = _length;
  vec = new double[length];
  for(int i=0; i<length; i++)
    vec[i] = array[i];
}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.5 Modules in Fortran

In the previous section we discussed classes and templates in C++. Classes offer several advantages, such as

- Allows us to place classes into structures
- Pass arguments to methods
- Allocate storage for objects
- Implement associations
- Encapsulate internal details into classes
- Implement inheritance in data structures

Classes contain a new data type and the procedures that can be performed by the class. The elements (or components) of the data type are the class data members, and the procedures are the class member functions. In Fortran a class is defined as a `MODULE` which contains an abstract data `TYPE` definition. The example we elaborate on here is a Fortran class for defining operations on single-particle quantum numbers such as the total angular momentum, the orbital momentum, the energy, spin etc.

We present the `MODULE single_particle_orbits` here and discuss several of its feature with links to C++ programming.

```
!    Definition of single particle data

MODULE single_particle_orbits
  TYPE, PUBLIC :: single_particle_descript
     INTEGER :: total_orbits
     INTEGER, DIMENSION(:), POINTER :: nn, ll, jj, spin
     CHARACTER*10, DIMENSION(:), POINTER :: orbit_status, &
                                         model_space
     REAL(KIND=8), DIMENSION(:), POINTER :: e
  END TYPE single_particle_descript

  TYPE (single_particle_descript), PUBLIC :: all_orbit, &
```

```
     neutron_data, proton_data
  CONTAINS

! various member functions here

  SUBROUTINE allocate_sp_array(this_array,n)
  TYPE (single_particle_descript), INTENT(INOUT) :: this_array
  INTEGER , INTENT(IN) :: n
  IF (ASSOCIATED (this_array%nn) ) &
    DEALLOCATE(this_array%nn)
  ALLOCATE(this_array%nn(n))
  IF (ASSOCIATED (this_array%ll) ) &
    DEALLOCATE(this_array%ll)
  ALLOCATE(this_array%ll(n))
  IF (ASSOCIATED (this_array%jj) ) &
    DEALLOCATE(this_array%jj)
  ALLOCATE(this_array%jj(n))
  IF (ASSOCIATED (this_array%spin) ) &
    DEALLOCATE(this_array%spin)
  ALLOCATE(this_array%spin(n))
  IF (ASSOCIATED (this_array%e) ) &
     DEALLOCATE(this_array%e)
  ALLOCATE(this_array%e(n))
  IF (ASSOCIATED (this_array%orbit_status) ) &
    DEALLOCATE(this_array%orbit_status)
    ALLOCATE(this_array%orbit_status(n))
  IF (ASSOCIATED (this_array%model_space) ) &
    DEALLOCATE(this_array%model_space)
    ALLOCATE(this_array%model_space(n))
! blank all characters and zero all other values
  DO i= 1, n
    this_array%model_space(i)= ' '
    this_array%orbit_status(i)= ' '
    this_array%e(i)=0.
    this_array%nn(i)=0
    this_array%ll(i)=0
    this_array%jj(i)=0
    this_array%nshell(i)=0
    this_array%itzp(i)=0
  ENDDO

  SUBROUTINE deallocate_sp_array(this_array)

   TYPE (single_particle_descript), INTENT(INOUT) :: this_array
   DEALLOCATE(this_array%nn)
   DEALLOCATE(this_array%ll)
   DEALLOCATE(this_array%jj)
   DEALLOCATE(this_array%spin)
   DEALLOCATE(this_array%e)
   DEALLOCATE(this_array%orbit_status); &
   DEALLOCATE(this_array%model_space)

   END SUBROUTINE deallocate_sp_array
!
!    Read in all relevant single-particle data
!
  SUBROUTINE single_particle_data
    IMPLICIT NONE
    CHARACTER*100 :: particle_species

    READ(5,*) particle_species
    WRITE(6,*) ' Particle species: '
    WRITE(6,*) particle_species
    SELECT CASE (particle_species)
      CASE ('electron')
        CALL read_electron_sp_data
      CASE ('proton&neutron')
        CALL read_nuclear_sp_data
    END SELECT

    END SUBROUTINE single_particle_data

END MODULE single_particle_orbits
```

The module ends with the `END MODULE single_particle_orbits` statement. We have defined a public variable `TYPE, PUBLIC :: single_particle_descript` which plays the same role as the `struct` type in C++. In addition we have defined several member functions which operate on various arrays and variables.

An example of a function which uses this module is given below and the module is accessed via the `USE single_particle_orbits` statement.

```
!
  PROGRAM main
  ....
  USE single_particle_orbits
  IMPLICIT NONE
  INTEGER :: i

  READ(5,*) all_orbit%total_orbits
  IF( all_orbit%total_orbits <= 0 ) THEN
     WRITE(6,*) 'WARNING, NO ELECTRON ORBITALS' ; STOP
  ENDIF
!    Setup all possible orbit information
!    Allocate space in heap for all single-particle data
  CALL allocate_sp_array(all_orbit,all_orbit%total_orbits)
!    Read electron single-particle data

  DO i=1, all_orbit%total_orbits
```

```
   READ(5,*) all_orbit%nn(i),all_orbit%ll, &
          all_orbit%jj(i),all_orbit%spin(i), &
          all_orbit%orbit_status(i), &
          all_orbit%model_space(i), all_orbit%e(i)
 ENDDO

! further instructions

 .......

! deallocate all arrays

 CALL deallocate_sp_array(all_orbit)


 END PROGRAM main
```

Inheritance allows one to create a hierarchy of classes in which the base class contains the common properties of the hierarchy and the derived classes can modify and specialize these properties. Specifically, a derived class contains all the class member functions of the base class and can add new ones. Further, a derived class contains all the class member functions of the base class and can modify them or add new ones. The value in using inheritance is to avoid duplicating code when creating classes which are similar to one another. Fortran does not support inheritance, but several features can be faked in Fortran! Consider the following declarations:

```
 TYPE proton_sp_orbit
    TYPE (single_particle_orbits), PUBLIC :: &
        proton_particle_descript
    INTEGER, DIMENSION(:), POINTER, PUBLIC :: itzp
 END TYPE proton_sp_orbit
```

To initialize the proton_sp_orbit TYPE, we could now define a new function

```
 SUBROUTINE allocate_proton_array(this_array,n)

 TYPE (single_particle_descript), INTENT(INOUT) :: this_array
 INTEGER , INTENT(IN) :: n
 IF (ASSOCIATED (this_array%itzp) ) &
   DEALLOCATE(this_array%itzp)
 CALL allocate_sp_array(this_array,n)
 this_array%itzp(i)=0

 END SUBROUTINE allocate_proton_array
```

and

```
 SUBROUTINE dellocate_proton_array(this_array)
 TYPE (single_particle_descript), INTENT(INOUT) :: this_array
 DEALLOCATE(this_array%itzp)
 CALL deallocate_sp_array(this_array)

 END SUBROUTINE deallocate_proton_array
```

and we could define a MODULE

```
 MODULE proton_class
   USE single_particle_orbits
   TYPE proton_sp_orbit
      TYPE (single_particle_orbits), PUBLIC :: &
          proton_particle_descript
      INTEGER, DIMENSION(:), POINTER, PUBLIC :: itzp
   END TYPE proton_sp_orbit
   INTERFACE allocate_proton
     MODULE PROCEDURE allocate_proton_array, read_proton_array
   END INTERFACE
   INTERFACE deallocate_proton
     MODULE PROCEDURE deallocate_proton_array
   END INTERFACE
   .....
   CONTAINS
   ....
!   various procedure

 END MODULE proton_class
```

```
  PROGRAM with_just_protons
  USE proton_class
  ....
  TYPE (proton_sp_orbit ) :: proton_data
  CALL allocate_proton(proton_data)
  ....
  CALL deallocate_proton_array(prton_data)
```

We have a written a new class which contains the data of the base class and all the procedures of the base class have been extended to work with the new derived class. Interface statements have to be used to give the procedure uniform names.

We can now derive further classes for other particle types such as neutrons, hyperons etc etc.

3.6 How to make Figures with Gnuplot

We end this chapter with a practical guide on making figures to be included in an eventual report file. **Gnuplot** is a simple plotting program which follows the Linux/Unix operating system. It is easy to use and allows also to generate figure files which can be included in a **LATEX** document. Here we show how to make simple plots online and how to make postscript versions of the plot or even a figure file which can be included in a **LATEX** document. There are other plotting programs such as **xmgrace** as well which follow Linux or Unix as operating systems. An excellent alternative which many of you are familiar with is to use Matlab to read in the data of a calculation and vizualize the results.

In order to check if gnuplot is present type

```
which gnuplot
```

If gnuplot is available, simply write

```
gnuplot
```

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

```
gnuplot>
```

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

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

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

or

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

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

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

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

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

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

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

```
gv mydata.ps
```

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

```
set terminal pslatex
set output "derivative.tex"
set xrange [-15:0]
set yrange [-10:8]
set xlabel "log$_{10}(h)$"
set ylabel "$\epsilon$"
plot "out.dat"  title "Relative error" w l
```

generates a LaTeX file **derivative.tex**. Alternatively, you could write the above commands in a file **derivative.gnu** and use **Gnuplot** as follows

```
gnuplot>load 'derivative.gnu'
```

You can then include this file in a LaTeX document as shown here

```
\begin{figure}
   \begin{center}
      \input{derivative}
   \end{center}
   \caption{Log-log plot of the relative error of the second
            derivative of $e^x$ as function of decreasing step
            lengths $h$. The second derivative was computed for
            $x=10$ in the program discussed above. See text for
            further details\label{fig:lossofprecision}}
 \end{figure}
```

Most figures included in this text have been generated using gnuplot.

Many of the above commands can all be baked in a Python code. The following example reads a file from screen with x and y data, and plots these data and saves the result as a postscript figure.

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

import sys
from Numeric import *
```

```
import Gnuplot

g = Gnuplot.Gnuplot(persist=1)

try:
   infilename = sys.argv[1]
except:
   print "Usage of this script", sys.argv[0], "infile", sys.argv[1]; sys.exit(1)
# Read file with data
ifile = open(infilename, 'r')
# Fill in x and y
x = [] ; y = []
for line in ifile:
   pair = line.split()
   x = float(pair[0]); y = float(pair[1])
ifile.close()
# convert to a form that the gnuplot interface can deal with
d = Gnuplot.Data(x, y, title='data from output file', with='lp')
g.xlabel('log10(h)') # make x label
g.ylabel('log10(|Exact-Computed|)/|Exact|')
g.plot(d)                # plot the data
g.hardcopy(filename="relerror.ps",terminal="postscript", enhanced=1, color=1)
```

3.7 Exercises

Computing derivatives numerically

We want you to compute the first derivative of

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

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

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

and

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

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

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

2. Make thereafter a program which computes the first derivative using Eqs. (3.14) and (3.15) as function of various step lengths h and let $h \to 0$. Compare with the exact answer.

   Your program should contain the following elements:

   - A vector (array) which contains the step lengths. Use dynamic memory allocation.
   - Vectors for the computed derivatives of Eqs. (3.14) and (3.15) for both single and double precision.
   - A function which computes the derivative and contains call by value and reference (for C++ users only).
   - Add a function which writes the results to file.

3. Compute thereafter

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

   as function of $log_{10}(h)$ for Eqs. (3.14) and (3.15) for both single and double precision. Plot the results and see if you can determine empirically the behavior of the total error as function of h.

prob 3.2: C++ class

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

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

C++ class

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

C++ class

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

C++ class

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

C++ class

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

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

C++ class

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

C++ class

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

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

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

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

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

where $p \geq 1$.

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

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

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

Bibliography

Chapter 4

Non-linear Equations

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

4.2 Particle in a Box Potential

You may have encountered examples of so-called transcendental equations when solving the Schrödinger equation (SE) for a particle in a box potential. The one-dimensional SE for a particle with mass m is

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

and our potential is defined as

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

Figure 4.1: Plot of $f(E)$ in Eq. (4.2) as function of energy |E| in MeV. Te function $f(E)$ is in units of megaelectronvolts MeV. Note well that the energy E is for bound states.

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)

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

and

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

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

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

and

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

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

$$\sqrt{2m(V_0 - |E|)}\cot(\sqrt{2ma^2(V_0 - |E|)}/\hbar) = -\sqrt{2m|E|}. \tag{4.1}$$

This equation is an example of the kind of equations which could be solved by some of the methods discussed below. The algorithms we discuss are the bisection method, the secant and Newton-Raphson's method.

In order to find the solution for Eq. (4.1), a simple procedure is to define a function

$$f(E) = \sqrt{2m(V_0 - |E|)}\cot(\sqrt{2ma^2(V_0 - |E|)}/\hbar) + \sqrt{2m|E|}. \tag{4.2}$$

and with chosen or given values for a and V_0 make a plot of this function and find the approximate region along the $E - axis$ where $f(E) = 0$. We show this in Fig. 4.1 for $V_0 = 20$ MeV, $a = 2$ fm and $m = 938$ MeV. Fig. 4.1 tells us that the solution is close to $|E| \approx 2.2$ (the binding energy of the deuteron). The methods we discuss below are then meant to give us a numerical solution for E where $f(E) = 0$ is satisfied and with E determined by a given numerical precision.

4.3 Iterative Methods

To solve an equation of the type $f(x) = 0$ means mathematically to find all numbers s[1] so that $f(s) = 0$. In all actual calculations we are always limited by a given precision

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

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

$$|x_0 - s| < \varepsilon,$$

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

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

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

$$|f(x_1) - f(x_2)| \le k\,|x_1 - x_2|\,,$$

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

$$f(x_1) - f(x_2) = f'(\xi)(x_1 - x_2),$$

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

$$|f(x_1) - f(x_2)| \le |f'(\xi)|\,|x_1 - x_2|\,.$$

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

1. We assume that f is defined in the interval $[a,b]$.

2. f satisfies the Lipschitz condition with $k < 1$.

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

$$|s - x_n| \le \frac{k}{1-k}\,|x_1 - x_2|\,. \tag{4.3}$$

The proof can be found in the text of Bulirsch and Stoer. Since it is difficult numerically to find exactly the point where $f(s) = 0$, in the actual numerical solution one implements three tests of the type

1. $$|x_n - s| < \varepsilon,$$

   and

2.
$$|f(s)| < \delta,$$

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

4.4 Bisection

This is an extremely simple method to code. The philosophy can best be explained by choosing a region in e.g., Fig. 4.1 which is close to where $f(E) = 0$. In our case $|E| \approx 2.2$. Choose a region $[a, b]$ so that $a = 1.5$ and $b = 3$. This should encompass the point where $f = 0$. Define then the point

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

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

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

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

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

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

and if we set $x_0=(a+b)/2$ and let x_n be the midpoints in the intervals we obtain after n iterations that Eq. (4.3) results in

$$|s-x_n| \le \frac{1}{2^{n+1}}|b-a|, \tag{4.4}$$

since the nth interval has length $|b-a|/2^n$. Note that this convergence criterion is independent of the actual function $f(x)$ as long as this function fulfils the conditions discussed in the conditions discussed in the previous subsection.

As an example, suppose we wish to find how many iteration steps are needed in order to obtain a relative precision of 10^{-12} for x_n in the interval $[50,63]$, that is

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

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

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

and with Eq. (4.4) we obtain

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

meaning $n \ge 37$. The code for the bisection method can look like this

```
    /*
    ** This function
    ** calculates a root between x1 and x2 of a function
    ** pointed to by (*func) using the method of bisection
    ** The root is returned with an accuracy of +- xacc.
    */

double bisection(double (*func)(double), double x1, double x2, double xacc)
{
  int     j;
  double  dx, f, fmid, xmid, rtb;

  f    = (*func)(x1);
  fmid = (*func)(x2);
  if(f*fmid >= 0.0) {
    cout << "\n\nError in function bisection():" << endl;
    cout << "\nroot in function must be within" << endl;
    cout << "x1 ='' << x1 << ``and x2 `` << x2 << endl;
    exit(1);
  }
  rtb = f < 0.0 ? (dx = x2 - x1, x1) : (dx = x1 - x2, x2);
  for(j = 0; j < max_iterations; j++) {
    fmid = (*func)(xmid = rtb + (dx *= 0.5));
    if (fmid <= 0.0) rtb=xmid;
    if(fabs(dx) < xacc || fmid == 0.0) return rtb;
  }
  cout << "Error in the bisection:" << endl; // should never reach this point
  cout "Too many iterations!" << endl;
}
// End: function bisection
```

In this function we transfer the lower and upper limit of the interval where we seek the solution, $[x_1,x_2]$. The variable `xacc` is the precision we opt for. Note that in this function the test $f(s) < \delta$ is not implemented. Rather, the test is done through $f(s) = 0$, which is not necessarily a good option.

Note also that this function transfer a pointer to the name of the given function through `double(*func)(double)`.

4.5 Newton-Raphson's Method

Perhaps the most celebrated of all one-dimensional root-finding routines is Newton's method, also called the Newton-Raphson method. This method is distinguished from the previously discussed methods by the fact that it requires the evaluation of both the function f and its derivative f' at arbitrary points. In this sense, it is taylored to cases with e.g., transcendental equations of the type shown in Eq. (4.2) where it is rather easy to evaluate the derivative. If you can only calculate the derivative numerically and/or your function is not of the smooth type, we discourage the use of this method.

The Newton-Raphson formula consists geometrically of extending the tangent line at a current point until it crosses zero, then setting the next guess to the abscissa of that zero-crossing. The mathematics behind this method is rather simple. Employing a Taylor expansion for x sufficiently close to the solution s, we have

$$f(s) = 0 = f(x) + (s-x)f'(x) + \frac{(s-x)^2}{2}f''(x) + \dots. \tag{4.5}$$

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

$$f(x) + (s-x)f'(x) \approx 0,$$

Figure 4.2: Example of a case where Newton-Raphson's method does not converge. For the function $f(x) = x - 2cos(x)$, we see that if we start at $x = 7$, the first iteration gives us that the first point where we cross the $x-$axis is given by x_1. However, using x_1 as a starting point for the next iteration results in a point x_2 which is close to a local minimum. The tangent here is close to zero and we will never approach the point where $f(x) = 0$.

yielding

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

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

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

This is Newton-Raphson's method. It has a simple geometric interpretation, namely x_{n+1} is the point where the tangent from $(x_n, f(x_n))$ crosses the $x-$axis. Close to the solution, Newton-Raphson converges fast to the desired result. However, if we are far from a root, where the higher-order terms in the series are important, the Newton-Raphson formula can give grossly inaccurate results. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function. If an iteration places a trial guess near such a local extremum, so that the first derivative nearly vanishes, then Newton-Raphson may fail totally. An example is shown in Fig. 4.2

It is also possible to extract the convergence behavior of this method. Assume that the function f has a continuous second derivative around the solution s. If we define

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

and using Eq. (4.5) we have

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

This gives

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

when $x_n \to s$. Our error constant k is then proportional to $|f''(s)|/|f'(s)|^2$ if the second derivative is different from zero. Clearly, if the first derivative is small, the convergence is slower. In general, if we are able to start the iterative procedure near a root and we can easily evaluate the derivative, this is the method of choice. In cases where we may need to evaluate the derivative numerically, the previously described methods are easier and most likely safer to implement with respect to loss of numerical precision. Recall that the numerical evaluation of derivatives involves differences between function values at different x_n.

We can rewrite the last equation as

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

with C a constant. If we assume that $C \sim 1$ and let $e_n \sim 10^{-8}$, this results in $e_{n+1} \sim 10^{-16}$, and demonstrates clearly why Newton-Raphson's method may converge faster than the bisection method.

Summarizing, this method has a solution when f'' is continuous and s is a simple zero of f. Then there is a neighborhood of s and a constant C such that if Newton-Raphson's method is started in that neighborhood, the successive points become steadily closer to s and satisfy

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

with $n \geq 0$. In some situations, the method guarantees to converge to a desired solution from an arbitrary starting point. In order for this to take place, the function f has to belong to $C^2(R)$, be increasing, convex and having a zero. Then this zero is unique and Newton's method converges to it from any starting point.

As a mere curiosity, suppose we wish to compute the square root of a number R, i.e., $\sqrt{R}$. Let $R > 0$ and define a function

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

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

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

a formula credited to Heron, a Greek engineer and architect who lived sometime between 100 B.C. and A.D. 100.

Suppose we wish to compute $\sqrt{13} = 3.6055513$ and start with $x_0 = 5$. The first iteration gives $x_1 = 3.8$, $x_2 = 3.6105263$, $x_3 = 3.6055547$ and $x_4 = 3.6055513$. With just four iterations and a not too optimal choice of x_0 we obtain the exact root to a precision of 8 digits. The above equation, together with range reduction , is used in the intrisic computational function which computes square roots.

Newton's method can be generalized to systems of several non-linear equations and variables. Consider the case with two equations

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

which we Taylor expand to obtain

$$\begin{array}{rl} 0 = f_1(x_1+h_1,x_2+h_2) = & f_1(x_1,x_2)+h_1\partial f_1/\partial x_1+h_2\partial f_1/\partial x_2+\dots \\ 0 = f_2(x_1+h_1,x_2+h_2) = & f_2(x_1,x_2)+h_1\partial f_2/\partial x_1+h_2\partial f_2/\partial x_2+\dots \end{array}.$$

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

$$\hat{\mathbf{J}} = \left(\begin{array}{cc} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{array} \right),$$

we can rephrase Newton's method as

$$\left(\begin{array}{c} x_1^{n+1} \\ x_2^{n+1} \end{array} \right) = \left(\begin{array}{c} x_1^{n} \\ x_2^{n} \end{array} \right) + \left(\begin{array}{c} h_1^{n} \\ h_2^{n} \end{array} \right),$$

where we have defined

$$\left(\begin{array}{c} h_1^{n} \\ h_2^{n} \end{array} \right) = -\hat{\mathbf{J}}^{-1} \left(\begin{array}{c} f_1(x_1^n,x_2^n) \\ f_2(x_1^n,x_2^n) \end{array} \right).$$

We need thus to compute the inverse of the Jacobian matrix and it is to understand that difficulties may arise in case $\hat{\mathbf{J}}$ is nearly singular.

It is rather straightforward to extend the above scheme to systems of more than two non-linear equations.

The code for Newton-Raphson's method can look like this

```
    /*
    ** This function
    ** calculates a root between x1 and x2 of a function pointed to
    ** by (*funcd) using the Newton-Raphson method. The user-defined
    ** function funcd() returns both the function value and its first
    ** derivative at the point x,
    ** The root is returned with an accuracy of +- xacc.
    */

double newtonraphson(void (*funcd)(double, double *, double *), double x1, double x2,
 double xacc)
{
  int   j;
  double df, dx, f, rtn;

  rtn = 0.5 * (x1 + x2);       // initial guess
  for(j = 0; j < max_iterations; j++) {
    (*funcd)(rtn, &f, &df);
    dx  = f/df;
    rtn -= dx;
    if((x1 - rtn) * (rtn - x2) < 0.0) {
      cout << "\n\nError in function newtonraphson:" << endl ;
      cout << "Jump out of interval bracket" << endl;
```

Figure 4.3: Plot of $f(E)$ Eq. (4.2) 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 $x-axis$.

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

We transfer again the lower and upper limit of the interval where we seek the solution, $[x_1, x_2]$ and the variable `xacc`. Firthermore, it transfers a pointer to the name of the given function through `double(*func)(double)`.

4.6 The Secant Method

For functions that are smooth near a root, the methods known respectively as false position (or regula falsi) and secant method generally converge faster than bisection but slower than Newton-Raphson. In both of these methods the function is assumed to be approximately linear in the local region of interest, and the next improvement in the root is taken as the point where the approximating line crosses the axis.

The algorithm for obtaining the solution for the secant method is rather simple. We start with the definition of the derivative

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

and combine it with the iterative expression of Newton-Raphson's

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

to obtain

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

which we rewrite to

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

This is the secant formula, implying that we are drawing a straight line from the point $(x_{n-1}, f(x_{n-1}))$ to $(x_n, f(x_n))$. Where it crosses the $x-axis$ we have the new point x_{n+1}. This is illustrated in Fig. 4.3.

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

In the numerical implementation found in the program library, the quantities x_{n-1}, x_n, x_{n+1} are changed to a, b and c respectively, i.e., we determine c by the point where a straight line from the point $(a, f(a))$ to $(b, f(b))$ crosses the $x-axis$, that is

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

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

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

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. 4.4 where we show the results of the first three iterations, i.e., the first point is $c = x_1$, the next iteration gives $c = x_2$ while the third iterations ends with $c = x_3$. We may risk that one of the endpoints is kept fixed while the other one only slowly converges to the desired solution.

The search for the solution s proceeds in much of the same fashion as for the bisection method, namely after each iteration one of the previous boundary points is discarded in favor of the latest estimate of the root. A variation of the secant method is the so-called false position method (regula falsi from Latin) where the interval [a,b] is chosen so that $f(a)f(b) < 0$, else there is no solution. This is rather similar to the bisection method. Another possibility is to determine the starting point for the iterative search using three points $(a, f(a))$, $(b, f(b))$ and $(c, f(c))$. One can thenuse Lagrange's interpolation formula for a polynomial, see the discussion in the previous chapter.

4.6.1 Broyden's Method

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

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

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

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

and proceeds using Newton's method

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

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

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

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

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

and then apply Newton's method

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

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

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

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

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

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

4.7 Exercises

Comparison of methods

Write a code which implements the bisection method, Newton-Raphson's method and the secant method.

Find the positive roots of

$$x^2 - 4x\sin x + (2\sin x)^2 = 0,$$

using these three methods and compare the achieved accuracy number of iterations needed to find the solution. Give a critical discussion of the methods.

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

Schrödinger's equation

We are going to study the solution of the Schrödinger equation (SE) for a system with a neutron and proton (the deuteron) moving in a simple box potential.

We begin our discussion of the SE with the neutron-proton (deuteron) system with a box potential $V(r)$. We define the radial part of the wave function $R(r)$ and introduce the definition $u(r) = rR(R)$ The radial part of the SE for two particles in their center-of-mass system and with orbital momentum $l = 0$ is then

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

with

$$m = 2\frac{m_p m_n}{m_p + m_n},$$

where m_p and m_n are the masses of the proton and neutron, respectively. We use here $m = 938$ MeV. Our potential is defined as

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

Bound states correspond to negative energy E and scattering states are given by positive energies. The SE takes the form (without specifying the sign of E)

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

and

$$\frac{d^2u(r)}{dr^2} + \frac{m}{\hbar^2}Eu(r) = 0 \quad r > a.$$

We are now going to search for eventual bound states, i.e., $E < 0$. The deuteron has only one bound state at energy $E = -2.223$ MeV. Discuss the boundary conditions on the wave function and use these to show that the solution to the SE is

$$u(r) = Asin(kr) \qquad r < a,$$

and

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

where A and B are constants. We have also defined

$$k = \sqrt{m(V_0 - |E|)}/\hbar,$$

and

$$\beta = \sqrt{m|E|}/\hbar.$$

Show then, using the continuity requirement on the wave function that at $r = a$ you obtain the transcendental equation

$$kcot(ka) = -\beta. \tag{4.6}$$

Insert values of $V_0 = 60$ MeV and $a = 1.45$ fm (1 fm = 10^{-15} m) and make a plot plotting programs) of Eq. (4.6) as function of energy E in order to find eventual eigenvalues. See if these values result in a bound state for E.

When you have localized on your plot the point(s) where Eq. (4.6) is satisfied, obtain a numerical value for E using the class you programmed in the previous exercise, including the Newton-Raphson's method, the bisection method and the secant method. Make an analysis of these three methods and discuss how many iterations are needed to find a stable solution.

What is smallest possible value of V_0 which gives a bound state?

Bibliography

Chapter 5

Numerical Integration

5.1 Introduction

In this chapter we discuss some of the classical methods for integrating a function. The methods we discuss are the trapezoidal, rectangular and Simpson's rule for equally spaced abscissas and integration approaches based on Gaussian quadrature. The latter are more suitable for the case where the abscissas are not equally spaced. The emphasis is on methods for evaluating few-dimensional (typically up to four dimensions) integrals. In chapter **??** we show how Monte Carlo methods can be used to compute multi-dimensional integrals. We discuss also how to compute singular integrals. We end this chapter with an extensive discussion on MPI and parallel computing. The examples focus on parallelization of algorithms for computing integrals.

5.2 Newton-Cotes Quadrature

The integral I=$\int_a^b f(x) dxhasaverysimplemeaning.IfweconsiderFig.$**??**$theintegral$I$simplyrepresentstheare$
$calledGaussianquadraturemethods.Bothmainmethodsencompassaplethoraofapproximationsandonlysomec$

In considering equal step methods, our basic approach is that of approximating a function $f(x)$ with a polynomial of at most degree $N-1$, given N integration points. If our polynomial is of degree 1, the function will be approximated with $f(x) \approx a_0 + a_1 x$. The algorithm for these integration methods is rather simple, and the number of approximations perhaps unlimited!

- Choose a step size
$$h = \frac{b-a}{N}$$
where N is the number of steps and a and b the lower and upper limits of integration.

- With a given step length we rewrite the integral as

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

- The strategy then is to find a reliable polynomial approximation for $f(x)$ in the various intervals. Choosing a given approximation for $f(x)$, we obtain a specific approximation to the integral.

- With this approximation to $f(x)$ we perform the integration by computing the integrals over all subintervals.

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

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

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

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

which we rewrite as $\int_a^{a+2h} f(x)dx = \int_{x_0-h}^{x_0+h} f(x)dx. We have chosen a midpoint$ x$_0$ and have defined $x_0 = a+h$. Using Lagrange's interpolation formula from Eq. (3.9), an equation we restate here,

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

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

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

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

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

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

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

we can easily calculate for example the second integral as

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

which can be simplified to

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

resulting in

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

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

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

and adding up we obtain $\int_{x_0-h}^{x_0+h} f(x)dx = \frac{h}{2}\left(f(x_0+h)+2f(x_0)+f(x_0-h)\right) + O(h^3), whichisthewell-knowntrapezoidalrule.Concerningtheerrorintheapproximationmade,$O(h^3) $= O((b-a)^3/N^3)$, you should note the following. *This is the local error!* Since we are splitting the integral from a to b in N pieces, we will have to perform approximately N such operations. This means that the *global error* goes like $\approx O(h^2)$. To see that, we use the trapezoidal rule to compute the integral of Eq. (5.2),

$$I = \int_a^b f(x)dx = h\left(f(a)/2 + f(a+h) + f(a+2h) + \cdots + f(b-h) + f_b/2\right), \tag{5.1}$$

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

Hereafter we use the shorthand notations $f_{-h} = f(x_0-h)$, $f_0 = f(x_0)$ and $f_h = f(x_0+h)$. The correct mathematical expression for the local error for the trapezoidal rule is

$$\int_a^b f(x)dx - \frac{b-a}{2}\left[f(a)+f(b)\right] = -\frac{h^3}{12} f^{(2)}(\xi),$$

and the global error reads

$$\int_a^b f(x)dx - T_h(f) = -\frac{b-a}{12} h^2 f^{(2)}(\xi),$$

where T_h is the trapezoidal result and $\xi \in [a,b]$.

The trapezoidal rule is easy to implement numerically through the following simple algorithm

- Choose the number of mesh points and fix the step.

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

A simple function which implements this algorithm is as follows

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

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

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

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

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

```
 trapezoidal_rule(a, b, n, &MyFunction )
```

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

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

Another very simple approach is the so-called midpoint or rectangle method. In this case the integration area is split in a given number of rectangles with length h and height given by the mid-point value of the function. This gives the following simple rule for approximating an integral

$$I = \int_a^b f(x)dx \approx h\sum_{i=1}^{N} f(x_{i-1/2}), \tag{5.2}$$

where $f(x_{i-1/2})$ is the midpoint value of f for a given rectangle. We will discuss its truncation error below. It is easy to implement this algorithm, as shown here

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

```
double rectangle_rule(double a, double b, int n, double (*func)(double))
{
    double rectangle_sum;
    double fa, fb, x, step;
    int  j;
    step=(b-a)/((double) n);
    rectangle_sum=0.;
    for (j = 0; j <= n; j++){
      x = (j+0.5)*step+; // midpoint of a given rectangle
      rectangle_sum+=(*func)(x); // add value of function.
    }
    rectangle_sum *= step; // multiply with step length.
    return rectangle_sum;
} // end rectangle_rule
```

The correct mathematical expression for the local error for the rectangular rule $R_i(h)$ for element i is

$$\int_{-h}^{h} f(x)dx - R_i(h) = -\frac{h^3}{24}f^{(2)}(\xi),$$

and the global error reads

$$\int_a^b f(x)dx - R_h(f) = -\frac{b-a}{24}h^2 f^{(2)}(\xi),$$

where R_h is the result obtained with rectangular rule and $\xi \in [a,b]$.

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

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

Using again Lagrange's interpolation formula we have

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

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

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

which is Simpson's rule. Note that the improved accuracy in the evaluation of the derivatives gives a better error approximation, $O(h^5)$ vs. $O(h^3)$. But this is again the *local error approximation*. Using Simpson's rule we can easily compute the integral of Eq. (5.2) to be

$$I = \int_a^b f(x)dx = \frac{h}{3}(f(a) + 4f(a+h) + 2f(a+2h) + \cdots + 4f(b-h) + f_b), \tag{5.3}$$

with a global error which goes like $O(h^4)$. More formal expressions for the local and global errors are for the local error

$$\int_a^b f(x)dx - \frac{b-a}{6}[f(a) + 4f((a+b)/2) + f(b)] = -\frac{h^5}{90}f^{(4)}(\xi),$$

and for the global error

$$\int_a^b f(x)dx - S_h(f) = -\frac{b-a}{180}h^4 f^{(4)}(\xi).$$

with $\xi \in [a,b]$ and S_h the results obtained with Simpson's method. The method can easily be implemented numerically through the following simple algorithm

- Choose the number of mesh points and fix the step.
- calculate $f(a)$ and $f(b)$
- Perform a loop over $n = 1$ to $n-1$ ($f(a)$ and $f(b)$ are known) and sum up the terms $4f(a+h) + 2f(a+2h) + 4f(a+3h) + \cdots + 4f(b-h)$. Each step in the loop corresponds to a given value $a+nh$. Odd values of n give 4 as factor while even values yield 2 as factor.
- Multiply the final result by $\frac{h}{3}$.

In more general terms, what we have done here is to approximate a given function $f(x)$ with a polynomial of a certain degree. One can show that given $n+1$ distinct points $x_0, \dots, x_n \in [a,b]$ and $n+1$ values $y_0, \dots, y_n$ there exists a unique polynomial $P_n(x)$ with the property

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

In the Lagrange representation discussed in chapter 3, this interpolating polynomial is given by

$$P_n = \sum_{k=0}^{n} l_k y_k,$$

with the Lagrange factors

$$l_k(x) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i} \quad k = 0, \dots, n,$$

see for example the text of Kress [3] or Burlich and Stoer [5] for details. If we for example set $n = 1$, we obtain

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

which we recognize as the equation for a straight line.

The polynomial interpolatory quadrature of order n with equidistant quadrature points $x_k = a + kh$ and step $h = (b-a)/n$ is called the Newton-Cotes quadrature formula of order n. General expressions can be found in for example Refs. [3, 5].

5.3 Adaptive Integration

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

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

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

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

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

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

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

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

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

  and

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

  With a given defined tolerance, being a small number provided by us, we estimate the difference $|I^{(1L)} + I^{(1R)} - I^{(0)}| <$ tolerance. If this test is satisfied, our first approximation is satisfactory.

- If not, we can set up a recursive procedure where the integral is split into subsequent subintervals until our tolerance is satisfied.

This recursive procedure can be easily implemented via the following function

```
//   Simple recursive function that implements the
//   adaptive integration using the trapezoidal rule
//   It is convenient to define as global variables
//   the tolerance and the number of recursive steps
const int maxrecursions = 50;
const double tolerance = 1.0E-10;
// Takes as input the integration limits, number of points, function to integrate
// and the number of steps
void adaptive_integration(double a, double b, double *Integral, int n, int steps,
    double (*func)(double))
    if ( steps > maxrecursions){
      cout << 'Too many recursive steps, the function varies too much' << endl;
      break;
    }
    double c = (a+b)*0.5;
    // the whole integral
    double I0 = trapezoidal_rule(a, b,n, func);
    // the left half
    double I1L = trapezoidal_rule(a, c,n, func);
    // the right half
    double I1R = trapezoidal_rule(c, b,n, func);
    if (fabs(I1L+I1R-I0) < tolerance ) integral = I0;
    else
    {
      adaptive_integration(a, c, integral, int n, ++steps, func)
      adaptive_integration(c, b, integral, int n, ++steps, func)
    }
}
```

```
// end function adaptive_integration
```

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

5.4 Gaussian Quadrature

The methods we have presented hitherto are taylored to problems where the mesh points x_i are equidistantly spaced, x_i differing from x_{i+1} by the step h. These methods are well suited to cases where the integrand may vary strongly over a certain region or if we integrate over the solution of a differential equation.

If however our integrand varies only slowly over a large interval, then the methods we have discussed may only slowly converge towards a chosen precision[1]. As an example,

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

may converge very slowly to a given precision if b is large and/or $f(x)$ varies slowly as function of x at large values. One can obviously rewrite such an integral by changing variables to $t = 1/x$ resulting in

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

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

However, there are cases where no trick may help and where the time expenditure in evaluating an integral is of importance. For such cases we would like to recommend methods based on Gaussian quadrature. Here one can catch at least two birds with a stone, namely, increased precision and fewer integration points. But it is important that the integrand varies smoothly over the interval, else we have to revert to splitting the interval into many small subintervals and the gain achieved may be lost.

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

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

where ω and x are the weights and the chosen mesh points, respectively. In our previous discussion, these mesh points were fixed at the beginning, by choosing a

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

given number of points N. The weigths ω resulted then from the integration method we applied. Simpson's rule, see Eq. (5.3) would give

$$\omega : \{h/3, 4h/3, 2h/3, 4h/3, \dots, 4h/3, h/3\},$$

for the weights, while the trapezoidal rule resulted in

$$\omega : \{h/2, h, h, \dots, h, h/2\}.$$

In general, an integration formula which is based on a Taylor series using N points, will integrate exactly a polynomial P of degree $N-1$. That is, the N weights ω_n can be chosen to satisfy N linear equations, see chapter 3 of Ref. [3]. A greater precision for a given amount of numerical work can be achieved if we are willing to give up the requirement of equally spaced integration points. In Gaussian quadrature (hereafter GQ), both the mesh points and the weights are to be determined. The points will not be equally spaced[2]. The theory behind GQ is to obtain an arbitrary weight ω through the use of so-called orthogonal polynomials. These polynomials are orthogonal in some interval say e.g., [-1,1]. Our points x_i are chosen in some optimal sense subject only to the constraint that they should lie in this interval. Together with the weights we have then $2N$ (N the number of points) parameters at our disposal.

Even though the integrand is not smooth, we could render it smooth by extracting from it the weight function of an orthogonal polynomial, i.e., we are rewriting I=$\int_a^b f(x)dx = \int_a^b W(x)g(x)dx \approx \sum_{i=1}^N \omega_i g(x_i), where$g$is smooth and$W$is the weight function, which is to be associ$

The weight function W is non-negative in the integration interval $x \in [a,b]$ such that for any $n \geq 0$, the integral $\int_a^b |x|^n W(x)dx$ is integrable. The naming weight function arises from the fact that it may be used to give more emphasis to one part of the interval than another. A quadrature formula $\int_a^b W(x)f(x)dx \approx \sum_{i=1}^N \omega_i f(x_i), with$N$distinct quadrature poin$ P_{2N-1} exactly, that is $\int_a^b W(x)p(x)dx = \sum_{i=1}^N \omega_i p(x_i), It is assumed that$W(x)$is continuous and positive and that$ Note that the replacement of $f \rightarrow Wg$ is normally a better approximation due to the fact that we may isolate possible singularities of W and its derivatives at the endpoints of the interval.

The quadrature weights or just weights (not to be confused with the weight function) are positive and the sequence of Gaussian quadrature formulae is convergent if the sequence Q_N of quadrature formulae

$$Q_N(f) \rightarrow Q(f) = \int_a^b f(x)dx,$$

in the limit $N \rightarrow \infty$. Then we say that the sequence

$$Q_N(f) = \sum_{i=1}^N \omega_i^{(N)} f(x_i^{(N)}),$$

[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.

is convergent for all polynomials p, that is

$$Q_N(p) = Q(p)$$

if there exits a constant C such that

$$\sum_{i=1}^{N} |\omega_i^{(N)}| \leq C,$$

for all N which are natural numbers.

The error for the Gaussian quadrature formulae of order N is given by

$$\int_a^b W(x)f(x)dx - \sum_{k=1}^{N} w_k f(x_k) = \frac{f^{2N}(\xi)}{(2N)!}\int_a^b W(x)[q_N(x)]^2 dx$$

where q_N is the chosen orthogonal polynomial and ξ is a number in the interval $[a,b]$. We have assumed that $f \in C^{2N}[a,b]$, viz. the space of all real or complex $2N$ times continuously differentiable functions.

In science there are several important orthogonal polynomials which arise from the solution of differential equations. Well-known examples are the Legendre, Hermite, Laguerre and Chebyshev polynomials. They have the following weight functions

Weight function	Interval	Polynomial
$W(x) = 1$	$x \in [-1,1]$	Legendre
$W(x) = e^{-x^2}$	$-\infty \leq x \leq \infty$	Hermite
$W(x) = x^{\alpha}e^{-x}$	$0 \leq x \leq \infty$	Laguerre
$W(x) = 1/(\sqrt{1-x^2})$	$-1 \leq x \leq 1$	Chebyshev

The importance of the use of orthogonal polynomials in the evaluation of integrals can be summarized as follows.

- As stated above, methods based on Taylor series using N points will integrate exactly a polynomial P of degree $N-1$. If a function $f(x)$ can be approximated with a polynomial of degree $N-1$

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

  with N mesh points we should be able to integrate exactly the polynomial P_{N-1}.

- Gaussian quadrature methods promise more than this. We can get a better polynomial approximation with order greater than N to $f(x)$ and still get away with only N mesh points. More precisely, we approximate

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

and with only N mesh points these methods promise that

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

The reason why we can represent a function $f(x)$ with a polynomial of degree $2N-1$ is due to the fact that we have $2N$ equations, N for the mesh points and N for the weights.

The mesh points are the zeros of the chosen orthogonal polynomial of order N, and the weights are determined from the inverse of a matrix. An orthogonal polynomials of degree N defined in an interval $[a,b]$ has precisely N distinct zeros on the open interval (a,b).

Before we detail how to obtain mesh points and weights with orthogonal polynomials, let us revisit some features of orthogonal polynomials by specializing to Legendre polynomials. In the text below, we reserve hereafter the labelling L_N for a Legendre polynomial of order N, while P_N is an arbitrary polynomial of order N. These polynomials form then the basis for the Gauss-Legendre method.

5.4.1 Orthogonal polynomials, Legendre

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

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

Here C is a constant. For $m_l = 0$ we obtain the Legendre polynomials as solutions, whereas $m_l \neq 0$ yields the so-called associated Legendre polynomials. This differential equation arises in for example the solution of the angular dependence of Schrödinger's equation with spherically symmetric potentials such as the Coulomb potential.

The corresponding polynomials P are

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

which, up to a factor, are the Legendre polynomials L_k. The latter fulfil the orthogonality relation $\int_{-1}^{1} L_i(x)L_j(x)dx = \frac{2}{2i+1}\delta_{ij}, andtherecursionrelation (j+1)L_{j+1}(x) + jL_{j-1}(x) - (2j+1)xL_j(x) = 0$.

It is common to choose the normalization condition

$$L_N(1) = 1.$$

With these equations we can determine a Legendre polynomial of arbitrary order with input polynomials of order $N-1$ and $N-2$.

As an example, consider the determination of L_0, L_1 and L_2. We have that

$$L_0(x) = c,$$

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

$$L_0(x) = 1.$$

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

$$L_1(x) = a + bx,$$

and using the orthogonality relation

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

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

$$L_1(x) = x.$$

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

$$L_2(x) = a + bx + cx^2,$$

using the orthogonality relations

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

and

$$\int_{-1}^{1} L_1(x)L_2(x)dx = 0,$$

and the condition $L_2(1) = 1$ we would get $\mathrm{L}_2(x) = \frac{1}{2}\left(3x^2 - 1\right)$.

We note that we have three equations to determine the three coefficients a, b and c.

Alternatively, we could have employed the recursion relation of Eq. (5.4.1), resulting in

$$2L_2(x) = 3xL_1(x) - L_0,$$

which leads to Eq. (5.4.1).

The orthogonality relation above is important in our discussion on how to obtain the weights and mesh points. Suppose we have an arbitrary polynomial Q_{N-1} of order $N-1$ and a Legendre polynomial $L_N(x)$ of order N. We could represent Q_{N-1} by the Legendre polynomials through $Q_{N-1}(x) = \sum_{k=0}^{N-1} \alpha_k L_k(x), where \alpha_k$'s are constants.

Using the orthogonality relation of Eq. (5.4.1) we see that $\int_{-1}^{1} L_N(x) Q_{N-1}(x) dx = \sum_{k=0}^{N-1} \int_{-1}^{1} L_N(x) \alpha_k L_k(x) dx = 0. We will use this result in our construction of mesh points and weights in the next subse$

In summary, the first few Legendre polynomials are

$$L_0(x) = 1,$$

$$L_1(x) = x,$$

$$L_2(x) = (3x^2 - 1)/2,$$

$$L_3(x) = (5x^3 - 3x)/2,$$

and

$$L_4(x) = (35x^4 - 30x^2 + 3)/8.$$

The following simple function implements the above recursion relation of Eq. (5.4.1). for computing Legendre polynomials of order N.

```
// This function computes the Legendre polynomial of degree N

double Legendre( int n, double x)
{
     double r, s, t;
     int m;
     r = 0; s = 1.;
     // Use recursion relation to generate p1 and p2
     for (m=0; m < n; m++ )
     {
       t = r; r = s;
       s = (2*m+1)*x*r - m*t;
       s /= (m+1);
  } // end of do loop
      return s;
}  // end of function Legendre
```

The variable s represents $L_{j+1}(x)$, while r holds $L_j(x)$ and t the value $L_{j-1}(x)$.

5.4.2 Integration points and weights with orthogonal polynomials

To understand how the weights and the mesh points are generated, we define first a polynomial of degree $2N-1$ (since we have $2N$ variables at hand, the mesh points and weights for N points). This polynomial can be represented through polynomial division by

$$P_{2N-1}(x) = L_N(x) P_{N-1}(x) + Q_{N-1}(x),$$

where $P_{N-1}(x)$ and $Q_{N-1}(x)$ are some polynomials of degree $N-1$ or less. The function $L_N(x)$ is a Legendre polynomial of order N.

Recall that we wanted to approximate an arbitrary function $f(x)$ with a polynomial P_{2N-1} in order to evaluate

$$\int_{-1}^{1} f(x)dx \approx \int_{-1}^{1} P_{2N-1}(x)dx.$$

We can use Eq. (5.4.1) to rewrite the above integral as

$$\int_{-1}^{1} P_{2N-1}(x)dx = \int_{-1}^{1} (L_N(x)P_{N-1}(x) + Q_{N-1}(x))dx = \int_{-1}^{1} Q_{N-1}(x)dx,$$

due to the orthogonality properties of the Legendre polynomials. We see that it suffices to evaluate the integral over $\int_{-1}^{1} Q_{N-1}(x)dx$ in order to evaluate $\int_{-1}^{1} P_{2N-1}(x)dx$. In addition, at the points x_k where L_N is zero, we have

$$P_{2N-1}(x_k) = Q_{N-1}(x_k) \qquad k = 0, 1, \ldots, N-1,$$

and we see that through these N points we can fully define $Q_{N-1}(x)$ and thereby the integral. Note that we have chosen to let the numbering of the points run from 0 to $N-1$. The reason for this choice is that we wish to have the same numbering as the order of a polynomial of degree $N-1$. This numbering will be useful below when we introduce the matrix elements which define the integration weights w_i.

We develope then $Q_{N-1}(x)$ in terms of Legendre polynomials, as done in Eq. (5.4.1), $\mathrm{Q}_{N-1}(x) = \sum_{i=0}^{N-1} \alpha_i L_i(x). Usingtheorthogonalitypropertyof theLegendrepolynomialswehave \int_{-1}^{1} Q_{N-1}(x)dx$ $\sum_{i=0}^{N-1} \alpha_i \int_{-1}^{1} L_0(x)L_i(x)dx = 2\alpha_0,$where we have just inserted $L_0(x) = 1$! Instead of an integration problem we need now to define the coefficient α_0. Since we know the values of Q_{N-1} at the zeros of L_N, we may rewrite Eq. (5.4.2) as $\mathrm{Q}_{N-1}(x_k) = \sum_{i=0}^{N-1} \alpha_i L_i(x_k) = \sum_{i=0}^{N-1} \alpha_i L_{ik} \qquad k = 0, 1, \ldots, N-1. SincetheLegendrepolynomialsarelinearlyindependentof$ are linear combinations of the others. This means that the matrix L_{ik} has an inverse with the properties

$$\hat{\mathbf{L}}^{-1}\hat{\mathbf{L}} = \hat{\mathbf{I}}.$$

Multiplying both sides of Eq. (5.4.2) with $\sum_{j=0}^{N-1} L_{ji}^{-1}$ results in $\sum_{i=0}^{N-1} (L^{-1})_{ki} Q_{N-1}(x_i) =$

$\alpha_k. We can derive this result in an alternative way by defining the vectors \hat{\mathbf{x}}_k = \begin{pmatrix} x_0 \\ x_1 \\ . \\ . \\ x_{N-1} \end{pmatrix} \qquad \hat{\alpha} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ . \\ . \\ \alpha_{N-1} \end{pmatrix}$,and

the matrix

$$\hat{\mathbf{L}} = \begin{pmatrix} L_0(x_0) & L_1(x_0) & \ldots & L_{N-1}(x_0) \\ L_0(x_1) & L_1(x_1) & \ldots & L_{N-1}(x_1) \\ \ldots & \ldots & \ldots & \ldots \\ L_0(x_{N-1}) & L_1(x_{N-1}) & \ldots & L_{N-1}(x_{N-1}) \end{pmatrix}.$$

We have then

$$Q_{N-1}(\hat{x}_k) = \hat{L}\hat{\alpha},$$

yielding (if $\hat{L}$ has an inverse)

$$\hat{L}^{-1}Q_{N-1}(\hat{x}_k) = \hat{\alpha},$$

which is Eq. (5.4.2).

Using the above results and the fact that

$$\int_{-1}^{1} P_{2N-1}(x)dx = \int_{-1}^{1} Q_{N-1}(x)dx,$$

we get

$$\int_{-1}^{1} P_{2N-1}(x)dx = \int_{-1}^{1} Q_{N-1}(x)dx = 2\alpha_0 = 2\sum_{i=0}^{N-1}(L^{-1})_{0i}P_{2N-1}(x_i).$$

If we identify the weights with $2(L^{-1})_{0i}$, where the points x_i are the zeros of L_N, we have an integration formula of the type

$$\int_{-1}^{1} P_{2N-1}(x)dx = \sum_{i=0}^{N-1}\omega_i P_{2N-1}(x_i)$$

and if our function $f(x)$ can be approximated by a polynomial P of degree $2N-1$, we have finally that

$$\int_{-1}^{1} f(x)dx \approx \int_{-1}^{1} P_{2N-1}(x)dx = \sum_{i=0}^{N-1}\omega_i P_{2N-1}(x_i).$$

In summary, the mesh points x_i are defined by the zeros of an orthogonal polynomial of degree N, that is L_N, while the weights are given by $2(L^{-1})_{0i}$.

5.4.3 Application to the case $N = 2$

Let us apply the above formal results to the case $N = 2$. This means that we can approximate a function $f(x)$ with a polynomial $P_3(x)$ of order $2N-1 = 3$.

The mesh points are the zeros of $L_2(x) = 1/2(3x^2-1)$. These points are $x_0 = -1/\sqrt{3}$ and $x_1 = 1/\sqrt{3}$.

Specializing Eq. (5.4.2)

$$Q_{N-1}(x_k) = \sum_{i=0}^{N-1}\alpha_i L_i(x_k) \qquad k = 0,1,\ldots,N-1.$$

to $N = 2$ yields

$$Q_1(x_0) = \alpha_0 - \alpha_1\frac{1}{\sqrt{3}},$$

and

$$Q_1(x_1) = \alpha_0 + \alpha_1 \frac{1}{\sqrt{3}},$$

since $L_0(x = \pm 1/\sqrt{3}) = 1$ and $L_1(x = \pm 1/\sqrt{3}) = \pm 1/\sqrt{3}$.

The matrix L_{ik} defined in Eq. (5.4.2) is then

$$\hat{\mathbf{L}} = \begin{pmatrix} 1 & -\frac{1}{\sqrt{3}} \\ 1 & \frac{1}{\sqrt{3}} \end{pmatrix},$$

with an inverse given by

$$\hat{\mathbf{L}}^{-1} = \frac{\sqrt{3}}{2} \begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ -1 & 1 \end{pmatrix}.$$

The weights are given by the matrix elements $2(L_{0k})^{-1}$. We have thence $\omega_0 = 1$ and $\omega_1 = 1$.

Obviously, there is no problem in changing the numbering of the matrix elements $i,k = 0,1,2,\dots,N-1$ to $i,k = 1,2,\dots,N$. We have chosen to start from zero, since we deal with polynomials of degree $N-1$.

Summarizing, for Legendre polynomials with $N = 2$ we have weights

$$\omega : \{1, 1\},$$

and mesh points

$$x : \left\{ -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}} \right\}.$$

If we wish to integrate

$$\int_{-1}^{1} f(x)dx,$$

with $f(x) = x^2$, we approximate

$$I = \int_{-1}^{1} x^2 dx \approx \sum_{i=0}^{N-1} \omega_i x_i^2.$$

The exact answer is $2/3$. Using $N = 2$ with the above two weights and mesh points we get

$$I = \int_{-1}^{1} x^2 dx = \sum_{i=0}^{1} \omega_i x_i^2 = \frac{1}{3} + \frac{1}{3} = \frac{2}{3},$$

the exact answer!

If we were to emply the trapezoidal rule we would get

$$I = \int_{-1}^{1} x^2 dx = \frac{b-a}{2} \left((a)^2 + (b)^2 \right)/2 = \frac{1-(-1)}{2} \left((-1)^2 + (1)^2 \right)/2 = 1!$$

With just two points we can calculate exactly the integral for a second-order polynomial since our methods approximates the exact function with higher order polynomial. How many points do you need with the trapezoidal rule in order to achieve a similar accuracy?

5.4.4 General integration intervals for Gauss-Legendre

Note that the Gauss-Legendre method is not limited to an interval [-1,1], since we can always through a change of variable

$$t = \frac{b-a}{2}x + \frac{b+a}{2},$$

rewrite the integral for an interval [a,b]

$$\int_a^b f(t)dt = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{(b-a)x}{2} + \frac{b+a}{2}\right)dx.$$

If we have an integral on the form

$$\int_0^\infty f(t)dt,$$

we can choose new mesh points and weights by using the mapping

$$\tilde{x}_i = tan\left\{\frac{\pi}{4}(1+x_i)\right\},$$

and

$$\tilde{\omega}_i = \frac{\pi}{4}\frac{\omega_i}{cos^2\left(\frac{\pi}{4}(1+x_i)\right)},$$

where x_i and ω_i are the original mesh points and weights in the interval $[-1,1]$, while $\tilde{x}_i$ and $\tilde{\omega}_i$ are the new mesh points and weights for the interval $[0,\infty)$.

To see that this is correct by inserting the the value of $x_i = -1$ (the lower end of the interval $[-1,1]$) into the expression for $\tilde{x}_i$. That gives $\tilde{x}_i = 0$, the lower end of the interval $[0,\infty)$. For $x_i = 1$, we obtain $\tilde{x}_i = \infty$. To check that the new weights are correct, recall that the weights should correspond to the derivative of the mesh points. Try to convince yourself that the above expression fulfills this condition.

5.4.5 Other orthogonal polynomials

Laguerre polynomials

If we are able to rewrite our integral of Eq. (5.4) with a weight function $W(x) = x^\alpha e^{-x}$ with integration limits $[0,\infty)$, we could then use the Laguerre polynomials.

The polynomials form then the basis for the Gauss-Laguerre method which can be applied to integrals of the form

$$I = \int_0^\infty f(x)dx = \int_0^\infty x^\alpha e^{-x} g(x)dx.$$

These polynomials arise from the solution of the differential equation

$$\left(\frac{d^2}{dx^2} - \frac{d}{dx} + \frac{\lambda}{x} - \frac{l(l+1)}{x^2}\right)\mathscr{L}(x) = 0,$$

where l is an integer $l \geq 0$ and λ a constant. This equation arises for example from the solution of the radial Schrödinger equation with a centrally symmetric potential such as the Coulomb potential. The first few polynomials are

$$\mathscr{L}_0(x) = 1,$$

$$\mathscr{L}_1(x) = 1 - x,$$

$$\mathscr{L}_2(x) = 2 - 4x + x^2,$$

$$\mathscr{L}_3(x) = 6 - 18x + 9x^2 - x^3,$$

and

$$\mathscr{L}_4(x) = x^4 - 16x^3 + 72x^2 - 96x + 24.$$

They fulfil the orthogonality relation

$$\int_0^\infty e^{-x}\mathscr{L}_n(x)^2 dx = 1,$$

and the recursion relation

$$(n+1)\mathscr{L}_{n+1}(x) = (2n+1-x)\mathscr{L}_n(x) - n\mathscr{L}_{n-1}(x).$$

Hermite polynomials

In a similar way, for an integral which goes like

$$I = \int_{-\infty}^\infty f(x)dx = \int_{-\infty}^\infty e^{-x^2} g(x)dx.$$

we could use the Hermite polynomials in order to extract weights and mesh points. The Hermite polynomials are the solutions of the following differential equation

$$\frac{d^2H(x)}{dx^2} - 2x\frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0.$$

A typical example is again the solution of Schrödinger’s equation, but this time with a harmonic oscillator potential. The first few polynomials are

$$H_0(x) = 1,$$

$$H_1(x) = 2x,$$

$$H_2(x) = 4x^2 - 2,$$

$$H_3(x) = 8x^3 - 12,$$

and

$$H_4(x) = 16x^4 - 48x^2 + 12.$$

They fulfil the orthogonality relation

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x)^2 dx = 2^n n! \sqrt{\pi},$$

and the recursion relation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x).$$

5.4.6 Applications to selected integrals

Before we proceed with some selected applications, it is important to keep in mind that since the mesh points are not evenly distributed, a careful analysis of the behavior of the integrand as function of x and the location of mesh points is mandatory. To give you an example, in the Table below we show the mesh points and weights for the integration interval [0,100] for $N = 10$ points obtained by the Gauss-Legendre method. Clearly, if your function oscillates strongly in any subinterval, this approach needs to be refined, either by choosing more points or by choosing other integration methods. Note also that for integration intervals like for example $x \in [0, \infty]$, the Gauss-Legendre method places more points at the beginning of the integration interval. If your integrand varies slowly for large values of x, then this method may be appropriate.

Let us here compare three methods for integrating, namely the trapezoidal rule, Simpson’s method and the Gauss-Legendre approach. We choose two functions to integrate:

$$\int_1^{100} \frac{\exp(-x)}{x} dx,$$

and

$$\int_0^3 \frac{1}{2+x^2} dx.$$

A program example which uses the trapezoidal rule, Simpson’s rule and the Gauss-Legendre method is included here. For the corresponding Fortran program, replace program1.cpp with program1.f90. The Python program is listed as program1.py.

Table 5.1: Mesh points and weights for the integration interval [0,100] with $N = 10$ using the Gauss-Legendre method.

i	x_i	ω_i
1	1.305	3.334
2	6.747	7.473
3	16.030	10.954
4	28.330	13.463
5	42.556	14.776
6	57.444	14.776
7	71.670	13.463
8	83.970	10.954
9	93.253	7.473
10	98.695	3.334

http://folk.uio.no/mhjensen/compphys/programs/chapter05/cpp/program1.cpp

```
#include <iostream>
#include "lib.h"
using namespace std;
//   Here we define various functions called by the main program
//   this function defines the function to integrate
double int_function(double x);
//  Main function begins here
int main()
{
    int n;
    double a, b;
    cout << "Read in the number of integration points" << endl;
    cin >> n;
    cout << "Read in integration limits" << endl;
    cin >> a >> b;
//  reserve space in memory for vectors containing the mesh points
//  weights and function values for the use of the gauss-legendre
//  method
    double *x = new double [n];
    double *w = new double [n];
//  set up the mesh points and weights
    gauss_legendre(a, b,x,w, n);
//  evaluate the integral with the Gauss-Legendre method
//  Note that we initialize the sum
    double int_gauss = 0.;
    for ( int i = 0; i < n; i++){
      int_gauss+=w[i]*int_function(x[i]);
    }
//   final output
```

```
    cout << "Trapez-rule = " << trapezoidal_rule(a, b,n, int_function)
       << endl;
    cout << "Simpson's rule = " << simpson(a, b,n, int_function)
       << endl;
    cout << "Gaussian quad = " << int_gauss << endl;
    delete [] x;
    delete [] w;
    return 0;
} // end of main program
// this function defines the function to integrate
double int_function(double x)
{
 double value = 4./(1.+x*x);
 return value;
} // end of function to evaluate
```

To be noted in this program is that we can transfer the name of a given function to integrate. In Table 5.2 we show the results for the first integral using various mesh points, while Table 5.3 displays the corresponding results obtained with the second integral. We note here that, since the area over where we integrate is rather

Table 5.2: Results for $\int_1^{100} \exp{(-x)}/x dx$ using three different methods as functions of the number of mesh points N.

N	Trapez	Simpson	Gauss-Legendre
10	1.821020	1.214025	0.1460448
20	0.912678	0.609897	0.2178091
40	0.478456	0.333714	0.2193834
100	0.273724	0.231290	0.2193839
1000	0.219984	0.219387	0.2193839

large and the integrand goes slowly to zero for large values of x, both the trapezoidal rule and Simpson's method need quite many points in order to approach the Gauss-Legendre method. This integrand demonstrates clearly the strength of the Gauss-Legendre method (and other GQ methods as well), viz., few points are needed in order to achieve a very high precision.

The second table however shows that for smaller integration intervals, both the trapezoidal rule and Simpson's method compare well with the results obtained with the Gauss-Legendre approach.

Table 5.3: Results for $\int_0^3 1/(2+x^2)dx$ using three different methods as functions of the number of mesh points N.

N	Trapez	Simpson	Gauss-Legendre
10	0.798861	0.799231	0.799233
20	0.799140	0.799233	0.799233
40	0.799209	0.799233	0.799233
100	0.799229	0.799233	0.799233
1000	0.799233	0.799233	0.799233

5.5 Treatment of Singular Integrals

So-called principal value (PV) integrals are often employed in physics, from Green's functions for scattering to dispersion relations. Dispersion relations are often related to measurable quantities and provide important consistency checks in atomic, nuclear and particle physics. A PV integral is defined as

$$I(x) = \mathscr{P}\int_a^b dt\frac{f(t)}{t-x} = \lim_{\varepsilon\to 0^+}\left[\int_a^{x-\varepsilon} dt\frac{f(t)}{t-x} + \int_{x+\varepsilon}^b dt\frac{f(t)}{t-x}\right],$$

and arises in applications of Cauchy's residue theorem when the pole x lies on the real axis within the interval of integration $[a,b]$. Here $\mathscr{P}$ stands for the principal value. *An important assumption is that the function $f(t)$ is continuous on the interval of integration.*

In case $f(t)$ is a closed form expression or it has an analytic continuation in the complex plane, it may be possible to obtain an expression on closed form for the above integral.

However, the situation which we are often confronted with is that $f(t)$ is only known at some points t_i with corresponding values $f(t_i)$. In order to obtain $I(x)$ we need to resort to a numerical evaluation.

To evaluate such an integral, let us first rewrite it as

$$\mathscr{P}\int_a^b dt\frac{f(t)}{t-x} = \int_a^{x-\Delta} dt\frac{f(t)}{t-x} + \int_{x+\Delta}^b dt\frac{f(t)}{t-x} + \mathscr{P}\int_{x-\Delta}^{x+\Delta} dt\frac{f(t)}{t-x},$$

where we have isolated the principal value part in the last integral.

Defining a new variable $u = t-x$, we can rewrite the principal value integral as I$_\Delta(x) = \mathscr{P}\int_{-\Delta}^{+\Delta} du\frac{f(u+x)}{u}.OnepossibilityistoTaylorexpand$f(u+x)$around$u=0$,andcomputederivativestoacert$
$\sum_{n=0}^{N_{max}} f^{(2n+1)}(x)\frac{\Delta^{2n+1}}{(2n+1)(2n+1)!}$.

To evaluate higher-order derivatives may be both time consuming and delicate from a numerical point of view, since there is always the risk of loosing precision when calculating derivatives numerically. Unless we have an analytic expression for

$f(u+x)$ and can evaluate the derivatives in a closed form, the above approach is not the preferred one.

Rather, we show here how to use the Gauss-Legendre method to compute Eq. (5.5). Let us first introduce a new variable $s = u/\Delta$ and rewrite Eq. (5.5) as $\mathrm{I}_\Delta(x) = \mathscr{P}\int_{-1}^{+1} ds \frac{f(\Delta s+x)}{s}$.

The integration limits are now from -1 to 1, as for the Legendre polynomials. The principal value in Eq. (5.5) is however rather tricky to evaluate numerically, mainly since computers have limited precision. We will here use a subtraction trick often used when dealing with singular integrals in numerical calculations. We introduce first the calculus relation

$$\int_{-1}^{+1} \frac{ds}{s} = 0.$$

It means that the curve $1/(s)$ has equal and opposite areas on both sides of the singular point $s = 0$.

If we then note that $f(x)$ is just a constant, we have also

$$f(x)\int_{-1}^{+1} \frac{ds}{s} = \int_{-1}^{+1} f(x)\frac{ds}{s} = 0.$$

Subtracting this equation from Eq. (5.5) yields $\mathrm{I}_\Delta(x) = \mathscr{P}\int_{-1}^{+1} ds \frac{f(\Delta s+x)}{s} = \int_{-1}^{+1} ds \frac{f(\Delta s+x)-f(x)}{s}, andth$
$x) - f(x)) = 0$ and for the particular case $s = 0$ the integrand is now finite.

Eq. (5.5) is now rewritten using the Gauss-Legendre method resulting in $\int_{-1}^{+1} ds \frac{f(\Delta s+x)-f(x)}{s} = \sum_{i=1}^{N} \omega_i \frac{f(\Delta s_i+x)-f(x)}{s_i}, where s_i$ are the mesh points (N in total) and ω_i are the weights.

In the selection of mesh points for a PV integral, it is important to use an even number of points, since an odd number of mesh points always picks $s_i = 0$ as one of the mesh points. The sum in Eq. (5.5) will then diverge.

Let us apply this method to the integral I(x)=P$\int_{-1}^{+1} dt \frac{e^t}{t}. The integrand diverges at$x=t=0$.We rewrite it$
$\int_{-1}^{+1} \frac{e^t-1}{t}, since$e$^x = e^0 = 1$. With Eq. (5.5) we have then $\int_{-1}^{+1} \frac{e^t-1}{t} \approx \sum_{i=1}^{N} \omega_i \frac{e^{t_i}-1}{t_i}$.

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

$$\int_{-\infty}^{\infty} \frac{dk}{k-k_0} = \int_{-\infty}^{0} \frac{dk}{k-k_0} + \int_{0}^{\infty} \frac{dk}{k-k_0} = 0.$$

A change of variable $u = -k$ in the integral with limits from $-\infty$ to 0 gives

$$\int_{-\infty}^{\infty} \frac{dk}{k-k_0} = \int_{\infty}^{0} \frac{-du}{-u-k_0} + \int_{0}^{\infty} \frac{dk}{k-k_0} = \int_{0}^{\infty} \frac{dk}{-k-k_0} + \int_{0}^{\infty} \frac{dk}{k-k_0} = 0.$$

It means that the curve $1/(k-k_0)$ has equal and opposite areas on both sides of the singular point k_0. If we break the integral into one over positive k and one over negative k, a change of variable $k \to -k$ allows us to rewrite the last equation as

$$\int_{0}^{\infty} \frac{dk}{k^2-k_0^2} = 0.$$

We can use this to express a principal values integral as

$$\mathscr{P} \int_{0}^{\infty} \frac{f(k)dk}{k^2-k_0^2} = \int_{0}^{\infty} \frac{(f(k)-f(k_0))dk}{k^2-k_0^2}, \tag{5.4}$$

where the right-hand side is no longer singular at $k = k_0$, it is proportional to the derivative df/dk, and can be evaluated numerically as any other integral.

Such a trick is often used when evaluating integral equations, as discussed in the next section.

5.6 Parallel Computing

We end this chapter by discussing modern supercomputing concepts like parallel computing. In particular, we will introduce you to the usage of the Message Passing Interface (MPI) library. MPI is a library, not a programming language. It specifies the names, calling sequences and results of functions or subroutines to be called from C++ or Fortran programs, and the classes and methods that make up the MPI C++ library. The programs that users write in Fortran or C++ are compiled with ordinary compilers and linked with the MPI library. MPI programs should be able to run on all possible machines and run all MPI implementetations without change. An excellent reference is the text by Karniadakis and Kirby II [2].

5.6.1 Brief survey of supercomputing concepts and terminologies

Since many discoveries in science are nowadays obtained via large-scale simulations, there is an ever-lasting wish and need to do larger simulations using shorter computer time. The development of the capacity for single-processor computers (even with increased processor speed and memory) can hardly keep up with the pace of scientific computing. The solution to the needs of the scientific computing and high-performance computing (HPC) communities has therefore been parallel computing.

The basic ideas of parallel computing is that multiple processors are involved to solve a global problem. The essence is to divide the entire computation evenly among collaborative processors.

Today's supercomputers are parallel machines and can achieve peak performances almost up to 10^{15} floating point operations per second, so-called peta-scale computers, see for example the list over the world's top 500 supercomputers at www.top500.org. This list gets updated twice per year and sets up the ranking according to a given supercomputer's performance on a benchmark code from the LINPACK library. The benchmark solves a set of linear equations using the best software for a given platform.

To understand the basic philosophy, it is useful to have a rough picture of how to classify different hardware models. We distinguish betwen three major groups, (i) conventional single-processor computers, normally called SISD (single-instruction-single-data) machines, (ii) so-called SIMD machines (single-instruction-multiple-data), which incorporate the idea of parallel processing using a large number of processing units to execute the same instruction on different data and finally (iii) modern parallel computers, so-called MIMD (multiple-instruction- multiple-data) machines that can execute different instruction streams in parallel on different data. On a MIMD machine the different parallel processing units perform operations independently of each others, only subject to synchronization via a given message passing interface at specified time intervals. MIMD machines are the dominating ones among present supercomputers, and we distinguish between two types of MIMD computers, namely shared memory machines and distributed memory machines. In shared memory systems the central processing units (CPU) share the same address space. Any CPU can access any data in the global memory. In distributed memory systems each CPU has its own memory. The CPUs are connected by some network and may exchange messages. A recent trend are so-called ccNUMA (cache-coherent-non-uniform-memory-access) systems which are clusters of SMP (symmetric multi-processing) machines and have a virtual shared memory.

Distributed memory machines, in particular those based on PC clusters, are nowadays the most widely used and cost-effective, although farms of PC clusters require large infrastuctures and yield additional expenses for cooling. PC clusters with Linux as operating systems are easy to setup and offer several advantages, since they are built from standard commodity hardware with the open source software (Linux) infrastructure. The designer can improve performance proportionally with added machines. The commodity hardware can be any of a number of mass-market, stand-alone compute nodes as simple as two networked computers each running Linux and sharing a file system or as complex as thousands of nodes with a high-speed, low-latency network. In addition to the increased speed of present individual processors (and most machines come today with dual cores or four cores, so-called quad-cores) the position of such commodity supercomputers has been strenghtened by the fact that a library like MPI has made parallel computing portable and easy. Although there are several implementations, they share the same core commands. Message-passing is a mature programming paradigm and widely accepted. It often provides an efficient match to the hardware.

5.6.2 Parallelism

When we discuss parallelism, it is common to subdivide different algorithms in three major groups.

- **Task parallelism**:the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulations and numerical integration are examples of possible applications. Since there is more or less no communication between different processors, task parallelism results in almost a perfect mathematical parallelism and is commonly dubbed embarassingly parallel (EP). The examples in this chapter fall under that category. The use of the MPI library is then limited to some few function calls and the programming is normally very simple.

- **Data parallelism**: use of multiple threads (e.g., one thread per processor) to dissect loops over arrays etc. This paradigm requires a single memory address space. Communication and synchronization between the processors are often hidden, and it is thus easy to program. However, the user surrenders much control to a specialized compiler. An example of data parallelism is compiler-based parallelization.

- **Message-passing**: all involved processors have an independent memory address space. The user is responsible for partitioning the data/work of a global problem and distributing the subproblems to the processors. Collaboration between processors is achieved by explicit message passing, which is used for data transfer plus synchronization.

  This paradigm is the most general one where the user has full control. Better parallel efficiency is usually achieved by explicit message passing. However, message-passing programming is more difficult. We will meet examples of this in connection with the solution eigenvalue problems in chapter 7 and of partial differential equations in chapter **??**.

Before we proceed, let us look at two simple examples. We will also use these simple examples to define the speedup factor of a parallel computation. The first case is that of the additions of two vectors of dimension n,

$$\mathbf{z} = \alpha\mathbf{x} + \beta\mathbf{y},$$

where α and β are two real or complex numbers and $\mathbf{z}, \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ or $\in \mathbb{C}^n$. For every element we have thus

$$z_i = \alpha x_i + \beta y_i.$$

For every element z_i we have three floating point operations, two multiplications and one addition. If we assume that these operations take the same time Δt, then the total time spent by one processor is

$$T_1 = 3n\Delta t.$$

Suppose now that we have access to a parallel supercomputer with P processors. Assume also that $P \le n$. We split then these addition and multiplication operations on every processor so that every processor performs $3n/P$ operations in total, resulting in a time $T_P = 3n\Delta t/P$ for every single processor. We also assume that the time needed to gather together these subsums is neglible

If we have perfect parallelism, our speedup should be P, the number of processors available. We see that this is the case by computing the relation between the time used in case of only one processor and the time used if we can access P processors. The speedup S_P is defined as

$$S_P = \frac{T_1}{T_P} = \frac{3n\Delta t}{3n\Delta t/P} = P,$$

a perfect speedup. As mentioned above, we call calculations that yield a perfect speedup for embarassingly parallel. The efficiency is defined as

$$\eta(P) = \frac{S(P)}{P}.$$

Our next example is that of the inner product of two vectors defined in Eq. (6.5),

$$c = \sum_{j=1}^{n} x_j y_j.$$

We assume again that $P \le n$ and define $I = n/P$. Each processor is assigned with its own subset of local multiplications $c_P = \sum_p x_p y_p$, where p runs over all possible terms for processor P. As an example, assume that we have four processors. Then we have

$$c_1 = \sum_{j=1}^{n/4} x_j y_j, \qquad c_2 = \sum_{j=n/4+1}^{n/2} x_j y_j,$$

$$c_3 = \sum_{j=n/2+1}^{3n/4} x_j y_j, \qquad c_4 = \sum_{j=3n/4+1}^{n} x_j y_j.$$

We assume again that the time for every operation is Δt. If we have only one processor, the total time is $T_1 = (2n-1)\Delta t$. For four processors, we must now add the time needed to add $c_1 + c_2 + c_3 + c_4$, which is $3\Delta t$ (three additions) and the time needed to communicate the local result c_P to all other processors. This takes roughly $(P-1)\Delta t_c$, where Δt_c need not equal Δt.

The speedup for four processors becomes now

$$S_4 = \frac{T_1}{T_4} = \frac{(2n-1)\Delta t}{(n/2-1)\Delta t + 3\Delta t + 3\Delta t_c} = \frac{4n-2}{10+n},$$

if $\Delta t = \Delta t_c$. For $n = 100$, the speedup is $S_4 = 3.62 < 4$. For P processors the inner products yields a speedup

$$S_P = \frac{(2n-1)}{(2I + P - 2)) + (P-1)\gamma},$$

with $\gamma = \Delta t_c / \Delta t$. Even with $\gamma = 0$, we see that the speedup is less than P.

The communication time Δt_c can reduce significantly the speedup. However, even if it is small, there are other factors as well which may reduce the efficiency η_p. For example, we may have an uneven load balance, meaning that not all the processors can perform useful work at all time, or that the number of processors doesn't match properly the size of the problem, or memory problems, or that a so-called startup time penalty known as latency may slow down the transfer of data. Crucial here is the rate at which messages are transferred

5.6.3 MPI with simple examples

When we want to parallelize a sequential algorithm, there are at least two aspects we need to consider, namely

- Identify the part(s) of a sequential algorithm that can be executed in parallel. This can be difficult.
- Distribute the global work and data among P processors. Stated differently, here you need to understand how you can get computers to run in parallel. From a practical point of view it means to implement parallel programming tools.

In this chapter we focus mainly on the last point. MPI is then a tool for writing programs to run in parallel, without needing to know much (in most cases nothing) about a given machine's architecture. MPI programs work on both shared memory and distributed memory machines. Furthermore, MPI is a very rich and complicated library. But it is not necessary to use all the features. The basic and most used functions have been optimized for most machine architectures

Before we proceed, we need to clarify some concepts, in particular the usage of the words process and processor. We refer to process as a logical unit which executes its own code, in an MIMD style. The processor is a physical device on which one or several processes are executed. The MPI standard uses the concept process consistently throughout its documentation. However, since we only consider situations where one processor is responsible for one process, we therefore use the two terms interchangeably in the discussion below, hopefully without creating ambiguities.

The six most important MPI functions are

- MPI_ Init - initiate an MPI computation
- MPI_Finalize - terminate the MPI computation and clean up
- MPI_Comm_size - how many processes participate in a given MPI computation.
- MPI_Comm_rank - which rank does a given process have. The rank is a number between 0 and size-1, the latter representing the total number of processes.
- MPI_Send - send a message to a particular process within an MPI computation
- MPI_Recv - receive a message from a particular process within an MPI computation.

The first MPI C++ program is a rewriting of our ’hello world’ program (without the computation of the sine function) from chapter 2. We let every process write "Hello world" on the standard output.

http://folk.uio.no/mhjensen/compphys/programs/chapter05/program2.cpp

```
//   First C++ example of MPI Hello world
using namespace std;
#include <mpi.h>
#include <iostream>

int main (int nargs, char* args[])
{
    int numprocs, my_rank;
//  MPI initializations
    MPI_Init (&nargs, &args);
    MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    cout << "Hello world, I have rank " << my_rank << " out of " << numprocs << endl;
// End MPI
```

```
    MPI_Finalize ();
   return 0;
}
```

The corresponding Fortran program reads

```
PROGRAM hello
  INCLUDE "mpif.h"
  INTEGER:: numprocs, my_rank, ierr

  CALL MPI_INIT(ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
  WRITE(*,*)"Hello world, I've rank ",my_rank," out of ",numprocs
  CALL MPI_FINALIZE(ierr)

END PROGRAM hello
```

MPI is a message-passing library where all the routines have a corresponding C++-bindings[3] `MPI_Command_name` or Fortran-bindings (function names are by convention in uppercase, but can also be in lower case) `MPI_COMMAND_NAME`

To use the MPI library you must include header files which contain definitions and declarations that are needed by the MPI library routines. The following line must appear at the top of any source code file that will make an MPI call. For Fortran you must put in the beginning of your program the declaration

```
INCLUDE 'mpif.h'
```

while for C++ you need to include the statement

```
#include "mpi.h"
```

These header files contain the declarations of functions, variabels etc. needed by the MPI library.

The first MPI call must be `MPI_INIT`, which initializes the message passing routines, as defined in for example

```
INTEGER :: ierr
CALL MPI_INIT(ierr)
```

for the Fortran example. The variable `ierr` is an integer which holds an error code when the call returns. The value of `ierr` is however of little use since, by default, MPI aborts the program when it encounters an error. However, `ierr` must be included when MPI starts. For the C++ code we have the call to the function

[3]The C++ bindings used in practice are the same as the C bindings, although reading older texts like [1, 2, 4] one finds extensive discussions on the difference between C and C++ bindings. Throughout this text we will use the C bindings.

```
MPI_Init(int *argc, char *argv)
```

where `argc` and `argv` are arguments passed to main. MPI does not use these arguments in any way, however, and in MPI-2 implementations, NULL may be passed instead. When you have finished you must call the function `MPI_Finalize`. In Fortran you use the statement

```
CALL MPI_FINALIZE(ierr)
```

while for C++ we use the function `MPI_Finalize()`.

In addition to these calls, we have also included calls to so-called inquiry functions. There are two MPI calls that are usually made soon after initialization. They are for C++,

```
MPI_COMM_SIZE((MPI_COMM_WORLD, &numprocs)
```

and

```
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
```

for Fortran. The function `MPI_COMM_SIZE` returns the number of tasks in a specified MPI communicator (comm when we refer to it in generic function calls below).

In MPI you can divide your total number of tasks into groups, called communicators. What does that mean? All MPI communication is associated with what one calls a communicator that describes a group of MPI processes with a name (context). The communicator designates a collection of processes which can communicate with each other. Every process is then identified by its rank. The rank is only meaningful within a particular communicator. A communicator is thus used as a mechanism to identify subsets of processes. MPI has the flexibility to allow you to define different types of communicators, see for example [4]. However, here we have used the communicator `MPI_COMM_WORLD` that contains all the MPI processes that are initiated when we run the program.

The variable `numprocs` refers to the number of processes we have at our disposal. The function `MPI_COMM_RANK` returns the rank (the name or identifier) of the tasks running the code. Each task (or processor) in a communicator is assigned a number `my_rank` from 0 to $\mathrm{numprocs}-1$.

We are now ready to perform our first MPI calculations.

Running codes with MPI

To compile and load the above C++ code (after having understood how to use a local cluster), we can use the command

```
mpicxx -O2 -o program2.x  program2.cpp
```

and try to run with ten nodes using the command

```
mpiexec -np 10 ./program2.x
```

If we wish to use the Fortran version we need to replace the C++ compiler statement `mpicc` with `mpif90` or equivalent compilers. The name of the compiler is obviously system dependent. The command `mpirun` may be used instead of `mpiexec`. Here you need to check your own system.

When we run MPI all processes use the same binary executable version of the code and all processes are running exactly the same code. The question is then how can we tell the difference between our parallel code running on a given number of processes and a serial code? There are two major distinctions you should keep in mind: (i) MPI lets each process have a particular rank to determine which instructions are run on a particular process and (ii) the processes communicate with each other in order to finalize a task. Even if all processes receive the same set of instructions, they will normally not execute the same instructions.We will discuss this point in connection with our integration example below.

The above example produces the following output

```
Hello world, I’ve rank 0 out of 10 procs.
Hello world, I’ve rank 1 out of 10 procs.
Hello world, I’ve rank 4 out of 10 procs.
Hello world, I’ve rank 3 out of 10 procs.
Hello world, I’ve rank 9 out of 10 procs.
Hello world, I’ve rank 8 out of 10 procs.
Hello world, I’ve rank 2 out of 10 procs.
Hello world, I’ve rank 5 out of 10 procs.
Hello world, I’ve rank 7 out of 10 procs.
Hello world, I’ve rank 6 out of 10 procs.
```

The output to screen is not ordered since all processes are trying to write to screen simultaneously. It is then the operating system which opts for an ordering. If we wish to have an organized output, starting from the first process, we may rewrite our program as follows

http://folk.uio.no/mhjensen/compphys/programs/chapter05/program3.cpp

```
//   Second C++ example of MPI Hello world
using namespace std;
#include <mpi.h>
#include <iostream>

int main (int nargs, char* args[])
{
   int numprocs, my_rank, i;
//  MPI initializations
    MPI_Init (&nargs, &args);
    MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    for (i = 0; i < numprocs; i++) {
      MPI_Barrier (MPI_COMM_WORLD);
      if (i == my_rank) {
        cout << "Hello world, I have rank " << my_rank << " out of " << numprocs <<
            endl;
        fflush (stdout);
      }
    }
// End MPI
     MPI_Finalize ();
   return 0;
}
```

Here we have used the `MPI_Barrier` function to ensure that every process has completed its set of instructions in a particular order. A barrier is a special collective operation that does not allow the processes to continue until all processes in the communicator (here `MPI_COMM_WORLD`) have called `MPI_Barrier`. The output is now

```
Hello world, I've rank 0 out of 10 procs.
Hello world, I've rank 1 out of 10 procs.
Hello world, I've rank 2 out of 10 procs.
Hello world, I've rank 3 out of 10 procs.
Hello world, I've rank 4 out of 10 procs.
Hello world, I've rank 5 out of 10 procs.
Hello world, I've rank 6 out of 10 procs.
Hello world, I've rank 7 out of 10 procs.
Hello world, I've rank 8 out of 10 procs.
Hello world, I've rank 9 out of 10 procs.
```

The barriers make sure that all processes have reached the same point in the code. Many of the collective operations like `MPI_ALLREDUCE` to be discussed later, have the same property; viz. no process can exit the operation until all processes have started. However, this is slightly more time-consuming since the processes synchronize between themselves as many times as there are processes. In the next Hello world example we use the send and receive functions in order to a have a synchronized action.

http://folk.uio.no/mhjensen/compphys/programs/chapter05/program4.cpp

```
//   Third C++ example of MPI Hello world
using namespace std;
#include <mpi.h>
```

```
#include <iostream>

int main (int nargs, char* args[])
{
    int numprocs, my_rank, flag;
//  MPI initializations
    MPI_Status status;
    MPI_Init (&nargs, &args);
    MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    //  Send and Receive example
    if (my_rank > 0)
      MPI_Recv (&flag, 1, MPI_INT, my_rank-1, 100, MPI_COMM_WORLD, &status);
      cout << "Hello world, I have rank " << my_rank << " out of " << numprocs <<
          endl;
    if (my_rank < numprocs-1)
        MPI_Send (&my_rank, 1, MPI_INT, my_rank+1, 100, MPI_COMM_WORLD);
// End MPI
     MPI_Finalize ();
   return 0;
}
```

The basic sending of messages is given by the function MPI_SEND, which in C++ is defined as

```
MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm
    comm)
```

while in Fortran we would call this function with the following parameters

```
CALL MPI_SEND(buf, count, MPI_TYPE, dest, tag, comm, ierr).
```

This single command allows the passing of any kind of variable, even a large array, to any group of tasks. The variable buf is the variable we wish to send while count is the number of variables we are passing. If we are passing only a single value, this should be 1. If we transfer an array, it is the overall size of the array. For example, if we want to send a 10 by 10 array, count would be $10 \times 10 = 100$ since we are actually passing 100 values.

We define the type of variable using MPI_TYPE in order to let MPI function know what to expect. The destination of the send is declared via the variable dest, which gives the ID number of the task we are sending the message to. The variable tag is a way for the receiver to verify that it is getting the message it expects. The message tag is an integer number that we can assign any value, normally a large number (larger than the expected number of processes). The communicator comm is the group ID of tasks that the message is going to. For complex programs, tasks may be divided into groups to speed up connections and transfers. In small programs, this will more than likely be in MPI_COMM_WORLD.

Furthermore, when an MPI routine is called, the Fortran or C++ data type which is passed must match the corresponding MPI integer constant. An integer is defined as `MPI_INT` in C++ and `MPI_INTEGER` in Fortran. A double precision real is `MPI_DOUBLE` in C++ and `MPI_DOUBLE_PRECISION` in Fortran and single precision real is `MPI_FLOAT` in C++ and `MPI_REAL` in Fortran. For further definitions of data types see chapter five of Ref. [4].

Once you have sent a message, you must receive it on another task. The function `MPI_RECV` is similar to the send call. In C++ we would define this as

```
MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm
    comm, MPI_Status *status )
```

while in Fortran we would use the call

```
CALL MPI_RECV(buf, count, MPI_TYPE, source, tag, comm, status, ierr)}.
```

The arguments that are different from those in `MPI_SEND` are `buf` which is the name of the variable where you will be storing the received data, `source` which replaces the destination in the send command. This is the return ID of the sender.

Finally, we have used `MPI_Status~status;` where one can check if the receive was completed. The source or tag of a received message may not be known if wildcard values are used in the receive function. In C++, MPI Status is a structure that contains further information. One can obtain this information using

```
MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count)}
```

The output of this code is the same as the previous example, but now process 0 sends a message to process 1, which forwards it further to process 2, and so forth.

Armed with this wisdom, performed all hello world greetings, we are now ready for serious work.

5.6.4 Numerical integration with MPI

To integrate numerically with MPI we need to define how to send and receive data types. This means also that we need to specify which data types to send to MPI functions.

The program listed here integrates

$$\pi = \int_0^1 dx \frac{4}{1+x^2}$$

by simply adding up areas of rectangles according to the algorithm discussed in Eq. (5.2), rewritten here

$$I = \int_a^b f(x)dx \approx h\sum_{i=1}^{N} f(x_{i-1/2}),$$

where $f(x) = 4/(1+x^2)$. This is a brute force way of obtaining an integral but suffices to demonstrate our first application of MPI to mathematical problems. What we do is to subdivide the integration range $x \in [0,1]$ into n rectangles. Increasing n should obviously increase the precision of the result, as discussed in the beginning of this chapter. The parallel part proceeds by letting every process collect a part of the sum of the rectangles. At the end of the computation all the sums from the processes are summed up to give the final global sum. The program below serves thus as a simple example on how to integrate in parallel. We will refine it in the next examples and we will also add a simple example on how to implement the trapezoidal rule.

http://folk.uio.no/mhjensen/compphys/programs/chapter05/program5.cpp

```
//   Reactangle rule and numerical integration using MPI send and Receive
using namespace std;
#include <mpi.h>
#include <iostream>

int main (int nargs, char* args[])
{
  int numprocs, my_rank, i, n = 1000;
  double local_sum, rectangle_sum, x, h;
  //  MPI initializations
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  //  Read from screen a possible new vaue of n
  if (my_rank == 0 && nargs > 1) {
    n = atoi(args[1]);
  }
  h = 1.0/n;
  // Broadcast n and h to all processes
  MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast (&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  // Every process sets up its contribution to the integral
  local_sum = 0.;
  for (i = my_rank; i < n; i += numprocs) {
    x = (i+0.5)*h;
    local_sum += 4.0/(1.0+x*x);
 }
  local_sum *= h;
  if (my_rank == 0) {
    MPI_Status status;
    rectangle_sum = local_sum;
    for (i=1; i < numprocs; i++) {
      MPI_Recv(&local_sum,1,MPI_DOUBLE,MPI_ANY_SOURCE,500,MPI_COMM_WORLD,&status);
      rectangle_sum += local_sum;
    }
    cout << "Result: " << rectangle_sum << endl;
36    } else
37      MPI_Send(&local_sum,1,MPI_DOUBLE,0,500,MPI_COMM_WORLD);
38    // End MPI
39    MPI_Finalize ();
40    return 0;
41  }
```

After the standard initializations with MPI such as

```
MPI_Init, MPI_Comm_size, MPI_Comm_rank,
```

`MPI_COMM_WORLD` contains now the number of processes defined by using for example

```
mpirun -np 10 ./prog.x
```

In line 14 we check if we have read in from screen the number of mesh points n. Note that in line 7 we fix $n = 1000$, however we have the possibility to run the code with a different number of mesh points as well. If `my_rank` equals zero, which correponds to the master node, then we read a new value of n if the number of arguments is larger than two. This can be done as follows when we run the code

```
mpiexec -np 10 ./prog.x  10000
```

In line 17 we define also the step length h. In lines 19 and 20 we use the broadcast function `MPI_Bcast`. We use this particular function because we want data on one processor (our master node) to be shared with all other processors. The broadcast function sends data to a group of processes. The MPI routine `MPI_Bcast` transfers data from one task to a group of others. The format for the call is in C++ given by the parameters of

```
MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);.
```

In case we have a floating point variable we need to declare

```
MPI_Bcast (&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

The general structure of this function is

```
MPI_Bcast( void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

All processes call this function, both the process sending the data (with rank zero) and all the other processes in `MPI_COMM_WORLD`. Every process has now copies of n and h, the number of mesh points and the step length, respectively.

We transfer the addresses of n and h. The second argument represents the number of data sent. In case of a one-dimensional array, one needs to transfer the number of array elements. If you have an $n \times m$ matrix, you must transfer $n \times m$. We need also to specify whether the variable type we transfer is a non-numerical such as a logical or character variable or numerical of the integer, real or complex type.

We transfer also an integer variable `int root`. This variable specifies the process which has the original copy of the data. Since we fix this value to zero in the call in lines 19 and 20, it means that it is the master process which keeps this information. For Fortran, this function is called via the statement

```
CALL MPI_BCAST(buff, count, MPI_TYPE, root, comm, ierr).
```

In lines 23-27, every process sums its own part of the final sum used by the rectangle rule. The receive statement collects the sums from all other processes in case `my_rank==0`, else an MPI send is performed.

The above function is not very elegant. Furthermore, the MPI instructions can be simplified by using the functions `MPI_Reduce` or `MPI_Allreduce`. The first function takes information from all processes and sends the result of the MPI operation to one process only, typically the master node. If we use `MPI_Allreduce`, the result is sent back to all processes, a feature which is useful when all nodes need the value of a joint operation. We limit ourselves to `MPI_Reduce` since it is only one process which will print out the final number of our calculation, The arguments to `MPI_Allreduce` are the same.

The `MPI_Reduce` function is defined as follows

```
MPI_Reduce( void *senddata, void* resultdata, int count, MPI_Datatype datatype,
    MPI_Op, int root, MPI_Comm comm)
```

The two variables `senddata` and `resultdata` are obvious, besides the fact that one sends the address of the variable or the first element of an array. If they are arrays they need to have the same size. The variable `count` represents the total dimensionality, 1 in case of just one variable, while `MPI_Datatype` defines the type of variable which is sent and received. The new feature is `MPI_Op`. `MPI_Op` defines the type of operation we want to do. There are many options, see again Refs. [1, 2, 4] for full list. In our case, since we are summing the rectangle contributions from every process we define `MPI_Op=MPI_SUM`. If we have an array or matrix we can search for the largest og smallest element by sending either `MPI_MAX` or `MPI_MIN`. If we want the location as well (which array element) we simply transfer `MPI_MAXLOC` or `MPI_MINOC`. If we want the product we write `MPI_PROD`. `MPI_Allreduce` is defined as

```
MPI_Allreduce( void *senddata, void* resultdata, int count, MPI_Datatype datatype,
    MPI_Op, MPI_Comm comm)
```

The function we list in the next example is the MPI extension of program1.cpp. The difference is that we employ only the trapezoidal rule. It is easy to extend this code to include gaussian quadrature or other methods.

It is also worth noting that every process has now its own starting and ending point. We read in the number of integration points n and the integration limits a and b. These are called `a` and `b`. They serve to define the local integration limits used by every process. The local integration limits are defined as

```
local_a = a + my_rank *(b-a)/numprocs
local_b = a + (my_rank-1) *(b-a)/numprocs.
```

These two variables are transfered to the method for the trapezoidal rule. These two methods return the local sum variable `local_sum`. `MPI_Reduce` collects all the local sums and returns the total sum, which is written out by the master node. The program below implements this. We have also added the possibility to measure the total time used by the code via the calls to `MPI_Wtime`.

http://folk.uio.no/mhjensen/compphys/programs/chapter05/program6.cpp

```cpp
//   Trapezoidal rule and numerical integration using MPI with MPI_Reduce
using namespace std;
#include <mpi.h>
#include <iostream>

//    Here we define various functions called by the main program

double int_function(double );
double trapezoidal_rule(double , double , int , double (*)(double));

//  Main function begins here
int main (int nargs, char* args[])
{
  int n, local_n, numprocs, my_rank;
  double a, b, h, local_a, local_b, total_sum, local_sum;
  double time_start, time_end, total_time;
  // MPI initializations
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  time_start = MPI_Wtime();
  // Fixed values for a, b and n
  a = 0.0 ; b = 1.0; n = 1000;
  h = (b-a)/n; // h is the same for all processes
  local_n = n/numprocs; // make sure n > numprocs, else integer division gives zero
  // Length of each process' interval of
  // integration = local_n*h.
  local_a = a + my_rank*local_n*h;
  local_b = local_a + local_n*h;
  total_sum = 0.0;
  local_sum = trapezoidal_rule(local_a, local_b, local_n, &int_function);
  MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  time_end = MPI_Wtime();
  total_time = time_end-time_start;
  if ( my_rank == 0) {
    cout << "Trapezoidal rule = " << total_sum << endl;
    cout << "Time = " << total_time << " on number of processors: " << numprocs <<
        endl;
  }
  // End MPI
  MPI_Finalize ();
  return 0;
} // end of main program

// this function defines the function to integrate
double int_function(double x)
{
  double value = 4./(1.+x*x);
  return value;
} // end of function to evaluate

// this function defines the trapezoidal rule
double trapezoidal_rule(double a, double b, int n, double (*func)(double))
{
  double trapez_sum;
  double fa, fb, x, step;
  int   j;
  step=(b-a)/((double) n);
  fa=(*func)(a)/2. ;
  fb=(*func)(b)/2. ;
  trapez_sum=0.;
  for (j=1; j <= n-1; j++){
    x=j*step+a;
    trapez_sum+=(*func)(x);
  }
  trapez_sum=(trapez_sum+fb+fa)*step;
  return trapez_sum;
} // end trapezoidal_rule
```

An obvious extension of this code is to read from file or screen the integration variables. One could also use the program library to call a particular integration method.

5.7 An Integration Class

We end this chapter by presenting the usage of the integral class defined in the program library. Here we have defined two header files, the `Function.h` and the `Integral.h` files. The program below uses the classes defined in these header files to compute the integral

$$\int_0^1 \exp(x)\cos(x).$$

```
#include <cmath>
#include <iostream>
#include "Function.h"
#include "Integral.h"
```

```
using namespace std;

class ExpCos: public Function{
  public:
    // Default constructor
    ExpCos(){}

    // Overloaded function operator().
    // Override the function operator() of the parent class.
    double operator()(double x){
      return exp(x)*cos(x);
    }
};

int main(){
  // Declare first an object of the function to be integrated
  ExpCos f;
  // Set integration bounds
  double a = 0.0;  // Lower bound
  double b = 1.0;  // Upper bound
  int npts = 100;  // Number of integration points


  // Declared (lhs) and instantiate an integral object of type Trapezoidal
  Integral *trapez = new Trapezoidal(a, b, npts, f);
  Integral *midpt = new MidPoint(a, b, npts, f);
  Integral *gl  = new Gauss_Legendre(a,b,npts, f);

  // Evaluate the integral of the function ExpCos and assign its
  // value to the variable result;
  double resultTP = trapez->evaluate();
  double resultMP = midpt->evaluate();
  double resultGL = gl->evaluate();

  // Print the result to screen
  cout << "Result with trapezoidal : " << resultTP << endl;
  cout << "Result with mid-point  : " << resultMP << endl;
  cout << "Result with Gauss-Legendre: " << resultGL << endl;
}
```

The header file `Function.h` is defined as

http://folk.uio.no/mhjensen/compphys/programs/chapter05/cpp/Function.h

```
/**
* @file Function.h
* Interface for mathematical functions with one or more independent variables.
* The subclasses are implemented as functors, i.e., objects behaving as functions.
* They overload the function operator().
```

```
*
* Example Usage:
// 1. Declare a functor, i.e., an object which
// overloads the function operator().
class Squared: public Function{
  public:
    // Overload function operator()
    double operator()(double x=0.0){
      return x*x;
    }
};

int main(){
  // Instance an object Functor
  Squared f;

  // Use the instance of the object as a normal function
  cout << f(3.0) << endl;
}
@endcode
*
**/

#ifndef FUNCTION_H
#define FUNCTION_H

#include "Array.h"

class Function{
  public:

  //! Destructor
  virtual ~Function(){}; // Not needed here.

    /**
    * @brief Overload the function operator().
    *
    * Used for evaluating functions with one independent variable.
    *
    **/
    virtual double operator()(double x){}

    /**
    * @brief Overload the function operator().
    *
    * Used for evaluating functions with more than one independent variable.
    **/
    virtual double operator()(const Array<double>& x){}
};
```

```
#endif
```

The header file `Integral.h` contains, with an example on how to use it, the following statements

http://folk.uio.no/mhjensen/compphys/programs/chapter05/cpp/Integral.h

```

#ifndef INTEGRAL_H
#define INTEGRAL_H

#include "Array.h"
#include "Function.h"
#include <cmath>

class Integral{
 protected:  // Access in the subclasses.
   double a; // Lower limit of integration.
   double b; // Upper limit of integration.
   int npts; // Number of integration points.
   Function &f; // Function to be integrated.

 public:

   /**
   * @brief Constructor.
   *
   * @param lower_. Lower limit of integration.
   * @param upper_. Upper limit of integration.
   * @param npts_. Number of points of integration.
   * @param f_. Reference to a functor representing the function to be integrated.
   **/
   Integral(double lower_, double upper_, int npts_, Function &f_);

   //! Destructor
   virtual ~Integral(){}

   /**
   * @brief Evaluate the integral.
   * @return The value of the integral in double precision.
   **/
   virtual double evaluate()=0;


   // virtual forloop

}; // End class Integral

class Trapezoidal: public Integral{
 private:
   double h;  // Step size.

  public:
   /**
   * @brief Constructor.
   *
   * @param lower_. Lower limit of integration.
   * @param upper_. Upper limit of integration.
   * @param npts_. Number of points of integration.
   * @param f_. Reference to a functor representing the function to be integrated.
   **/
   Trapezoidal(double lower_, double upper_, int npts_, Function &f_);

   //! Destructor
   ~Trapezoidal(){}

   /**
   * Evaluate the integral of a function f using the trapezoidal rule.
   * @return The value of the integral in double precision.
   **/
   double evaluate();
}; // End class Trapezoidal

class MidPoint: public Integral{
  private:
   double h;   // Step size.

  public:
   /**
   * @brief Constructor.
   *
   * @param lower_. Lower limit of integration.
   * @param upper_. Upper limit of integration.
   * @param npts_. Number of points of integration.
   * @param f_. Reference to a functor representing the function to be integrated.
   **/
   MidPoint(double lower_, double upper_, int npts_, Function &f_);

   //! Destructor
   ~MidPoint(){}

   /**
   * Evaluate the integral of a function f using the midpoint approximation.
   *
   * @return The value of the integral in double precision.
   **/
   double evaluate();
};
```

```
class Gauss_Legendre: public Integral{
 private:
   static const double ZERO = 1.0E-10;
   static const double PI = 3.14159265359;
   double h;

 public:
   /**
   * @brief Constructor.
   *
   * @param lower_. Lower limit of integration.
   * @param upper_. Upper limit of integration.
   * @param npts_. Number of points of integration.
   * @param f_. Reference to a functor representing the function to be integrated.
   **/
   Gauss_Legendre(double lower_, double upper_, int npts_, Function &f_);

   //! Destructor
   ~Gauss_Legendre(){}

   /**
   * Evaluate the integral of a function f using the Gauss-Legendre approximation.
   *
   * @return The value of the integral in double precision.
   **/
   double evaluate();
};
#endif
```

5.8 Exercises

Use Lagrange's interpolation formula for a second-order polynomial

$$P_2(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}y_2 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}y_1 + \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}y_0,$$

and insert this formula in the integral

$$\int_{-h}^{+h} f(x)dx \approx \int_{-h}^{+h} P_2(x)dx,$$

and derive Simpson's rule. You need to define properly the values x_0, x_1 and x_2 and link them with the integration limits x_0-h and x_0+h. Simpson's formula reads

$$\int_{-h}^{+h} f(x)dx = \frac{h}{3}(f_h + 4f_0 + f_{-h}) + O(h^5).$$

Write thereafter a class which implements both the Trapezoidal rule and Simpson's rule. You can for example follow the example given in the last section of this chapter. You can look up the header file for this class at http://folk.uio.no/mhjensen/compphys/programs/c

Write a program which then uses the above class containing the Trapezoidal rule and Simpson's rule to implement the adaptive algorithm discussed in section 5.3. Compute the integrals

$$I = \int_0^1 \frac{4}{1+x^2} = \pi,$$

and

$$I = \int_0^\infty x \exp{(-x)} \sin x = \frac{1}{2}.$$

Discuss strategies for choosing the integration limits using these methods

Add now to your integration class the possibility for extrapolating $h \to 0$ using Richardson's deferred extrapolation technique, see Eq. (3.13) and exercise 3.2 in chapter 3.

Write a class which includes your own functions for Gaussian quadrature using Legendre, Hermite and Laguerre polynomials. You can write your own functions for these methods or use those included with the programs of this book. For the latter see for example the programs in the directory programs/chapter05. The functions are called gausslegendre.cpp, gausshermite.cpp and gausslaguerre.cpp.

Use the Legendre and Laguerre polynomials to evaluate again

$$I = \int_0^\infty x \exp{(-x)} \sin x = \frac{1}{2}.$$

The task here is to integrate a six-dimensional integral which is used to determine the ground state correlation energy between two electrons in a helium atom. The integral appears in many quantum mechanical applications. However, if you are not too familiar with quantum mechanics, you can simply look at the mathematical details. We will employ both Gauss-Legendre and Gauss-Laguerre quadrature. Furthermore, you will need to parallelize your code. You can use your class from the previous problem.

We assume that the wave function of each electron can be modelled like the single-particle wave function of an electron in the hydrogen atom. The single-particle wave function for an electron i in the $1s$ state is given in terms of a dimensionless variable (the wave function is not properly normalized)

$$\mathbf{r}_i = x_i \mathbf{e}_x + y_i \mathbf{e}_y + z_i \mathbf{e}_z,$$

as

$$\psi_{1s}(\mathbf{r}_i) = e^{-\alpha r_i},$$

where α is a parameter and

$$r_i = \sqrt{x_i^2 + y_i^2 + z_i^2}.$$

We will fix $\alpha = 2$, which should correspond to the charge of the helium atom $Z = 2$.

The ansatz for the wave function for two electrons is then given by the product of two so-called $1s$ wave functions as

$$\Psi(\mathbf{r}_1, \mathbf{r}_2) = e^{-\alpha(r_1+r_2)}.$$

Note that it is not possible to find a closed-form solution to Schrödinger's equation for two interacting electrons in the helium atom.

The integral we need to solve is the quantum mechanical expectation value of the correlation energy between two electrons which repel each other via the classical Coulomb interaction, namely

$$\langle \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \rangle = \int_{-\infty}^{\infty} d\mathbf{r}_1 d\mathbf{r}_2 e^{-2\alpha(r_1+r_2)} \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|}.$$

Note that our wave function is not normalized. There is a normalization factor missing, but for this project we don't need to worry about that.

This integral can be solved in closed form and the answer is $5\pi^2/16^2$. Can you derive this value?

1. Use Gauss-Legendre quadrature and compute the integral by integrating for each variable $x_1, y_1, z_1, x_2, y_2, z_2$ from $-\infty$ to ∞. How many mesh points do you need before the results converges at the level of the third leading digit? Hint: the single-particle wave function $e^{-\alpha r_i}$ is more or less zero at $r_i \approx ?$ (find the appropriate limit). You can therefore replace the integration limits $-\infty$ and ∞ with $-?$ and $?$, respectively. You need to check that this approximation is satisfactory, that is, make a plot of the function and check if the abovementioned limits are appropriate. You need also to account for the potential problems which may arise when $|\mathbf{r}_1 - \mathbf{r}_2| = 0$.

2. The Legendre polynomials are defined for $x \in [-1, 1]$. The previous exercise gave a very unsatisfactory ad hoc procedure. We wish to improve our results. It can therefore be useful to change to another coordinate frame and employ the Laguerre polynomials. The Laguerre polynomials are defined for $x \in [0, \infty)$ and if we change to spherical coordinates

$$d\mathbf{r}_1 d\mathbf{r}_2 = r_1^2 dr_1 r_2^2 dr_2 dcos(\theta_1) dcos(\theta_2) d\phi_1 d\phi_2,$$

with

$$\frac{1}{r_{12}} = \frac{1}{\sqrt{r_1^2 + r_2^2 - 2r_1 r_2 cos(\beta)}}$$

and

$$cos(\beta) = cos(\theta_1)cos(\theta_2) + sin(\theta_1)sin(\theta_2)cos(\phi_1 - \phi_2))$$

we can rewrite the above integral with different integration limits. Find these limits and replace the Gauss-Legendre approach in a) with Laguerre polynomials. Do your results improve? Compare with the results from a).

3. Make a detailed analysis of the time used by both methods and compare your results. Parallelize your codes and check that you have an optimal speed up.

Bibliography

[1] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1999.

[2] G. E. Karniadakis and R. M. Kirby II. *Parallel scientific computing in C++ and MPI*. Cambridge, 2005.

[3] R. Kress. *Numerical Analysis*. Springer, 1998.

[4] M. Snir, S. Otto, S. Huss-Ledermann, D. Walker, and J. Dongarra. *MPI, the Complete Reference, Vols I and II*. The MIT Press, 1998.

[5] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer Verlag, 1983.

Part II

Linear Algebra and Eigenvalues

Chapter 6

Linear Algebra

6.1 Introduction

This chapter introduces several matrix related topics, from the solution of linear equations, computing determinants, conjugate-gradient methods, spline interpolation to efficient handling of matrices.

In this chapter we deal with basic matrix operations, such as the solution of linear equations, calculate the inverse of a matrix, its determinant etc. The solution of linear equations is an important part of numerical mathematics and arises in many applications in the sciences. Here we focus in particular on so-called direct or elimination methods, which are in principle determined through a finite number of arithmetic operations. Iterative methods will also be discussed.

This chapter serves also the purpose of introducing important programming details such as handling memory allocation for matrices and the usage of the libraries which follow these lectures.

The algorithms we describe and their original source codes are taken from the widely used software package LAPACK [1], which follows two other popular packages developed in the 1970s, namely EISPACK and LINPACK. The latter was developed for linear equations and least square problems while the former was developed for solving symmetric, unsymmetric and generalized eigenvalue problems. From LAPACK's website http://www.netlib.org it is possible to download for free all source codes from this library. Both C++ and Fortran versions are available. Another important library is BLAS [12], which stands for Basic Linear Algebra Subprogram. It contains efficient codes for algebraic operations on vectors, matrices and vectors and matrices. Basically all modern supercomputer include this library, with efficient algorithms. Else, Matlab offers a very efficient programming environment for dealing with matrices. The classic text from where we have taken most of the formalism exposed here is the book on matrix computations by Golub and Van Loan [9]. Good recent introductory texts are Kincaid and Cheney [10] and Datta

[4]. For more advanced ones see Trefethen and Bau III [17], Kress [11] and Demmel [5]. Ref. [9] contains an extensive list of textbooks on eigenvalue problems and linear algebra. LAPACK [1] contains also extensive listings to the research literature on matrix computations. For the introduction of the auxiliary library Blitz++ [18], which allows for a very efficient way of handling arrays in C++ we refer to the online manual at http://www.oonumerics.org. A library we highly recommend is Armadillo, see http://arma.sourceforge.org. Armadillo is an open-source C++ linear algebra library aiming towards a good balance between speed and ease of use. Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions. Various matrix and vector operations are provided through optional integration with BLAS and LAPACK.

6.2 Mathematical Intermezzo

The matrices we will deal with are primarily square real symmetric or hermitian ones, assuming thereby that an $n\times n$ matrix $\mathbf{A}\in\mathbb{R}^{n\times n}$ for a real matrix[1] and $\mathbf{A}\in\mathbb{C}^{n\times n}$ for a complex matrix. For the sake of simplicity, we take a matrix $\mathbf{A}\in\mathbb{R}^{4\times 4}$ and a corresponding identity matrix $\mathbf{I}$

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \qquad \mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{6.1}$$

where $a_{ij}\in\mathbb{R}$. The inverse of a matrix, if it exists, is defined by

$$\mathbf{A}^{-1}\cdot\mathbf{A} = I.$$

In the following discussion, matrices are always two-dimensional arrays while vectors are one-dimensional arrays. In our nomenclature we will restrict boldfaced capitals letters such as $\mathbf{A}$ to represent a general matrix, which is a two-dimensional array, while a_{ij} refers to a matrix element with row number i and column number j. Similarly, a vector being a one-dimensional array, is labelled $\mathbf{x}$ and represented as

[1] A reminder on mathematical symbols may be appropriate here. The symbol $\mathbb{R}$ is the set of real numbers. Correspondingly, $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{C}$ represent the set of natural, integer and complex numbers, respectively. A symbol like $\mathbb{R}^n$ stands for an n-dimensional real Euclidean space, while $C[a,b]$ is the space of real or complex-valued continuous functions on the interval $[a,b]$, where the latter is a closed interval. Similalry, $C^m[a,b]$ is the space of m-times continuously differentiable functions on the interval $[a,b]$. For more symbols and notations, see the main text.

(for a real vector)

$$\mathbf{x} \in \mathbb{R}^n \iff \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix},$$

with pertinent vector elements $x_i \in \mathbb{R}$. Note that this notation implies $x_i \in \mathbb{R}^{4\times 1}$ and that the members of $\mathbf{x}$ are column vectors. The elements of $x_i \in \mathbb{R}^{1\times 4}$ are row vectors.

Table 6.2 lists some essential features of various types of matrices one may encounter. Some of the matrices we will encounter are listed here

Table 6.1: Matrix properties

Relations	Name	matrix elements
$\mathbf{A} = \mathbf{A}^T$	symmetric	$a_{ij} = a_{ji}$
$\mathbf{A} = \left(\mathbf{A}^T\right)^{-1}$	real orthogonal	$\sum_k a_{ik}a_{jk} = \sum_k a_{ki}a_{kj} = \delta_{ij}$
$\mathbf{A} = \mathbf{A}^*$	real matrix	$a_{ij} = a_{ij}^*$
$\mathbf{A} = \mathbf{A}^\dagger$	hermitian	$a_{ij} = a_{ji}^*$
$\mathbf{A} = \left(\mathbf{A}^\dagger\right)^{-1}$	unitary	$\sum_k a_{ik}a_{jk}^* = \sum_k a_{ki}^*a_{kj} = \delta_{ij}$

1. Diagonal if $a_{ij} = 0$ for $i \neq j$,

2. Upper triangular if $a_{ij} = 0$ for $i > j$, which for a 4×4 matrix is of the form

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{nn} \end{pmatrix}$$

3. Lower triangular if $a_{ij} = 0$ for $i < j$

$$\begin{pmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

4. Upper Hessenberg if $a_{ij} = 0$ for $i > j+1$, which is similar to a upper triangular except that it has non-zero elements for the first subdiagonal row

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

5. Lower Hessenberg if $a_{ij} = 0$ for $i < j+1$

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

6. Tridiagonal if $a_{ij} = 0$ for $|i-j| > 1$

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

There are many more examples, such as lower banded with bandwidth p for $a_{ij} = 0$ for $i > j+p$, upper banded with bandwidth p for $a_{ij} = 0$ for $i < j+p$, block upper triangular, block lower triangular etc.

For a real $n \times n$ matrix $\mathbf{A}$ the following properties are all equivalent

1. If the inverse of $\mathbf{A}$ exists, $\mathbf{A}$ is nonsingular.
2. The equation $\mathbf{Ax} = 0$ implies $\mathbf{x} = 0$.
3. The rows of $\mathbf{A}$ form a basis of $\mathbb{R}^n$.
4. The columns of $\mathbf{A}$ form a basis of $\mathbb{R}^n$.
5. $\mathbf{A}$ is a product of elementary matrices.
6. 0 is not an eigenvalue of $\mathbf{A}$.

The basic matrix operations that we will deal with are addition and subtraction

$$\mathbf{A} = \mathbf{B} \pm \mathbf{C} \Longrightarrow a_{ij} = b_{ij} \pm c_{ij}, \tag{6.2}$$

scalar-matrix multiplication

$$\mathbf{A} = \gamma\mathbf{B} \Longrightarrow a_{ij} = \gamma b_{ij},$$

vector-matrix multiplication

$$\mathbf{y} = \mathbf{Ax} \Longrightarrow y_i = \sum_{j=1}^{n} a_{ij}x_j, \tag{6.3}$$

matrix-matrix multiplication

$$\mathbf{A} = \mathbf{BC} \Longrightarrow a_{ij} = \sum_{k=1}^{n} b_{ik}c_{kj}, \tag{6.4}$$

transposition

$$\mathbf{A} = \mathbf{B}^T \Longrightarrow a_{ij} = b_{ji},$$

and if $\mathbf{A} \in \mathbb{C}^{n\times n}$, conjugation results in

$$\mathbf{A} = \overline{\mathbf{B}}^T \Longrightarrow a_{ij} = \overline{b}_{ji},$$

where a variable $\overline{z} = x - \imath y$ denotes the complex conjugate of $z = x + \imath y$. In a similar way we have the following basic vector operations, namely addition and subtraction

$$\mathbf{x} = \mathbf{y} \pm \mathbf{z} \Longrightarrow x_i = y_i \pm z_i,$$

scalar-vector multiplication

$$\mathbf{x} = \gamma\mathbf{y} \Longrightarrow x_i = \gamma y_i,$$

vector-vector multiplication (called Hadamard multiplication)

$$\mathbf{x} = \mathbf{y}\mathbf{z} \Longrightarrow x_i = y_i z_i,$$

the inner or so-called dot product

$$c = \mathbf{y}^T\mathbf{z} \Longrightarrow c = \sum_{j=1}^{n} y_j z_j, \tag{6.5}$$

with c a constant and the outer product, which yields a matrix,

$$\mathbf{A} = \mathbf{y}\mathbf{z}^T \Longrightarrow a_{ij} = y_i z_j, \tag{6.6}$$

Other important operations are vector and matrix norms. A class of vector norms are the so-called p-norms

$$||\mathbf{x}||_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}},$$

where $p \geq 1$. The most important are the 1, 2 and ∞ norms given by

$$||\mathbf{x}||_1 = |x_1| + |x_2| + \cdots + |x_n|,$$

$$||\mathbf{x}||_2 = (|x_1|^2 + |x_2|^2 + \cdots + |x_n|^2)^{\frac{1}{2}} = (\mathbf{x}^T\mathbf{x})^{\frac{1}{2}},$$

and

$$||\mathbf{x}||_\infty = \max |x_i|,$$

for $1 \leq i \leq n$. From these definitions, one can derive several important relations, of which the so-called Cauchy-Schwartz inequality is of great importance for many algorithms. For any $\mathbf{x}$ and $\mathbf{y}$ being real-valued or complex-valued quantities, the inner product space satisfies

$$|\mathbf{x}^T\mathbf{y}| \leq ||\mathbf{x}||_2||\mathbf{y}||_2,$$

and the equality is obeyed only if $\mathbf{x}$ and $\mathbf{y}$ are linearly dependent. An important relation which follows from the Cauchy-Schwartz relation is the famous triangle relation, which states that for any $\mathbf{x}$ and $\mathbf{y}$ in a real or complex, the inner product space satisfies

$$||\mathbf{x}+\mathbf{y}||_2 \leq ||\mathbf{x}||_2 + ||\mathbf{y}||_2.$$

Proofs can be found in for example Ref. [9]. As discussed in chapter 2, the analysis of the relative error is important in our studies of loss of numerical precision. Using a vector norm we can define the relative error for the machine representation of a vector $\mathbf{x}$. We assume that $fl(\mathbf{x}) \in \mathbb{R}^n$ is the machine representation of a vector $\mathbf{x} \in \mathbb{R}^n$. If $\mathbf{x} \neq 0$, we define the relative error as

$$\varepsilon = \frac{||fl(\mathbf{x}) - \mathbf{x}||}{||\mathbf{x}||}.$$

Using the ∞-norm one can define a relative error that can be translated into a statement on the correct significant digits of $fl(\mathbf{x})$,

$$\frac{||fl(\mathbf{x}) - \mathbf{x}||_\infty}{||\mathbf{x}||_\infty} \approx 10^{-l},$$

where the largest component of $fl(\mathbf{x})$ has roughly l correct significant digits.

We can define similar matrix norms as well. The most frequently used are the Frobenius norm

$$||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2},$$

and the p-norms

$$||\mathbf{A}||_p = \frac{||\mathbf{Ax}||_p}{||\mathbf{x}||_p},$$

assuming that $\mathbf{x} \neq 0$. We refer the reader to the text of Golub and Van Loan [9] for a further discussion of these norms.

The way we implement these operations will be discussed below, as it depends on the programming language we opt for.

6.3 Programming Details

Many programming problems arise from improper treatment of arrays. In this section we will discuss some important points such as array declaration, memory allocation and array transfer between functions. We distinguish between two cases: (a) array declarations where the array size is given at compilation time, and (b) where the array size is determined during the execution of the program, so-called dymanic memory allocation. Useful references on C++ programming details, in particular on the use of pointers and memory allocation, are Reek's text [15] on pointers in C, Berryhill's monograph [3] on scientific programming in C++ and finally Franek's text [8] on memory as a programming concept in C and C++. Good allround texts on C++ programming in engineering and science are the books by Flowers [7] and Barton and Nackman [2]. See also the online lecture notes on C++ at http://heim.ifi.uio.no/~hpl/INF-VERK4830. For Fortran we recommend the online lectures at http://folk.uio.no/gunnarw/INF-VERK4820. These web pages contain extensive references to other C++ and Fortran resources. Both web pages contain enough material, lecture notes and exercises, in order to serve as material for own studies.

Figure 6.1: Segmentation fault, again and again! Alas, this is a situation you will most likely end up in, unless you initialize, access, allocate and deallocate properly your arrays. Many program development environments such as Dev C++ at www.bloodshed.net provide debugging possibilities. Beware however that there may be segmentation errors which occur due to errors in libraries of the operating system. (Drawing: courtesy by Victoria Popsueva 2003.)

6.3.1 Declaration of fixed-sized vectors and matrices

In the program below we discuss some essential features of vector and matrix handling where the dimensions are declared in the program code.

In **line a** we have a standard C++ declaration of a vector. The compiler reserves memory to store five integers. The elements are `vec[0], vec[1],....,vec[4]`. Note that the numbering of elements starts with zero. Declarations of other data types are similar, including structure data.

The symbol vec is an element in memory containing the address to the first element `vec[0]` and is a pointer to a vector of five integer elements.

In **line b** we have a standard fixed-size C++ declaration of a matrix. Again the elements start with zero, `matr[0][0], matr[0][1], ....., matr[0][4], matr[1][0],....` This sequence of elements also shows how data are stored in memory. For example, the element `matr[1][0]` follows `matr[0][4]`. This is important in order to produce an efficient code and avoid memory stride.

There is one further important point concerning matrix declaration. In a similar way as for the symbol **vec**, **matr** is an element in memory which contains an address to a vector of three elements, but now these elements are not integers. Each element is a vector of five integers. This is the correct way to understand the declaration in **line b**. With respect to pointers this means that matr is *pointer-to-a-pointer-to-an-integer* which we can write $**$matr. Furthermore $*$matr is *a-pointer-to-a-pointer* of five integers. This interpretation is important when we want to transfer vectors and matrices to a function.

In **line c** we transfer `vec[]` and `matr[][]` to the function sub_1(). To be specific, we transfer the addresses of `vec[]` and matr[][] to sub_1().

In **line d** we have the function definition of subfunction(). The **int** vec[] is a pointer to an integer. Alternatively we could write **int** $*$vec. The first version is better. It shows that it is a vector of several integers, but not how many. The second version could equally well be used to transfer the address to a single integer element. Such a declaration does not distinguish between the two cases.

The next definition is **int** `matr[][5]`. This is a pointer to a vector of five elements and the compiler must be told that each vector element contains five integers. Here an alternative version could be int ($*$matr)[5] which clearly specifies that matr is a pointer to a vector of five integers.

```
 int main()
{
  int k,m, row = 3, col = 5;
  int  vec[5];   // line a
  int  matr[3][5]; // line b
  // Fill in vector vec
  for (k = 0; k < col; k++) vec[k] = k;
  // fill in matr
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++) matr[m][k] = m + 10*k;
  }
  // write out the vector
  cout << `` Content of vector vec:'' << endl;
  for (k = 0; k < col; k++){
     cout << vec[k] << endl;
  }
  // Then write out the matrix
  cout << `` Content of matrix matr:'' << endl;
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++){
       cout << matr[m][k] << endl;
     }
  }
  subfunction(row, col, vec, matr); // line c
  return 0;
} // end main function
```

```
void subfunction(int row, int col, int vec[], int matr[][5]); // line d
{
  int k, m;
  // write out the vector
  cout << `` Content of vector vec in subfunction:'' << endl;
  for (k = 0; k < col; k++){
     cout << vec[k] << endl;
  }
  // Then write out the matrix
  cout << `` Content of matrix matr in subfunction:'' << endl;
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++){
       cout << matr[m][k] << endl;
     }
  }
} // end of function subfunction
```

There is at least one drawback with such a matrix declaration. If we want to change the dimension of the matrix and replace 5 by something else we have to do the same change in all functions where this matrix occurs.

There is another point to note regarding the declaration of variables in a function which includes vectors and matrices. When the execution of a function terminates, the memory required for the variables is released. In the present case memory for all variables in main() are reserved during the whole program execution, but variables which are declared in subfunction() are released when the execution returns to main().

6.3.2 Runtime Declarations of Vectors and Matrices in C++

We change thereafter our program in order to include dynamic allocation of arrays. As mentioned in the previous subsection a fixed size declaration of vectors and matrices before compilation is in many cases bad. You may not know beforehand the actually needed sizes of vectors and matrices. In large projects where memory is a limited factor it could be important to reduce memory requirement for matrices which are not used any more. In C an C++ it is possible and common to postpone size declarations of arrays untill you really know what you need and also release memory reservations when it is not needed any more. The following program shows how we could change the previous one with static declarations to dynamic allocation of arrays.

```
 int main()
{
  int k,m, row = 3, col = 5;
  int  vec[5];   // line a
  int  matr[3][5]; // line b

  cout << `` Read in number of rows'' << endl; // line c
  cin >> row;
  cout << `` Read in number of columns'' << endl;
  cin >> col;

  vec = new int[col];                 // line d
  matr = (int **)matrix(row,col,sizeof(int)); // line e
  // Fill in vector vec
  for (k = 0; k < col; k++) vec[k] = k;
  // fill in matr
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++) matr[m][k] = m + 10*k;
  }
  // write out the vector
  cout << `` Content of vector vec:'' << endl;
  for (k = 0; k < col; k++){
     cout << vec[k] << endl;
  }
  // Then write out the matrix
  cout << `` Content of matrix matr:'' << endl;
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++){
       cout << matr[m][k] << endl;
     }
  }
  subfunction(row, col, vec, matr); // line f
  free_matrix((void **) matr); // line g
  delete vec[];
  return 0;
} // end main function

void subfunction(int row, int col, int vec[], int matr[][5]); // line h
{
  int k, m;
  // write out the vector
  cout << `` Content of vector vec in subfunction:'' << endl;
  for (k = 0; k < col; k++){
     cout << vec[k] << endl;
  }
  // Then write out the matrix
  cout << `` Content of matrix matr in subfunction:'' << endl;
  for (m = 0; m < row; m++){
     for (k = 0; k < col ; k++){
       cout << matr[m][k] << endl;
     }
  }
} // end of function subfunction
```

In **line a** we declare a pointer to an integer which later will be used to store an address to the first element of a vector. Similarily, **line b** declares a pointer-to-a-pointer which will contain the address to a pointer of row vectors, each with col integers. This will then become a matrix with dimensionality [col][col]

In **line c** we read in the size of vec[] and matr[][] through the numbers row and col.

Next we reserve memory for the vector in **line d**. In **line e** we use a user-defined function to reserve necessary memory for matrix[row][col] and again matr contains the address to the reserved memory location.

The remaining part of the function main() are as in the previous case down to **line f**. Here we have a call to a user-defined function which releases the reserved memory of the matrix. In this case this is not done automatically.

In **line g** the same procedure is performed for vec[]. In this case the standard C++ library has the necessary function.

Next, in **line h** an important difference from the previous case occurs. First, the vector declaration is the same, but the matr declaration is quite different. The corresponding parameter in the call to sub_1[] in **line g** is a double pointer. Consequently, matr in **line h** must be a double pointer.

Except for this difference sub_1() is the same as before. The new feature in the program below is the call to the user-defined functions **matrix** and **free_matrix**. These functions are defined in the library file **lib.cpp**. The code for the dynamic memory allocation is given below.

http://folk.uio.no/compphys/programs/FYS3150/cpp/cpluspluslibrary/lib.cpp

```
 /*
  * The function
  *     void **matrix()
  * reserves dynamic memory for a two-dimensional matrix
  * using the C++ command new . No initialization of the elements.
  * Input data:
  *  int row    - number of rows
  *  int col    - number of columns
  *  int num_bytes- number of bytes for each
  *             element
  * Returns a void **pointer to the reserved memory location.
  */

void **matrix(int row, int col, int num_bytes)
 {
 int    i, num;
 char   **pointer, *ptr;

 pointer = new(nothrow) char* [row];
 if(!pointer) {
   cout << "Exception handling: Memory allocation failed";
```

```
    cout << " for "<< row << "row addresses !" << endl;
    return NULL;
  }
  i = (row * col * num_bytes)/sizeof(char);
  pointer[0] = new(nothrow) char [i];
  if(!pointer[0]) {
    cout << "Exception handling: Memory allocation failed";
    cout << " for address to " << i << " characters !" << endl;
    return NULL;
  }
  ptr = pointer[0];
  num = col * num_bytes;
  for(i = 0; i < row; i++, ptr += num ) {
    pointer[i] = ptr;
  }
  return (void **)pointer;
  } // end: function void **matrix()
```

As an alternative, you could write your own allocation and deallocation of matrices. This can be done rather straightforwardly with the following statements. Recall first that a matrix is represented by a double pointer that points to a contiguous memory segment holding a sequence of double* pointers in case our matrix is a double precision variable. Then each double* pointer points to a row in the matrix. A declaration like `double** A;` means that A$[i]$ is a pointer to the $i+1$-th row A$[i]$ and A$[i][j]$ is matrix entry (i,j). The way we would allocate memory for such a matrix of dimensionality $n\times n$ is for example using the following piece of code

```
int n;
double ** A;

A = new double*[n]
for ( i = 0; i < n; i++)
   A[i] = new double[N];
```

When we declare a matrix (a two-dimensional array) we must first declare an array of double variables. To each of this variables we assign an allocation of a single-dimensional array. A conceptual picture on how a matrix **A** is stored in memory is shown in Fig. 6.2.

Allocated memory should always be deleted when it is no longer needed. We free memory using the statements

```
for ( i = 0; i < n; i++)
   delete[] A[i];
delete[] A;
```

`delete[]A;`, which frees an array of pointers to matrix rows.

However, including a library like Blitz++ http://www.oonumerics.org or Armadillo makes life much easier when dealing with matrices.

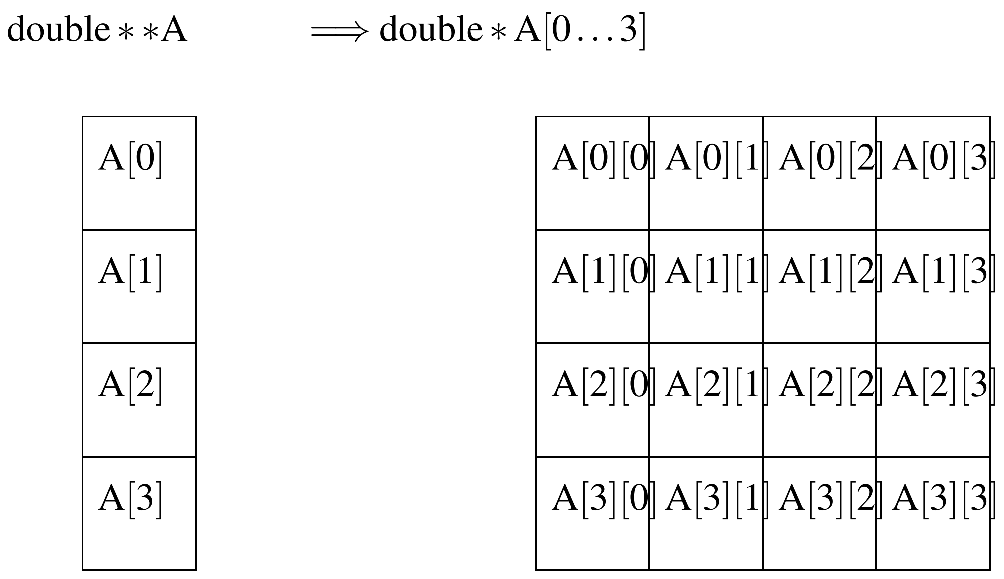
double * *A
⟹ double * A[0...3]
A[0]
A[1]
A[2]
A[3]
A[0][0] A[0][1] A[0][2] A[0][3]
A[1][0] A[1][1] A[1][2] A[1][3]
A[2][0] A[2][1] A[2][2] A[2][3]
A[3][0] A[3][1] A[3][2] A[3][3]

Figure 6.2: Conceptual representation of the allocation of a matrix in C++.

6.3.3 Matrix Operations and C++ and Fortran Features of Matrix handling

Many program libraries for scientific computing are written in Fortran, often also in older version such as Fortran 77. When using functions from such program libraries, there are some differences between C++ and Fortran encoding of matrices and vectors worth noticing. Here are some simple guidelines in order to avoid some of the most common pitfalls.

First of all, when we think of an $n \times n$ matrix in Fortran and C++, we typically would have a mental picture of a two-dimensional block of stored numbers. The computer stores them however as sequential strings of numbers. The latter could be stored as row-major order or column-major order. What do we mean by that? Recalling that for our matrix elements a_{ij}, i refers to rows and j to columns, we could store a matrix in the sequence $a_{11}a_{12}\dots a_{1n}a_{21}a_{22}\dots a_{2n}\dots a_{nn}$ if it is row-major order (we go along a given row i and pick up all column elements j) or it could be stored in column-major order $a_{11}a_{21}\dots a_{n1}a_{12}a_{22}\dots a_{n2}\dots a_{nn}$.

Fortran stores matrices in the latter way, i.e., by column-major, while C++ stores them by row-major. It is crucial to keep this in mind when we are dealing with matrices, because if we were to organize the matrix elements in the wrong way, important properties like the transpose of a real matrix or the inverse can be wrong, and obviously yield wrong physics. Fortran subscripts begin typically with 1, although it is no problem in starting with zero, while C++ starts with 0 for the first element. This means that $A(1,1)$ in Fortran is equivalent to $A[0][0]$ in C++. Moreover, since the sequential storage in memory means that nearby matrix elements are close to each other in the memory locations (and thereby easier to fetch) , operations involving e.g., additions of matrices may take more time if we do not respect the given ordering.

To see this, consider the following coding of matrix addition in C++ and Fortran. We have $n \times n$ matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ and we wish to evaluate $\mathbf{A} = \mathbf{B} + \mathbf{C}$ according to Eq. (6.2). In C++ this would be coded like

```
for(i=0 ; i < n ; i++) {
  for(j=0 ; j < n ; j++) {
    a[i][j]=b[i][j]+c[i][j]
  }
}
```

while in Fortran we would have

```
DO j=1, n
  DO i=1, n
    a(i,j)=b(i,j)+c(i,j)
  ENDDO
ENDDO
```

Fig. 6.3 shows how a 3×3 matrix $\mathbf{A}$ is stored in both row-major and column-major ways.

Interchanging the order of i and j can lead to a considerable enhancement in process time. In Fortran we write the above statements in a much simpler way `a=b+c`. However, the addition still involves $\sim n^2$ operations. Matrix multiplication or taking the inverse requires $\sim n^3$ operations. The matrix multiplication of Eq. (6.4) of two matrices $\mathbf{A} = \mathbf{BC}$ could then take the following form in C++

```
for(i=0 ; i < n ; i++) {
  for(j=0 ; j < n ; j++) {
    for(k=0 ; k < n ; k++) {
      a[i][j]+=b[i][k]*c[k][j]
    }
  }
}
```

and in Fortran we have

```
DO j=1, n
  DO i=1, n
    DO k = 1, n
      a(i,j)=a(i,j)+b(i,k)*c(k,j)
    ENDDO
  ENDDO
ENDDO
```

However, Fortran has an intrisic function called MATMUL, and the above three loops can be coded in a single statement `a=MATMUL(b,c)`. Fortran contains several array manipulation statements, such as dot product of vectors, the transpose of a matrix etc etc. The outer product of two vectors is however not included in Fortran. The coding of Eq. (6.6) takes then the following form in C++

```
for(i=0 ; i < n ; i++) {
  for(j=0 ; j < n ; j++) {
     a[i][j]+=x[i]*y[j]
  }
}
```

and in Fortran we have

```
DO j=1, n
  DO i=1, n
     a(i,j)=a(i,j)+x(j)*y(i)
  ENDDO
ENDDO
```

A matrix-matrix multiplication of a general $n \times n$ matrix with

$$a(i,j) = a(i,j) + b(i,k) * c(k,j),$$

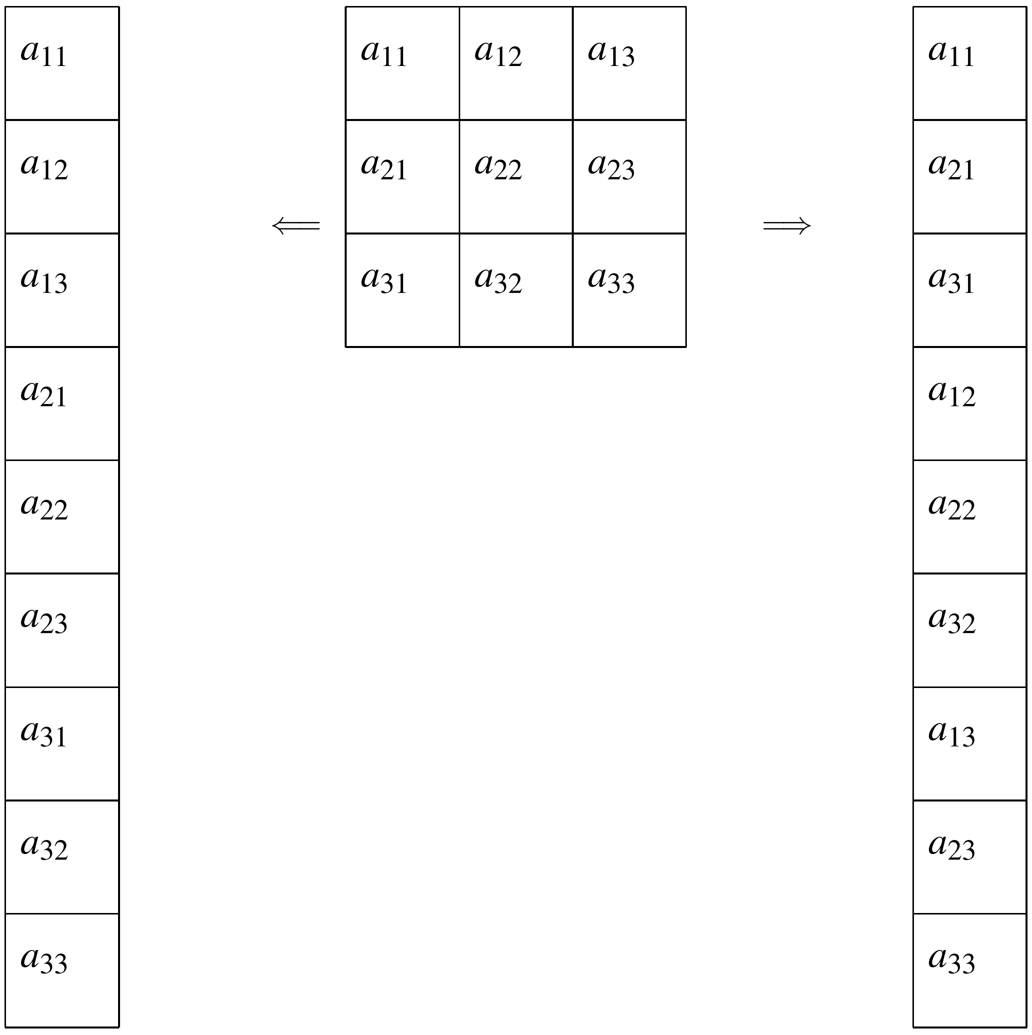

Figure 6.3: Row-major storage of a matrix to the left (C++ way) and column-major to the right (Fortran way).

in its inner loops requires a multiplication and an addition. We define now a flop (floating point operation) as one of the following floating point arithmetic operations, viz addition, subtraction, multiplication and division. The above two floating point operations (flops) are done n^3 times meaning that a general matrix multiplication requires $2n^3$ flops if we have a square matrix. If we assume that our computer performs 10^9 flops per second, then to perform a matrix multiplication of a 1000×1000 case should take two seconds. This can be reduced if we multiply two matrices which are upper triangular such as

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{44} \end{pmatrix}.$$

The multiplication of two upper triangular matrices $\mathbf{BC}$ yields another upper triangular matrix $\mathbf{A}$, resulting in the following C++ code

```
for(i=0 ; i < n ; i++) {
  for(j=i ; j < n ; j++) {
    for(k=i ; k < j ; k++) {
      a[i][j]+=b[i][k]*c[k][j]
    }
  }
}
```

The fact that we have the constraint $i \leq j$ leads to the requirement for the computation of a_{ij} of $2(j-i+1)$ flops. The total number of flops is then

$$\sum_{i=1}^{n}\sum_{j=1}^{n} 2(j-i+1) = \sum_{i=1}^{n}\sum_{j=1}^{n-i+1} 2j \approx \sum_{i=1}^{n} \frac{2(n-i+1)^2}{2},$$

where we used that $\sum_{j=1}^{n} j = n(n+1)/2 \approx n^2/2$ for large n values. Using in addition that $\sum_{j=1}^{n} j^2 \approx n^3/3$ for large n values, we end up with approximately $n^3/3$ flops for the multiplication of two upper triangular matrices. This means that if we deal with matrix multiplication of upper triangular matrices, we reduce the number of flops by a factor six if we code our matrix multiplication in an efficient way.

It is also important to keep in mind that computers are finite, we can thus not store infinitely large matrices. To calculate the space needed in memory for an $n \times n$ matrix with double precision, 64 bits or 8 bytes for every matrix element, one needs simply compute $n \times n \times 8$ bytes . Thus, if $n = 10000$, we will need close to 1GB of storage. Decreasing the precision to single precision, only halves our needs.

A further point we would like to stress, is that one should in general avoid fixed (at compilation time) dimensions of matrices. That is, one could always specify that a given matrix $\mathbf{A}$ should have size $A[100][100]$, while in the actual execution one may use only $A[10][10]$. If one has several such matrices, one may run out of memory, while the actual processing of the program does not imply that. Thus, we will always recommend that you use dynamic memory allocation, and deallocation of arrays when they are no longer needed. In Fortran one uses the intrisic functions **ALLOCATE** and **DEALLOCATE**, while C++ employs the functions **new** and **delete**.

Strassen's algorithm

As we have seen, the straightforward algorithm for matrix-matrix multiplication will require p multiplications and $p-1$ additions for each of the $m\times n$ elements. The total number of floating-point operations is then $mn(2p-1)\sim\mathscr{O}(mnp)$. When the matrices A and B can be divided into four equally sized blocks,

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \tag{6.7}$$

we get eight multiplications of smaller blocks,

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22} \\ A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22} \end{bmatrix}. \tag{6.8}$$

Strassen discovered in 1968 how the number of multiplications could be reduced from eight to seven [9]. Following Strassen's approach we define some intermediates,

$$\begin{aligned} S_1 &= A_{21}+A_{22}, & T_1 &= B_{12}-B_{11}, \\ S_2 &= S_1-A_{11}, & T_2 &= B_{22}-T_1, \\ S_3 &= A_{11}-A_{21}, & T_3 &= B_{22}-B_{12}, \\ S_4 &= A_{12}-S_2, & T_4 &= B_{21}-T_2, \end{aligned} \tag{6.9}$$

and need seven multiplications,

$$\begin{aligned} P_1 &= A_{11}B_{11}, & U_1 &= P_1+P_2, \\ P_2 &= A_{12}B_{21}, & U_2 &= P_1+P_4, \\ P_3 &= S_1T_1, & U_3 &= U_2+P_5, \\ P_4 &= S_2T_2, & U_4 &= U_3+P_7, \\ P_5 &= S_3T_3, & U_5 &= U_3+P_3, \\ P_6 &= S_4B_{22}, & U_6 &= U_2+P_3, \\ P_7 &= A_{22}T_4, & U_7 &= U_6+P_6, \end{aligned} \tag{6.10}$$

to find the resulting C matrix as

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} U_1 & U_7 \\ U_4 & U_5 \end{bmatrix}. \tag{6.11}$$

In spite of the seemingly additional work, we have reduced the number of multiplications from eight to seven. Since the multiplications are the computational bottleneck compared to addition and subtraction, the number of flops are reduced.

In the case of square $n\times n$ matrices with n equal to a power of two, $n=2^m$, the divided blocks will have $\frac{n}{2}=2^{m-1}$. Letting $f(m)$ be the number of flops needed for the full matrix and applying Strassen recursively we find the total number of flops to be

$$f(m)=7f(m-1)=7^2f(m-2)=\cdots=7^mf(0), \tag{6.12}$$

where $f(0)$ is the one floating-point operation needed for multiplication of two numbers (two $2^0\times 2^0$ matrices). For large matrices this can prove efficient, yielding a much better scaling,

$$\mathcal{O}\left(7^m\right)=\mathcal{O}\left(2^{\log_2 7^m}\right)=\mathcal{O}\left(2^{m\log_2 7}\right)=\mathcal{O}\left(n^{\log_2 7}\right)\approx\mathcal{O}\left(n^{2.807}\right), \tag{6.13}$$

effectively saving $7/8=12.5\%$ each time it is applied.

Fortran Allocate Statement and Mathematical Operations on Arrays

An array is declared in the declaration section of a program, module, or procedure using the dimension attribute. Examples include

```
REAL, DIMENSION (10) :: x,y
REAL, DIMENSION (1:10) :: x,y
INTEGER, DIMENSION (-10:10) :: prob
INTEGER, DIMENSION (10,10) :: spin
```

The default value of the lower bound of an array is 1. For this reason the first two statements are equivalent to the first. The lower bound of an array can be negative. The last two statements are examples of two-dimensional arrays.

Rather than assigning each array element explicitly, we can use an array constructor to give an array a set of values. An array constructor is a one-dimensional list of values, separated by commas, and delimited by "(/" and "/)". An example is

```
a(1:3) = (/ 2.0, -3.0, -4.0 /)
```

is equivalent to the separate assignments

```
a(1) = 2.0
a(2) = -3.0
a(3) = -4.0
```

One of the better features of Fortran is dynamic storage allocation. That is, the size of an array can be changed during the execution of the program. To see how the dynamic allocation works in Fortran, consider the following simple example where we set up a 4×4 unity matrix.

```
     ......
     IMPLICIT NONE
!    The definition of the matrix, using dynamic allocation
     REAL, ALLOCATABLE, DIMENSION(:,:) :: unity
!    The size of the matrix
     INTEGER :: n
!    Here we set the dim n=4
     n=4
! Allocate now place in memory for the matrix
     ALLOCATE ( unity(n,n) )
! all elements are set equal zero
     unity=0.
!    setup identity matrix
     DO i=1,n
       unity(i,i)=1.
     ENDDO
     DEALLOCATE ( unity)
     .......
```

We always recommend to use the deallocation statement, since this frees space in memory. If the matrix is transferred to a function from a calling program, one can transfer the dimensionality n of that matrix with the call. Another possibility is to determine the dimensionality with the `SIZE` function. Writing a statement like `n=SIZE(unity,DIM=1)` gives the number of rows, while using DIM=2 gives the number of columns. Note however that this involves an extra call to a function. If speed matters, one should avoid such calls.

6.4 Linear Systems

In this section we outline some of the most used algorithms to solve sets of linear equations. These algorithms are based on Gaussian elimination [9, 11] and will allow us to catch several birds with a stone. We will show how to rewrite a matrix $\mathbf{A}$ in terms of an upper and a lower triangular matrix, from which we easily can solve linear equation, compute the inverse of $\mathbf{A}$ and obtain the determinant. We start with Gaussian elimination, move to the more efficient LU-algorithm, which forms the basis for many linear algebra applications, and end the discussion with special cases such as the Cholesky decomposition and linear system of equations with a tridiagonal matrix.

We begin however with an example which demonstrates the importance of being able to solve linear equations. Suppose we want to solve the following boundary value equation

$$-\frac{d^2u(x)}{dx^2} = f(x,u(x)),$$

with $x \in (a,b)$ and with boundary conditions $u(a) = u(b) = 0$. We assume that f is a continuous function in the domain $x \in (a,b)$. Since, except the few cases where it is possible to find analytic solutions, we will seek approximate solutions, we choose to represent the approximation to the second derivative from the previous chapter

$$f'' = \frac{f_h - 2f_0 + f_{-h}}{h^2} + O(h^2).$$

We subdivide our interval $x \in (a,b)$ into n subintervals by setting $x_i = a + ih$, with $i = 0,1,\dots,n+1$. The step size is then given by $h = (b-a)/(n+1)$ with $n \in \mathbb{N}$. For the internal grid points $i = 1,2,\dots n$ we replace the differential operator with the above formula resulting in

$$u''(x_i) \approx \frac{u(x_i+h) - 2u(x_i) + u(x_i-h)}{h^2},$$

which we rewrite as

$$u_i^{''} \approx \frac{u_{i+1} - 2u_i + u_{i-i}}{h^2}.$$

We can rewrite our original differential equation in terms of a discretized equation with approximations to the derivatives as

$$-\frac{u_{i+1} - 2u_i + u_{i-i}}{h^2} = f(x_i, u(x_i)),$$

with $i = 1,2,\dots,n$. We need to add to this system the two boundary conditions $u(a) = u_0$ and $u(b) = u_{n+1}$. If we define a matrix

$$\mathbf{A} = \frac{1}{h^2}\left(\begin{array}{cccccc} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & \dots & \dots & \dots & \dots & \dots \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{array}\right)$$

and the corresponding vectors $\mathbf{u} = (u_1, u_2, \dots, u_n)^T$ and $\mathbf{f}(\mathbf{u}) = f(x_1, x_2, \dots, x_n, u_1, u_2, \dots, u_n)^T$ we can rewrite the differential equation including the boundary conditions as a system of linear equations with a large number of unknowns

$$\mathbf{Au} = \mathbf{f}(\mathbf{u}). \tag{6.14}$$

We assume that the solution u exists and is unique for the exact differential equation, viz that the boundary value problem has a solution. But the discretization of the above differential equation leads to several questions, such as how well does the approximate solution resemble the exact one as $h \to 0$, or does a given small value of h allow us to establish existence and uniqueness of the solution.

Here we specialize to two particular cases. Assume first that the function f does not depend on $u(x)$. Then our linear equation reduces to

$$\mathbf{Au} = \mathbf{f}, \tag{6.15}$$

which is nothing but a simple linear equation with a tridiagonal matrix $\mathbf{A}$. We will solve such a system of equations in subsection 6.4.3.

If we assume that our boundary value problem is that of a quantum mechanical particle confined by a harmonic oscillator potential, then our function f takes the form (assuming that all constants $m = \hbar = \omega = 1$) $f(x_i, u(x_i)) = -x_i^2 u(x_i) + 2\lambda u(x_i)$ with λ being the eigenvalue. Inserting this into our equation, we define first a new matrix $\mathbf{A}$ as

$$\mathbf{A} = \begin{pmatrix} \frac{2}{h^2}+x_1^2 & -\frac{1}{h^2} & & & & \\ -\frac{1}{h^2} & \frac{2}{h^2}+x_2^2 & -\frac{1}{h^2} & & & \\ & -\frac{1}{h^2} & \frac{2}{h^2}+x_3^2 & -\frac{1}{h^2} & & \\ & \dots & \dots & \dots & \dots & \dots \\ & & & -\frac{1}{h^2} & \frac{2}{h^2}+x_{n-1}^2 & -\frac{1}{h^2} \\ & & & & -\frac{1}{h^2} & \frac{2}{h^2}+x_n^2 \end{pmatrix}, \tag{6.16}$$

which leads to the following eigenvalue problem

$$\begin{pmatrix} \frac{2}{h^2}+x_1^2 & -\frac{1}{h^2} & & & & \\ -\frac{1}{h^2} & \frac{2}{h^2}+x_2^2 & -\frac{1}{h^2} & & & \\ & -\frac{1}{h^2} & \frac{2}{h^2}+x_3^2 & -\frac{1}{h^2} & & \\ & \dots & \dots & \dots & \dots & \dots \\ & & & -\frac{1}{h^2} & \frac{2}{h^2}+x_{n-1}^2 & -\frac{1}{h^2} \\ & & & & -\frac{1}{h^2} & \frac{2}{h^2}+x_n^2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \\ \\ \\ u_n \end{pmatrix} = 2\lambda \begin{pmatrix} u_1 \\ u_2 \\ \\ \\ \\ u_n \end{pmatrix}.$$

We will solve this type of equations in chapter 7. These lecture notes contain however several other examples of rewriting mathematical expressions into matrix problems. In chapter 5 we show how a set of linear integral equation when discretized can be transformed into a simple matrix inversion problem. The specific example we study in that chapter is the rewriting of Schrödinger's equation for scattering problems. Other examples of linear equations will appear in our discussion of ordinary and partial differential equations.

6.4.1 Gaussian Elimination

Any discussion on the solution of linear equations should start with Gaussian elimination. This text is no exception. We start with the linear set of equations

$$\mathbf{Ax} = \mathbf{w}.$$

We assume also that the matrix $\mathbf{A}$ is non-singular and that the matrix elements along the diagonal satisfy $a_{ii} \neq 0$. We discuss later how to handle such cases. In the discussion we limit ourselves again to a matrix $\mathbf{A} \in \mathbb{R}^{4\times4}$, resulting in a set of linear equations of the form

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix}.$$

or

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 = & \; w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 = & \; w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 = & \; w_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 = & \; w_4. \end{aligned}$$

The basic idea of Gaussian elimination is to use the first equation to eliminate the first unknown x_1 from the remaining $n-1$ equations. Then we use the new second equation to eliminate the second unknown x_2 from the remaining $n-2$ equations. With $n-1$ such eliminations we obtain a so-called upper triangular set of equations of the form

$$\begin{aligned} b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 = & \; y_1 \\ b_{22}x_2 + b_{23}x_3 + b_{24}x_4 = & \; y_2 \\ b_{33}x_3 + b_{34}x_4 = & \; y_3 \\ b_{44}x_4 = & \; y_4. \end{aligned}$$

We can solve this system of equations recursively starting from x_n (in our case x_4) and proceed with what is called a backward substitution. This process can be expressed mathematically as

$$x_m = \frac{1}{b_{mm}}\left(y_m - \sum_{k=m+1}^{n} b_{mk}x_k\right) \qquad m = n-1, n-2, \dots, 1.$$

To arrive at such an upper triangular system of equations, we start by eliminating the unknown x_1 for $j = 2, n$. We achieve this by multiplying the first equation by a_{j1}/a_{11} and then subtract the result from the jth equation. We assume obviously that $a_{11} \neq 0$ and that $\mathbf{A}$ is not singular. We will come back to this problem below.

Our actual 4×4 example reads after the first operation

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & (a_{22} - \frac{a_{21}a_{12}}{a_{11}}) & (a_{23} - \frac{a_{21}a_{13}}{a_{11}}) & (a_{24} - \frac{a_{21}a_{14}}{a_{11}}) \\ 0 & (a_{32} - \frac{a_{31}a_{12}}{a_{11}}) & (a_{33} - \frac{a_{31}a_{13}}{a_{11}}) & (a_{34} - \frac{a_{31}a_{14}}{a_{11}}) \\ 0 & (a_{42} - \frac{a_{41}a_{12}}{a_{11}}) & (a_{43} - \frac{a_{41}a_{13}}{a_{11}}) & (a_{44} - \frac{a_{41}a_{14}}{a_{11}}) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ w_2^{(2)} \\ w_3^{(2)} \\ w_4^{(2)} \end{pmatrix}.$$

or

$$
\begin{aligned}
b_{11}x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= \quad y_1 \\
a_{22}^{(2)}x_2 + a_{23}^{(2)}x_3 + a_{24}^{(2)}x_4 &= \quad w_2^{(2)} \\
a_{32}^{(2)}x_2 + a_{33}^{(2)}x_3 + a_{34}^{(2)}x_4 &= \quad w_3^{(2)} \\
a_{42}^{(2)}x_2 + a_{43}^{(2)}x_3 + a_{44}^{(2)}x_4 &= \quad w_4^{(2)},
\end{aligned}
\tag{6.17}
$$

with the new coefficients

$$b_{1k} = a_{1k}^{(1)} \quad k = 1, \dots, n,$$

where each $a_{1k}^{(1)}$ is equal to the original a_{1k} element. The other coefficients are

$$a_{jk}^{(2)} = a_{jk}^{(1)} - \frac{a_{j1}^{(1)}a_{1k}^{(1)}}{a_{11}^{(1)}} \quad j,k = 2, \dots, n,$$

with a new right-hand side given by

$$y_1 = w_1^{(1)}, \; w_j^{(2)} = w_j^{(1)} - \frac{a_{j1}^{(1)}w_1^{(1)}}{a_{11}^{(1)}} \quad j = 2, \dots, n.$$

We have also set $w_1^{(1)} = w_1$, the original vector element. We see that the system of unknowns $x_1, \dots, x_n$ is transformed into an $(n-1)\times(n-1)$ problem.

This step is called forward substitution. Proceeding with these substitutions, we obtain the general expressions for the new coefficients

$$a_{jk}^{(m+1)} = a_{jk}^{(m)} - \frac{a_{jm}^{(m)}a_{mk}^{(m)}}{a_{mm}^{(m)}} \quad j,k = m+1, \dots, n,$$

with $m = 1, \dots, n-1$ and a right-hand side given by

$$w_j^{(m+1)} = w_j^{(m)} - \frac{a_{jm}^{(m)}w_m^{(m)}}{a_{mm}^{(m)}} \quad j = m+1, \dots, n.$$

This set of $n-1$ elimations leads us to Eq. (6.17), which is solved by back substitution. If the arithmetics is exact and the matrix $\mathbf{A}$ is not singular, then the computed answer will be exact. However, as discussed in the two preceeding chapters, computer arithmetics is not exact. We will always have to cope with truncations and possible losses of precision. Even though the matrix elements along the diagonal are not zero, numerically small numbers may appear and subsequent divisions may lead to large numbers, which, if added to a small number may yield losses of precision.

Suppose for example that our first division in $(a_{22} - a_{21}a_{12}/a_{11})$ results in -10^7, that is $a_{21}a_{12}/a_{11}$. Assume also that a_{22} is one. We are then adding $10^7 + 1$. With single precision this results in 10^7. Already at this stage we see the potential for producing wrong results.

The solution to this set of problems is called pivoting, and we distinguish between partial and full pivoting. Pivoting means that if small values (especially zeros) do appear on the diagonal we remove them by rearranging the matrix and vectors by permuting rows and columns. As a simple example, let us assume that at some stage during a calculation we have the following set of linear equations

$$\begin{pmatrix} 1 & 3 & 4 & 6 \\ 0 & 10^{-8} & 198 & 19 \\ 0 & -91 & 51 & 9 \\ 0 & 7 & 76 & 541 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}.$$

The element at row $i = 2$ and column 2 is 10^{-8} and may cause problems for us in the next forward substitution. The element $i = 2, j = 3$ is the largest in the second row and the element $i = 3, j = 2$ is the largest in the third row. The small element can be removed by rearranging the rows and/or columns to bring a larger value into the $i = 2, j = 2$ element.

In partial or column pivoting, we rearrange the rows of the matrix and the right-hand side to bring the numerically largest value in the column onto the diagonal. For our example matrix the largest value of column two is in element $i = 3, j = 2$ and we interchange rows 2 and 3 to give

$$\begin{pmatrix} 1 & 3 & 4 & 6 \\ 0 & -91 & 51 & 9 \\ 0 & 10^{-8} & 198 & 19 \\ 0 & 7 & 76 & 541 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_3 \\ y_2 \\ y_4 \end{pmatrix}.$$

Note that our unknown variables x_i remain in the same order which simplifies the implementation of this procedure. The right-hand side vector, however, has been rearranged. Partial pivoting may be implemented for every step of the solution process, or only when the diagonal values are sufficiently small as to potentially cause a problem. Pivoting for every step will lead to smaller errors being introduced through numerical inaccuracies, but the continual reordering will slow down the calculation.

The philosophy behind full pivoting is much the same as that behind partial pivoting. The main difference is that the numerically largest value in the column or row containing the value to be replaced. In our example above the magnitude of element $i = 2, j = 3$ is the greatest in row 2 or column 2. We could rearrange the columns in order to bring this element onto the diagonal. This will also entail a rearrangement of the solution vector x. The rearranged system becomes, interchanging columns two and three,

$$\begin{pmatrix} 1 & 6 & 3 & 4 \\ 0 & 198 & 10^{-8} & 19 \\ 0 & 51 & -91 & 9 \\ 0 & 76 & 7 & 541 \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_2 \\ x_4 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}.$$

The ultimate degree of accuracy can be provided by rearranging both rows and columns so that the numerically largest value in the submatrix not yet processed is brought onto the diagonal. This process may be undertaken for every step, or only when the value on the diagonal is considered too small relative to the other values in the matrix. In our case, the matrix element at $i = 4, j = 4$ is the largest. We could here interchange rows two and four and then columns two and four to bring this matrix element at the diagonal position $i = 2, j = 2$. When interchanging columns and rows, one needs to keep track of all permutations performed. Partial and full pivoting are discussed in most texts on numerical linear algebra. For an in-depth discussion we recommend again the text of Golub and Van Loan [9], in particular chapter three. See also the discussion of chapter two in Ref. [14]. The library functions you end up using, be it via Matlab, the library included with this text or other ones, do all include pivoting.

If it is not possible to rearrange the columns or rows to remove a zero from the diagonal, then the matrix A is singular and no solution exists.

Gaussian elimination requires however many floating point operations. An $n \times n$ matrix requires for the simultaneous solution of a set of r different right-hand sides, a total of $n^3/3 + rn^2 - n/3$ multiplications. Adding the cost of additions, we end up with $2n^3/3 + O(n^2)$ floating point operations, see Kress [11] for a proof. An $n \times n$ matrix of dimensionalty $n = 10^3$ requires, on a modern PC with a processor that allows for something like 10^9 floating point operations per second (flops), approximately one second. If you increase the size of the matrix to $n = 10^4$ you need 1000 seconds, or roughly 16 minutes.

Although the direct Gaussian elmination algorithm allows you to compute the determinant of $\mathbf{A}$ via the product of the diagonal matrix elements of the triangular matrix, it is seldomly used in normal applications. The more practical elimination is provided by what is called lower and upper decomposition. Once decomposed, one can use this matrix to solve many other linear systems which use the same matrix $\mathbf{A}$, viz with different right-hand sides. With an LU decomposed matrix, the number of floating point operations for solving a set of linear equations scales as $O(n^2)$. One should however note that to obtain the LU decompsed matrix requires roughly $O(n^3)$ floating point operations. Finally, LU decomposition allows for an efficient computation of the inverse of $\mathbf{A}$.

6.4.2 LU Decomposition of a Matrix

A frequently used form of Gaussian elimination is L(ower)U(pper) factorization also known as LU Decomposition or Crout or Dolittle factorisation. In this section we describe how one can decompose a matrix A in terms of a matrix L with elements only below the diagonal (and thereby the naming lower) and a matrix U which contains both the diagonal and matrix elements above the diagonal (leading to the labelling upper). Consider again the matrix $\mathbf{A}$ given in Eq. (6.1). The LU decomposition method means that we can rewrite this matrix as the product of two matrices $\mathbf{L}$ and $\mathbf{U}$ where

$$\mathbf{A} = \mathbf{LU} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix}. \tag{6.18}$$

LU decomposition forms the backbone of other algorithms in linear algebra, such as the solution of linear equations given by

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= w_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= w_4. \end{aligned}$$

The above set of equations is conveniently solved by using LU decomposition as an intermediate step, see the next subsection for more details on how to solve linear equations with an LU decomposed matrix.

The matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ has an LU factorization if the determinant is different from zero. If the LU factorization exists and $\mathbf{A}$ is non-singular, then the LU factorization is unique and the determinant is given by

$$det\{\mathbf{A}\} = u_{11}u_{22}\dots u_{nn}.$$

For a proof of this statement, see chapter 3.2 of Ref. [9].

The algorithm for obtaining L and U is actually quite simple. We start always with the first column. In our simple (4×4) case we obtain then the following equations for the first column

$$\begin{aligned} a_{11} &= u_{11} \\ a_{21} &= l_{21}u_{11} \\ a_{31} &= l_{31}u_{11} \\ a_{41} &= l_{41}u_{11}, \end{aligned}$$

which determine the elements u_{11}, l_{21}, l_{31} and l_{41} in **L** and **U**. Writing out the equations for the second column we get

$$\begin{aligned} a_{12} &= u_{12} \\ a_{22} &= l_{21}u_{12}+u_{22} \\ a_{32} &= l_{31}u_{12}+l_{32}u_{22} \\ a_{42} &= l_{41}u_{12}+l_{42}u_{22}. \end{aligned}$$

Here the unknowns are u_{12}, u_{22}, l_{32} and l_{42} which can all be evaluated by means of the results from the first column and the elements of **A**. Note an important feature. When going from the first to the second column we do not need any further information from the matrix elements a_{i1}. This is a general property throughout the whole algorithm. Thus the memory locations for the matrix **A** can be used to store the calculated matrix elements of **L** and **U**. This saves memory.

We can generalize this procedure into three equations

$$\begin{aligned} i<j: \quad l_{i1}u_{1j}+l_{i2}u_{2j}+\cdots+l_{ii}u_{ij} &= a_{ij} \\ i=j: \quad l_{i1}u_{1j}+l_{i2}u_{2j}+\cdots+l_{ii}u_{jj} &= a_{ij} \\ i>j: \quad l_{i1}u_{1j}+l_{i2}u_{2j}+\cdots+l_{ij}u_{jj} &= a_{ij} \end{aligned}$$

which gives the following algorithm:
Calculate the elements in **L** and **U** columnwise starting with column one. For each column (j):

- Compute the first element u_{1j} by

$$u_{1j}=a_{1j}.$$

- Next, we calculate all elements $u_{ij}, i=2,\dots,j-1$

$$u_{ij}=a_{ij}-\sum_{k=1}^{i-1} l_{ik}u_{kj}.$$

- Then calculate the diagonal element u_{jj}

$$u_{jj}=a_{jj}-\sum_{k=1}^{j-1} l_{jk}u_{kj}. \tag{6.19}$$

- Finally, calculate the elements $l_{ij}, i>j$

$$l_{ij}=\frac{1}{u_{jj}}\left(a_{ij}-\sum_{k=1}^{i-1} l_{ik}u_{kj}\right), \tag{6.20}$$

The algorithm is known as Doolittle's algorithm since the diagonal matrix elements of $\mathbf{L}$ are 1. For the case where the diagonal elements of $\mathbf{U}$ are 1, we have what is called Crout's algorithm. For the case where $\mathbf{U} = \mathbf{L}^T$ so that $u_{ii} = l_{ii}$ for $1 \leq i \leq n$ we can use what is called the Cholesky factorization algorithm. In this case the matrix $\mathbf{A}$ has to fulfill several features; namely, it should be real, symmetric and positive definite. A matrix is positive definite if the quadratic form $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$. Establishing this feature is not easy since it implies the use of an arbitrary vector $\mathbf{x} \neq 0$. If the matrix is positive definite and symmetric, its eigenvalues are always real and positive. We discuss the Cholesky factorization below.

A crucial point in the LU decomposition is obviously the case where u_{jj} is close to or equals zero, a case which can lead to serious problems. Consider the following simple 2×2 example taken from Ref. [17]

$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

The algorithm discussed above fails immediately, the first step simple states that $u_{11} = 0$. We could change slightly the above matrix by replacing 0 with 10^{-20} resulting in

$$\mathbf{A} = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix},$$

yielding

$$\begin{aligned} u_{11} &= 10^{-20} \\ l_{21} &= 10^{20} \end{aligned}$$

and $u_{12} = 1$ and

$$u_{22} = a_{11} - l_{21} = 1 - 10^{20},$$

we obtain

$$\mathbf{L} = \begin{pmatrix} 1 & 0 \\ 10^{20} & 1 \end{pmatrix},$$

and

$$\mathbf{U} = \begin{pmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{pmatrix},$$

With the change from 0 to a small number like 10^{-20} we see that the LU decomposition is now stable, but it is not backward stable. What do we mean by that? First we note that the matrix $\mathbf{U}$ has an element $u_{22} = 1 - 10^{20}$. Numerically, since we do have a limited precision, which for double precision is approximately $\varepsilon_M \sim 10^{-16}$ it means that this number is approximated in the machine as $u_{22} \sim -10^{20}$ resulting in a machine representation of the matrix as

$$\mathbf{U} = \begin{pmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{pmatrix}.$$

If we multiply the matrices **LU** we have

$$\left(\begin{array}{cc} 1 & 0 \\ 10^{20} & 1 \end{array}\right)\left(\begin{array}{cc} 10^{-20} & 1 \\ 0 & -10^{20} \end{array}\right) = \left(\begin{array}{cc} 10^{-20} & 1 \\ 1 & 0 \end{array}\right) \neq \mathbf{A}.$$

We do not get back the original matrix **A**!

The solution is pivoting (interchanging rows in this case) around the largest element in a column j. Then we are actually decomposing a rowwise permutation of the original matrix **A**. The key point to notice is that Eqs. (6.19) and (6.20) are equal except for the case that we divide by u_{jj} in the latter one. The upper limits are always the same $k = j-1 (= i-1)$. This means that we do not have to choose the diagonal element u_{jj} as the one which happens to fall along the diagonal in the first instance. Rather, we could promote one of the undivided l_{ij}'s in the column $i = j+1, \dots N$ to become the diagonal of U. The partial pivoting in Crout's or Doolittle's methods means then that we choose the largest value for u_{jj} (the pivot element) and then do the divisions by that element. Then we need to keep track of all permutations performed. For the above matrix **A** it would have sufficed to interchange the two rows and start the LU decomposition with

$$\mathbf{A} = \left(\begin{array}{cc} 1 & 1 \\ 0 & 1 \end{array}\right).$$

The error which is done in the LU decomposition of an $n \times n$ matrix if no zero pivots are encountered is given by, see chapter 3.3 of Ref. [9],

$$\mathbf{LU} = \mathbf{A} + \mathbf{H},$$

with

$$|\mathbf{H}| \leq 3(n-1)\mathbf{u}\left(|\mathbf{A}| + |\mathbf{L}||\mathbf{U}|\right) + O(\mathbf{u}^2),$$

with $|\mathbf{H}|$ being the absolute value of a matrix and **u** is the error done in representing the matrix elements of the matrix **A** as floating points in a machine with a given precision ε_M, viz. every matrix element of **u** is

$$|fl(a_{ij}) - a_{ij}| \leq u_{ij},$$

with $|u_{ij}| \leq \varepsilon_M$ resulting in

$$|fl(\mathbf{A}) - \mathbf{A}| \leq \mathbf{u}|\mathbf{A}|.$$

The programs which perform the above described LU decomposition are called as follows

```
C++:        ludcmp(double **a, int n, int *indx, double *d)
Fortran:    CALL lu_decompose(a, n, indx, d)
```

Both the C++ and Fortran 90/95 programs receive as input the matrix to be LU decomposed. In C++ this is given by the double pointer `**a`. Further, both functions need the size of the matrix n. It returns the variable d, which is ± 1 depending on whether we have an even or odd number of row interchanges, a pointer $indx$ that records the row permutation which has been effected and the LU decomposed matrix. Note that the original matrix is destroyed.

Cholesky's Factorization

If the matrix A is real, symmetric and positive definite, then it has a unique factorization (called Cholesky factorization)

$$A = LU = LL^T$$

where L^T is the upper matrix, implying that

$$L^T_{ij} = L_{ji}.$$

The algorithm for the Cholesky decomposition is a special case of the general LU-decomposition algorithm. The algorithm of this decomposition is as follows

- Calculate the diagonal element L_{ii} by setting up a loop for $i = 0$ to $i = n-1$ (C++ indexing of matrices and vectors)

$$L_{ii} = \left(A_{ii} - \sum_{k=0}^{i-1} L_{ik}^2 \right)^{1/2}.$$

- within the loop over i, introduce a new loop which goes from $j = i+1$ to $n-1$ and calculate

$$L_{ji} = \frac{1}{L_{ii}} \left(A_{ij} - \sum_{k=0}^{i-1} L_{ik} l_{jk} \right).$$

For the Cholesky algorithm we have always that $L_{ii} > 0$ and the problem with exceedingly large matrix elements does not appear and hence there is no need for pivoting.

To decide whether a matrix is positive definite or not needs some careful analysis. To find criteria for positive definiteness, one needs two statements from matrix theory, see Golub and Van Loan [9] for examples. First, the leading principal submatrices of a positive definite matrix are positive definite and non-singular and secondly a matrix is positive definite if and only if it has an $\mathbf{LDL}^T$ factorization with positive diagonal elements only in the diagonal matrix $\mathbf{D}$. A positive definite matrix has to be symmetric and have only positive eigenvalues.

The easiest way therefore to test whether a matrix is positive definite or not is to solve the eigenvalue problem $\mathbf{Ax} = \lambda \mathbf{x}$ and check that all eigenvalues are positive.

6.4.3 Solution of Linear Systems of Equations

With the LU decomposition it is rather simple to solve a system of linear equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 &= w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= w_3 \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= w_4. \end{aligned}$$

This can be written in matrix form as

$$\mathbf{Ax} = \mathbf{w}.$$

where $\mathbf{A}$ and $\mathbf{w}$ are known and we have to solve for $\mathbf{x}$. Using the LU dcomposition we write

$$\mathbf{Ax} \equiv \mathbf{LUx} = \mathbf{w}. \tag{6.21}$$

This equation can be calculated in two steps

$$\mathbf{Ly} = \mathbf{w}; \qquad \mathbf{Ux} = \mathbf{y}. \tag{6.22}$$

To show that this is correct we use to the LU decomposition to rewrite our system of linear equations as

$$\mathbf{LUx} = \mathbf{w},$$

and since the determinat of $\mathbf{L}$ is equal to 1 (by construction since the diagonals of $\mathbf{L}$ equal 1) we can use the inverse of $\mathbf{L}$ to obtain

$$\mathbf{Ux} = \mathbf{L^{-1}w} = \mathbf{y},$$

which yields the intermediate step

$$\mathbf{L^{-1}w} = \mathbf{y}$$

and multiplying with $\mathbf{L}$ on both sides we reobtain Eq. (6.22). As soon as we have $\mathbf{y}$ we can obtain $\mathbf{x}$ through $\mathbf{Ux} = \mathbf{y}$.

For our four-dimentional example this takes the form

$$\begin{aligned} y_1 &= w_1 \\ l_{21}y_1 + y_2 &= w_2 \\ l_{31}y_1 + l_{32}y_2 + y_3 &= w_3 \\ l_{41}y_1 + l_{42}y_2 + l_{43}y_3 + y_4 &= w_4. \end{aligned}$$

and

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + u_{14}x_4 = & \ y_1 \\ u_{22}x_2 + u_{23}x_3 + u_{24}x_4 = & \ y_2 \\ u_{33}x_3 + u_{34}x_4 = & \ y_3 \\ u_{44}x_4 = & \ y_4 \end{aligned}$$

This example shows the basis for the algorithm needed to solve the set of n linear equations. The algorithm goes as follows

- Set up the matrix **A** and the vector **w** with their correct dimensions. This determines the dimensionality of the unknown vector **x**.
- Then LU decompose the matrix **A** through a call to the function

C++:	ludcmp(double a, int n, int indx, double &d)
Fortran:	CALL lu_decompose(a, n, indx, d)

  This functions returns the LU decomposed matrix **A**, its determinant and the vector indx which keeps track of the number of interchanges of rows. If the determinant is zero, the solution is malconditioned.
- Thereafter you call the function

C++:	lubksb(double a, int n, int indx, double w)
Fortran:	CALL lu_linear_equation(a, n, indx, w)

  which uses the LU decomposed matrix **A** and the vector **w** and returns **x** in the same place as **w**. Upon exit the original content in **w** is destroyed. If you wish to keep this information, you should make a backup of it in your calling function.

6.4.4 Inverse of a Matrix and the Determinant

The basic definition of the determinant of **A** is

$$det\{\mathbf{A}\} = \sum_p (-1)^p a_{1p_1} \cdot a_{2p_2} \cdots a_{np_n},$$

where the sum runs over all permutations p of the indices $1, 2, \ldots, n$, altogether $n!$ terms. To calculate the inverse of **A** is a formidable task. Here we have to calculate *the complementary cofactor* a^{ij} of each element a_{ij} which is the $(n-1)$determinant obtained by striking out the row i and column j in which the element a_{ij} appears.

The inverse of **A** is then constructed as the transpose of a matrix with the elements $(-)^{i+j}a^{ij}$. This involves a calculation of n^2 determinants using the formula above. A simplified method is highly needed.

With the LU decomposed matrix **A** in Eq. (6.18) it is rather easy to find the determinant

$$det\{\mathbf{A}\} = det\{\mathbf{L}\} \times det\{\mathbf{U}\} = det\{\mathbf{U}\},$$

since the diagonal elements of **L** equal 1. Thus the determinant can be written

$$det\{\mathbf{A}\} = \prod_{k=1}^{N} u_{kk}.$$

The inverse is slightly more difficult. However, with an LU decomposed matrix this reduces to solving a set of linear equations. To see this, we recall that if the inverse exists then

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I},$$

the identity matrix. With an LU decomposed matrix we can rewrite the last equation as

$$\mathbf{LUA}^{-1} = \mathbf{I}.$$

If we assume that the first column (that is column 1) of the inverse matrix can be written as a vector with unknown entries

$$\mathbf{A}_1^{-1} = \begin{pmatrix} a_{11}^{-1} \\ a_{21}^{-1} \\ \dots \\ a_{n1}^{-1} \end{pmatrix},$$

then we have a linear set of equations

$$\mathbf{LU}\begin{pmatrix} a_{11}^{-1} \\ a_{21}^{-1} \\ \dots \\ a_{n1}^{-1} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \dots \\ 0 \end{pmatrix}.$$

In a similar way we can compute the unknow entries of the second column,

$$\mathbf{LU}\begin{pmatrix} a_{12}^{-1} \\ a_{22}^{-1} \\ \dots \\ a_{n2}^{-1} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ \dots \\ 0 \end{pmatrix},$$

and continue till we have solved all n sets of linear equations.

A calculation of the inverse of a matrix could then be implemented in the following way:

- Set up the matrix to be inverted.
- Call the LU decomposition function.
- Check whether the determinant is zero or not.
- Then solve column by column the sets of linear equations.

The following codes compute the inverse of a matrix using either C++ or Fortran as programming languages. They are both included in the library packages, but we include them explicitely here as well as two distinct programs which use these functions. We list first the C++ code.

http://folk.uio.no/compphys/programs/chapter06/cpp/program1.cpp

```
/* The function
**           inverse()
** perform a mtx inversion of the input matrix a[][] with
** dimension n.
*/
void inverse(double **a, int n)
{
 int       i,j, *indx;
 double    d, *col, **y;

 // allocate space in memory
 indx = new int[n];
 col = new double[n];
 y   = (double **) matrix(n, n, sizeof(double));
 // first we need to LU decompose the matrix
 ludcmp(a, n, indx, &d);
 // find inverse of a[][] by columns
 for(j = 0; j < n; j++) {
  // initialize right-side of linear equations
  for(i = 0; i < n; i++) col[i] = 0.0;
  col[j] = 1.0;
  lubksb(a, n, indx, col);
  // save result in y[][]
  for(i = 0; i < n; i++) y[i][j] = col[i];
 }
 // return the inverse matrix in a[][]

 for(i = 0; i < n; i++) {
  for(j = 0; j < n; j++) a[i][j] = y[i][j];
 }
 free_matrix((void **) y); // release local memory
 delete [] col;
 delete []indx;

} // End: function inverse()
```

We first need to LU decompose the matrix. Thereafter we solve linear equations by using the back substitution method calling the function **lubksb** and obtain finally the inverse matrix.

An example of a C++ function which calls this function is also given in the following program and reads

http://folk.uio.no/compphys/programs/chapter06/cpp/program1.cpp

```
// Simple matrix inversion example
#include <iostream>
#include <new>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
#include "lib.h"

using namespace std;

/* function declarations */

void inverse(double **, int);
/*
** This program sets up a simple 3x3 symmetric matrix
** and finds its determinant and inverse
*/

int main()
{
  int       i, j, k, result, n = 3;
  double    **matr, sum,
    a[3][3] = { {1.0, 3.0, 4.0},
      {3.0, 4.0, 6.0},
      {4.0, 6.0, 8.0}};
  // memory for inverse matrix
  matr = (double **) matrix(n, n, sizeof(double));
  // various print statements in the original code are omitted

  inverse(matr, n); // calculate and return inverse matrix
  ....
  return 0;
} // End: function main()
```

In order to use the program library you need to include the **lib.h** file using the `#include "lib.h"` statement. This function utilizes the library function **matrix** and **free_matrix** to allocate and free memory during execution. The matrix $a[3][3]$ is set at compilation time. Alternatively, you could have used either Blitz++ or Armadillo.

The corresponding Fortran program for the inverse of a matrix reads http://folk.uio.no/compphys/programs/FYS3150/f90library/f90lib.f90

```
!
!         Routines to do mtx inversion, from Numerical
!         Recipes, Teukolsky et al. Routines included
!         below are MATINV, LUDCMP and LUBKSB. See chap 2
!         of Numerical Recipes for further details
!
SUBROUTINE matinv(a,n, indx, d)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  INTEGER :: i, j
  REAL(DP), DIMENSION(n,n), INTENT(INOUT) :: a
  REAL(DP), ALLOCATABLE :: y(:,:)
  REAL(DP) :: d
  INTEGER, , INTENT(INOUT) :: indx(n)

  ALLOCATE (y( n, n))
  y=0.
  !    setup identity matrix
  DO i=1,n
    y(i,i)=1.
  ENDDO
  !    LU decompose the matrix just once
  CALL lu_decompose(a,n,indx,d)

  !    Find inverse by columns
  DO j=1,n
    CALL lu_linear_equation(a,n,indx,y(:,j))
  ENDDO
  !    The original matrix a was destroyed, now we equate it with the inverse y
  a=y
  DEALLOCATE ( y )

END SUBROUTINE matinv
```

The Fortran program **matinv** receives as input the same variables as the C++ program and calls the function for LU decomposition **lu_decompose** and the function to solve sets of linear equations **lu_linear_equation**. The program listed under programs/chapter4/program1.f90 performs the same action as the C++ listed above. In order to compile and link these programs it is convenient to use a so-called **makefile**. Examples of these are found under the same catalogue as the above programs.

Scattering Equation and Principal Value Integrals via Matrix Inversion

In quantum mechanics, it is often common to rewrite Schrödinger's equation in momentum space, after having made a so-called partial wave expansion of the interaction. We will not go into the details of these expressions but limit ourselves to study the equivalent problem for so-called scattering states, meaning that the total energy of two particles which collide is larger than or equal zero. The benefit of rewriting the equation in momentum space, after having performed a Fourier transformation, is that the coordinate space equation, being an integro-differantial equation, is transformed into an integral equation. The latter can be solved by standard matrix inversion techniques. Furthermore, the results of solving these equation can be related directly to experimental observables like the scattering phase shifts. The latter tell us how much the incoming two-particle wave function is modified by a collision. Here we take a more technical stand and consider the technical aspects of solving an integral equation with a principal value.

For scattering states, $E > 0$, the corresponding equation to solve is the so-called Lippman-Schwinger equation. This is an integral equation where we have to deal with the amplitude $R(k,k')$ (reaction matrix) defined through the integral equation

$$R_l(k,k') = V_l(k,k') + \frac{2}{\pi}\mathscr{P}\int_0^\infty dqq^2V_l(k,q)\frac{1}{E-q^2/m}R_l(q,k'), \tag{6.23}$$

where the total kinetic energy of the two incoming particles in the center-of-mass system is

$$E = \frac{k_0^2}{m}. \tag{6.24}$$

The symbol $\mathscr{P}$ indicates that Cauchy's principal-value prescription is used in order to avoid the singularity arising from the zero of the denominator. We will discuss below how to solve this problem. Equation (6.23) represents then the problem you will have to solve numerically. The interaction between the two particles is given by a partial-wave decomposed version $V_l(k,k')$, where l stands for a quantum number like the orbital momentum. We have assumed that interaction does not coupled to partial waves with different orbital momenta. The variables k and k' are the outgoing and incoming relative momenta of the two interacting particles.

The matrix $R_l(k,k')$ relates to the experimental the phase shifts δ_l through its diagonal elements as

$$R_l(k_0,k_0) = -\frac{tan\delta_l}{mk_0}, \tag{6.25}$$

where m is the reduced mass of the interacting particles. Furthemore, the interaction between the particles, V, carries

In order to solve the Lippman-Schwinger equation in momentum space, we need first to write a function which sets up the integration points. We need to do that since we are going to approximate the integral through

$$\int_a^b f(x)dx \approx \sum_{i=1}^N w_i f(x_i),$$

where we have fixed N integration points through the corresponding weights w_i and points x_i. These points can for example be determined using Gaussian quadrature.

The principal value in Eq. (6.23) is rather tricky to evaluate numerically, mainly since computers have limited precision. We will here use a subtraction trick often used when dealing with singular integrals in numerical calculations. We use the calculus relation from the previous section

$$\int_{-\infty}^{\infty}\frac{dk}{k-k_0}=0,$$

or

$$\int_{0}^{\infty}\frac{dk}{k^2-k_0^2}=0.$$

We can use this to express a principal values integral as

$$\mathscr{P}\int_0^{\infty}\frac{f(k)dk}{k^2-k_0^2}=\int_0^{\infty}\frac{(f(k)-f(k_0))dk}{k^2-k_0^2}, \tag{6.26}$$

where the right-hand side is no longer singular at $k=k_0$, it is proportional to the derivative df/dk, and can be evaluated numerically as any other integral.

We can then use the trick in Eq. (6.26) to rewrite Eq. (6.23) as

$$R(k,k')=V(k,k')+\frac{2}{\pi}\int_0^{\infty}dq\frac{q^2V(k,q)R(q,k')-k_0^2V(k,k_0)R(k_0,k')}{(k_0^2-q^2)/m}. \tag{6.27}$$

We are interested in obtaining $R(k_0,k_0)$, since this is the quantity we want to relate to experimental data like the phase shifts.

How do we proceed in order to solve Eq. (6.27)?

1. Using the mesh points k_j and the weights ω_j, we can rewrite Eq. (6.27) as

$$R(k,k')=V(k,k')+\frac{2}{\pi}\sum_{j=1}^{N}\frac{\omega_jk_j^2V(k,k_j)R(k_j,k')}{(k_0^2-k_j^2)/m}-\frac{2}{\pi}k_0^2V(k,k_0)R(k_0,k')\sum_{n=1}^{N}\frac{\omega_n}{(k_0^2-k_n^2)/m}. \tag{6.28}$$

This equation contains now the unknowns $R(k_i,k_j)$ (with dimension $N\times N$) and $R(k_0,k_0)$.

2. We can turn Eq. (6.28) into an equation with dimension $(N+1)\times(N+1)$ with an integration domain which contains the original mesh points k_j for $j=1,N$ and the point which corresponds to the energy k_0. Consider the latter as the 'observable' point. The mesh points become then k_j for $j=1,n$ and $k_{N+1}=k_0$.

3. With these new mesh points we define the matrix

$$A_{i,j}=\delta_{i,j}-V(k_i,k_j)u_j, \tag{6.29}$$

where δ is the Kronecker δ and

$$u_j = \frac{2}{\pi}\frac{\omega_j k_j^2}{(k_0^2 - k_j^2)/m} \qquad j = 1, N \tag{6.30}$$

and

$$u_{N+1} = -\frac{2}{\pi}\sum_{j=1}^{N}\frac{k_0^2\omega_j}{(k_0^2 - k_j^2)/m}. \tag{6.31}$$

The first task is then to set up the matrix A for a given k_0. This is an $(N+1)\times(N+1)$ matrix. It can be convenient to have an outer loop which runs over the chosen observable values for the energy k_0^2/m. *Note that all mesh points k_j for $j = 1, N$ must be different from k_0. Note also that $V(k_i, k_j)$ is an $(N+1)\times(N+1)$ matrix.*

4. With the matrix A we can rewrite Eq. (6.28) as a matrix problem of dimension $(N+1)\times(N+1)$. All matrices R, A and V have this dimension and we get

$$A_{i,l}R_{l,j} = V_{i,j}, \tag{6.32}$$

or just

$$AR = V. \tag{6.33}$$

5. Since we already have defined A and V (these are stored as $(N+1)\times(N+1)$ matrices) Eq. (6.33) involves only the unknown R. We obtain it by matrix inversion, i.e.,

$$R = A^{-1}V. \tag{6.34}$$

Thus, to obtain R, we need to set up the matrices A and V and invert the matrix A. With the inverse A^{-1} we perform a matrix multiplication with V and obtain R.

With R we can in turn evaluate the phase shifts by noting that

$$R(k_{N+1}, k_{N+1}) = R(k_0, k_0), \tag{6.35}$$

and we are done.

Inverse of the Vandermonde Matrix

In chapter 3 we discussed how to interpolate a function f which is known only at $n+1$ points $x_0, x_1, x_2, \dots, x_n$ with corresponding values $f(x_0), f(x_1), f(x_2), \dots, f(x_n)$. The latter is often a typical outcome of a large scale computation or from an experiment. In most cases in the sciences we do not have a closed-form expression for a function f. The function is only known at specific points.

We seek a functional form for a function f which passes through the above pairs of values

$$(x_0, f(x_0)), (x_1, f(x_1)), (x_2, f(x_2)), \dots, (x_n, f(x_n)).$$

This is normally achieved by expanding the function $f(x)$ in terms of well-known polynomials $\phi_i(x)$, such as Legendre, Chebyshev, Laguerre etc. The function is then approximated by a polynomial of degree n $p_n(x)$

$$f(x) \approx p_n(x) = \sum_{i=0}^{n} a_i \phi_i(x),$$

where a_i are unknown coefficients and $\phi_i(x)$ are a priori well-known functions. The simplest possible case is to assume that $\phi_i(x) = x^i$, resulting in an approximation

$$f(x) \approx a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

Our function is known at the points $n+1$ points $x_0, x_1, x_2, \dots, x_n$, leading to $n+1$ equations of the type

$$f(x_i) \approx a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n.$$

We can then obtain the unknown coefficients by rewriting our problem as

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & \dots & x_2^n \\ 1 & x_3 & x_3^2 & \dots & \dots & x_3^n \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_n & x_n^2 & \dots & \dots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \dots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ f(x_3) \\ \dots \\ f(x_n) \end{pmatrix},$$

an expression which can be rewritten in a more compact form as

$$\mathbf{X}\mathbf{a} = \mathbf{f},$$

with

$$\mathbf{X} = \begin{pmatrix} 1 & x_0 & x_0^2 & \dots & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & \dots & x_2^n \\ 1 & x_3 & x_3^2 & \dots & \dots & x_3^n \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_n & x_n^2 & \dots & \dots & x_n^n \end{pmatrix}.$$

This matrix is called a Vandermonde matrix and is by definition non-singular since all points x_i are different. The inverse exists and we can obtain the unknown coefficients by inverting $\mathbf{X}$, resulting in

$$\mathbf{a} = \mathbf{X}^{-1}\mathbf{f}.$$

Although this algorithm for obtaining an interpolating polynomial which approximates our data set looks very simple, it is an inefficient algorithm since the computation of the inverse requires $O(n^3)$ flops. The methods we discussed in chapter 3, together with spline interpolation discussed in the next section, are much more effective from a numerical point of view. There is also another subtle point. Although we have a data set with $n+1$ points, this does not necessarily mean that our function $f(x)$ is well represented by a polynomial of degree n. On the contrary, our function $f(x)$ may be a parabola (second-order in n), meaning that we have a large excess of data points. In such cases a least-square fit or a spline interpolation may be better approaches to represent the function. Spline interpolation will be discussed in the next section.

6.4.5 Tridiagonal Systems of Linear Equations

We start with the linear set of equations from Eq. (6.15), viz

$$\mathbf{A}\mathbf{u} = \mathbf{f},$$

where $\mathbf{A}$ is a tridiagonal matrix which we rewrite as

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{pmatrix}$$

where a,b,c are one-dimensional arrays of length $1:n$. In the example of Eq. (6.15) the arrays a and c are equal, namely $a_i = c_i = -1/h^2$. We can rewrite Eq. (6.15) as

$$\mathbf{A}\mathbf{u} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \dots \\ \dots \\ \dots \\ f_n \end{pmatrix}.$$

A tridiagonal matrix is a special form of banded matrix where all the elements are zero except for those on and immediately above and below the leading diagonal. The above tridiagonal system can be written as

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = f_i,$$

for $i = 1, 2, \ldots, n$. We see that u_{-1} and u_{n+1} are not required and we can set $a_1 = c_n = 0$. In many applications the matrix is symmetric and we have $a_i = c_i$. The algorithm for solving this set of equations is rather simple and requires two steps only, a forward substitution and a backward substitution. These steps are also common to the algorithms based on Gaussian elimination that we discussed previously. However, due to its simplicity, the number of floating point operations is in this case proportional with $O(n)$ while Gaussian elimination requires $2n^3/3 + O(n^2)$ floating point operations. In case your system of equations leads to a tridiagonal matrix, it is clearly an overkill to employ Gaussian elimination or the standard LU decomposition. You will encounter several applications involving tridiagonal matrices in our discussion of partial differential equations in chapter **??**.

Our algorithm starts with forward substitution with a loop over of the elements i and can be expressed via the following piece of code taken from the Numerical Recipe text of Teukolsky *et al* [14]

```
btemp = b[1];
u[1] = f[1]/btemp;
for(i=2 ; i <= n ; i++) {
  temp[i] = c[i-1]/btemp;
  btemp = b[i]-a[i]*temp[i];
  u[i] = (f[i] - a[i]*u[i-1])/btemp;
}
```

Note that you should avoid cases with $b_1 = 0$. If that is the case, you should rewrite the equations as a set of order $n-1$ with u_2 eliminated. Finally we perform the backsubstitution leading to the following code

```
for(i=n-1 ; i >= 1 ; i--) {
  u[i] -= temp[i+1]*u[i+1];
}
```

Note that our sums start with $i = 1$ and that one should avoid cases with $b_1 = 0$. If that is the case, you should rewrite the equations as a set of order $n-1$ with u_2 eliminated. However, a tridiagonal matrix problem is not a guarantee that we can find a solution. The matrix $\mathbf{A}$ which rephrases a second derivative in a discretized form

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix},$$

fulfills the condition of a weak dominance of the diagonal, with $|b_1| > |c_1|$, $|b_n| > |a_n|$ and $|b_k| \geq |a_k| + |c_k|$ for $k = 2, 3, \ldots, n-1$. This is a relevant but not sufficient condition to guarantee that the matrix **A** yields a solution to a linear equation problem. The matrix needs also to be irreducible. A tridiagonal irreducible matrix means that all the elements a_i and c_i are non-zero. If these two conditions are present, then **A** is nonsingular and has a unique LU decomposition.

We can obviously extend our boundary value problem to include a first derivative as well

$$-\frac{d^2u(x)}{dx^2}+g(x)\frac{du(x)}{dx}+h(x)u(x)=f(x),$$

with $x\in[a,b]$ and with boundary conditions $u(a)=u(b)=0$. We assume that f, g and h are continuous functions in the domain $x\in[a,b]$ and that $h(x)\geq 0$. Then the differential equation has a unique solution. We subdivide our interval $x\in[a,b]$ into n subintervals by setting $x_i=a+ih$, with $i=0,1,\dots,n+1$. The step size is then given by $h=(b-a)/(n+1)$ with $n\in\mathbb{N}$. For the internal grid points $i=1,2,\dots n$ we replace the differential operators with

$$u_i^{''}\approx\frac{u_{i+1}-2u_i+u_{i-i}}{h^2}.$$

for the second derivative while the first derivative is given by

$$u_i^{'}\approx\frac{u_{i+1}-u_{i-i}}{2h}.$$

We rewrite our original differential equation in terms of a discretized equation as

$$-\frac{u_{i+1}-2u_i+u_{i-i}}{h^2}+g_i\frac{u_{i+1}-u_{i-i}}{2h}+h_iu_i=f_i,$$

with $i=1,2,\dots,n$. We need to add to this system the two boundary conditions $u(a)=u_0$ and $u(b)=u_{n+1}$. This equation can again be rewritten as a tridiagonal matrix problem. We leave it as an exercise to the reader to find the matrix elements, find the conditions for having weakly dominant diagonal elements and that the matrix is irreducible.

6.5 Spline Interpolation

Cubic spline interpolation is among one of the most used methods for interpolating between data points where the arguments are organized as ascending series. In the library program we supply such a function, based on the so-called cubic spline method to be described below. The linear equation solver we developed in the previous section for tridiagonal matrices can be reused for spline interpolation.

A spline function consists of polynomial pieces defined on subintervals. The different subintervals are connected via various continuity relations.

Assume we have at our disposal $n+1$ points $x_0, x_1, \dots x_n$ arranged so that $x_0 < x_1 < x_2 < \dots x_{n-1} < x_n$ (such points are called knots). A spline function s of degree k with $n+1$ knots is defined as follows

- On every subinterval $[x_{i-1}, x_i)$ s is a polynomial of degree $\leq k$.
- s has $k-1$ continuous derivatives in the whole interval $[x_0, x_n]$.

As an example, consider a spline function of degree $k=1$ defined as follows

$$s(x) = \begin{cases} s_0(x) = a_0x + b_0 & x \in [x_0, x_1) \\ s_1(x) = a_1x + b_1 & x \in [x_1, x_2) \\ \dots & \dots \\ s_{n-1}(x) = a_{n-1}x + b_{n-1} & x \in [x_{n-1}, x_n] \end{cases} \tag{6.36}$$

In this case the polynomial consists of series of straight lines connected to each other at every endpoint. The number of continuous derivatives is then $k-1=0$, as expected when we deal with straight lines. Such a polynomial is quite easy to construct given $n+1$ points $x_0, x_1, \dots x_n$ and their corresponding function values.

The most commonly used spline function is the one with $k=3$, the so-called cubic spline function. Assume that we have in addition to the $n+1$ knots a series of functions values $y_0 = f(x_0), y_1 = f(x_1), \dots y_n = f(x_n)$. By definition, the polynomials s_{i-1} and s_i are thence supposed to interpolate the same point i, i.e., s$_{i-1}(x_i) = y_i = s_i(x_i), with 1$ $\leq i \leq n-1$. In total we have n polynomials of the type s$_i(x) = a_{i0} + a_{i1}x + a_{i2}x^2 + a_{i3}x^3, yielding 4$n$coefficientstodetermine. Everysubintervalprovidesinadditiontwoconditionsy_i = s(x_i), andy_{i+1} = s(x_{i+1}), tobefulfilled. Ifwealsoassumethats'ands''arecontinuous, thens'_{i-1}(x_i) = s'_i(x_i), yields$n-1$conditions. Similarly, s''_{i-1}(x_i) = s''_i(x_i), resultsinadditional$n-1$conditions. Intotalwehave4$nco
$2equationstodeterminethem, leavinguswith 2 degreesoffreedomtobedetermined.$

Using the last equation we define two values for the second derivative, namely s''$_i(x_i) = f_i, ands''_i(x_{i+1}) = f_{i+1}, andsettingupastraightlinebetween$f$_i$ and f_{i+1} we have s$''_i(x) = \frac{f_i}{x_{i+1}-x_i}(x_{i+1}-x) + \frac{f_{i+1}}{x_{i+1}-x_i}(x-x_i), andintegratingtwiceoneobtainss_i(x) = \frac{f_i}{6(x_{i+1}-x_i)}(x_{i+1}-x)^3 + \frac{f_{i+1}}{6(x_{i+1}-x_i)}(x-x_i)^3 + c(x-x_i) + d(x_{i+1}-x). Usingtheconditions$s$_i(x_i) = y_i$ and $s_i(x_{i+1}) = y_{i+1}$ we can in turn determine the constants c and d resulting in

$$\begin{aligned} s_i(x) = & \quad \frac{f_i}{6(x_{i+1}-x_i)}(x_{i+1}-x)^3 + \frac{f_{i+1}}{6(x_{i+1}-x_i)}(x-x_i)^3 \\ + & \quad \left(\frac{y_{i+1}}{x_{i+1}-x_i} - \frac{f_{i+1}(x_{i+1}-x_i)}{6}\right)(x-x_i) + \left(\frac{y_i}{x_{i+1}-x_i} - \frac{f_i(x_{i+1}-x_i)}{6}\right)(x_{i+1}-x). \end{aligned} \tag{6.37}$$

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'$_{i-1}(x_i) = s'_i(x_i), andset$x=x$_i$. Defining $h_i = x_{i+1} - x_i$ we obtain finally the following expression h$_{i-1}f_{i-1} + 2(h_i + h_{i-1})f_i + h_if_{i+1} = \frac{6}{h_i}(y_{i+1}-y_i) - \frac{6}{h_{i-1}}(y_i - y_{i-1}), andintroducingtheshorthands$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 $\begin{bmatrix} u_1 & h_1 & 0 & \dots & & & \\ h_1 & u_2 & h_2 & 0 & \dots & & \\ 0 & h_2 & u_3 & h_3 & 0 & \dots & \\ \dots & & \dots & \dots & \dots & \dots & \dots \\ & \dots & & & 0 & h_{n-3} & u_{n-2} & h_{n-2} \\ & & & & & 0 & h_{n-2} & u_{n-1} \end{bmatrix}$

$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \dots \\ v_{n-2} \\ v_{n-1} \end{bmatrix}$ *.Notethatthisisasetoftridiagonalequationsandcanbesolvedthroughonly*O(n)*operations.*

It is easy to write your own program for the cubic spline method when you have written a slover for tridiagonal equations. We split the program into two tasks, one which finds the polynomial approximation and one which uses the polynomials approximation to find an interpolated value for a function. These functions are included in the programs of this chapter, see the codes cubicpsline.cpp and cubicsinterpol.cpp. Alternatively, you can solve exercise 6.4!

6.6 Iterative Methods

Till now we have dealt with so-called direct solvers such as Gaussian elimination and LU decomposition. Iterative solvers offer another strategy and are much used in partial differential equations. We start with a guess for the solution and then iterate till the solution does not change anymore.

6.6.1 Jacobi's method

It is a simple method for solving

$$\hat{A}\mathbf{x} = \mathbf{b},$$

where $\hat{A}$ is a matrix and $\mathbf{x}$ and $\mathbf{b}$ are vectors. The vector $\mathbf{x}$ is the unknown.

It is an iterative scheme where we start with a guess for the unknown, and after $k+1$ iterations we have

$$\mathbf{x}^{(k+1)} = \hat{D}^{-1}(\mathbf{b} - (\hat{L} + \hat{U})\mathbf{x}^{(k)}),$$

with $\hat{A} = \hat{D} + \hat{U} + \hat{L}$ and $\hat{D}$ being a diagonal matrix, $\hat{U}$ an upper triangular matrix and $\hat{L}$ a lower triangular matrix.

If the matrix $\hat{A}$ is positive definite or diagonally dominant, one can show that this method will always converge to the exact solution.

We can demonstrate Jacobi's method by a 4×4 matrix problem. We assume a guess for the initial vector elements, labeled $x_i^{(0)}$. This guess represents our first iteration. The new values are obtained by substitution

$$\begin{aligned}
x_1^{(1)} =& \quad (b_1 - a_{12}x_2^{(0)} - a_{13}x_3^{(0)} - a_{14}x_4^{(0)})/a_{11} \\
x_2^{(1)} =& \quad (b_2 - a_{21}x_1^{(0)} - a_{23}x_3^{(0)} - a_{24}x_4^{(0)})/a_{22} \\
x_3^{(1)} =& \quad (b_3 - a_{31}x_1^{(0)} - a_{32}x_2^{(0)} - a_{34}x_4^{(0)})/a_{33} \\
x_4^{(1)} =& \quad (b_4 - a_{41}x_1^{(0)} - a_{42}x_2^{(0)} - a_{43}x_3^{(0)})/a_{44},
\end{aligned}$$

which after $k+1$ iterations result in

$$\begin{aligned}
x_1^{(k+1)} =& \quad (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11} \\
x_2^{(k+1)} =& \quad (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22} \\
x_3^{(k+1)} =& \quad (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)} - a_{34}x_4^{(k)})/a_{33} \\
x_4^{(k+1)} =& \quad (b_4 - a_{41}x_1^{(k)} - a_{42}x_2^{(k)} - a_{43}x_3^{(k)})/a_{44},
\end{aligned}$$

We can generalize the above equations to

$$x_i^{(k+1)} = (b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k)})/a_{ii}$$

or in an even more compact form as

$$\mathbf{x}^{(k+1)} = \hat{D}^{-1}(\mathbf{b} - (\hat{L}+\hat{U})\mathbf{x}^{(k)}),$$

with $\hat{A} = \hat{D} + \hat{U} + \hat{L}$ and $\hat{D}$ being a diagonal matrix, $\hat{U}$ an upper triangular matrix and $\hat{L}$ a lower triangular matrix.

6.6.2 Gauss-Seidel

Our 4×4 matrix problem

$$\begin{aligned}
x_1^{(k+1)} =& \quad (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11} \\
x_2^{(k+1)} =& \quad (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22} \\
x_3^{(k+1)} =& \quad (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)} - a_{34}x_4^{(k)})/a_{33} \\
x_4^{(k+1)} =& \quad (b_4 - a_{41}x_1^{(k)} - a_{42}x_2^{(k)} - a_{43}x_3^{(k)})/a_{44},
\end{aligned}$$

can be rewritten as

$$\begin{aligned}
x_1^{(k+1)} =& \quad (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - a_{14}x_4^{(k)})/a_{11} \\
x_2^{(k+1)} =& \quad (b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - a_{24}x_4^{(k)})/a_{22} \\
x_3^{(k+1)} =& \quad (b_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)} - a_{34}x_4^{(k)})/a_{33} \\
x_4^{(k+1)} =& \quad (b_4 - a_{41}x_1^{(k+1)} - a_{42}x_2^{(k+1)} - a_{43}x_3^{(k+1)})/a_{44},
\end{aligned}$$

which allows us to utilize the preceding solution (forward substitution). This improves normally the convergence behavior and leads to the Gauss-Seidel method!

We can generalize these equations to the following form

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j>i} a_{ij} x_j^{(k)} - \sum_{j<i} a_{ij} x_j^{(k+1)} \right), \quad i = 1, 2, \ldots, n.$$

The procedure is generally continued until the changes made by an iteration are below some tolerance.

The convergence properties of the Jacobi method and the Gauss-Seidel method depend on the matrix $\hat{A}$. These methods converge when the matrix is symmetric positive-definite, or is strictly or irreducibly diagonally dominant. Both methods sometimes converge even if these conditions are not satisfied.

6.6.3 Successive over-relaxation

We can rewrite the above in a slightly more formal way and extend the methods to what is called successive over-relaxation. Given a square system of n linear equations with unknown $\mathbf{x}$:

$$\hat{A}\mathbf{x} = \mathbf{b}$$

where:

$$\hat{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Then A can be decomposed into a diagonal component D, and strictly lower and upper triangular components L and U:

$$\hat{A} = \hat{D} + \hat{L} + \hat{U},$$

where

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

The system of linear equations may be rewritten as:

$$(D + \omega L)\mathbf{x} = \omega \mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x}$$

for a constant $\omega > 1$. The method of successive over-relaxation is an iterative technique that solves the left hand side of this expression for x, using previous value for x on the right hand side. Analytically, this may be written as:

$$\mathbf{x}^{(k+1)} = (D+\omega L)^{-1}\left(\omega\mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x}^{(k)}\right).$$

However, by taking advantage of the triangular form of $(D+\omega L)$, the elements of $x^{(k+1)}$ can be computed sequentially using forward substitution:

$$x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j>i} a_{ij}x_j^{(k)} - \sum_{j<i} a_{ij}x_j^{(k+1)}\right), \quad i = 1,2,\ldots,n.$$

The choice of relaxation factor is not necessarily easy, and depends upon the properties of the coefficient matrix. For symmetric, positive-definite matrices it can be proven that $0 < \omega < 2$ will lead to convergence, but we are generally interested in faster convergence rather than just convergence.

6.6.4 Conjugate Gradient Method

The success of the Conjugate Gradient method for finding solutions of non-linear problems is based on the theory for of conjugate gradients for linear systems of equations. It belongs to the class of iterative methods for solving problems from linear algebra of the type

$$\hat{\mathbf{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}.$$

In the iterative process we end up with a problem like

$$\hat{\mathbf{r}} = \hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}},$$

where $\hat{\mathbf{r}}$ is the so-called residual or error in the iterative process.

The residual is zero when we reach the minimum of the quadratic equation

$$P(\hat{\mathbf{x}}) = \frac{1}{2}\hat{\mathbf{x}}^T\hat{\mathbf{A}}\hat{\mathbf{x}} - \hat{\mathbf{x}}^T\hat{\mathbf{b}},$$

with the constraint that the matrix $\hat{\mathbf{A}}$ is positive definite and symmetric. If we search for a minimum of the quantum mechanical variance, then the matrix $\hat{\mathbf{A}}$, which is called the Hessian, is given by the second-derivative of the variance. This quantity is always positive definite. If we vary the energy, the Hessian may not always be positive definite.

In the Conjugate Gradient method we define so-called conjugate directions and two vectors $\hat{\mathbf{s}}$ and $\hat{\mathbf{t}}$ are said to be conjugate if

$$\hat{\mathbf{s}}^T\hat{\mathbf{A}}\hat{\mathbf{t}} = 0.$$

The philosophy of the Conjugate Gradient method is to perform searches in various conjugate directions of our vectors $\hat{\mathbf{x}}_i$ obeying the above criterion, namely

$$\hat{\mathbf{x}}_i^T \hat{\mathbf{A}} \hat{\mathbf{x}}_j = 0.$$

Two vectors are conjugate if they are orthogonal with respect to this inner product. Being conjugate is a symmetric relation: if $\hat{\mathbf{s}}$ is conjugate to $\hat{\mathbf{t}}$, then $\hat{\mathbf{t}}$ is conjugate to $\hat{\mathbf{s}}$.

An example is given by the eigenvectors of the matrix

$$\hat{\mathbf{v}}_i^T \hat{\mathbf{A}} \hat{\mathbf{v}}_j = \lambda \hat{\mathbf{v}}_i^T \hat{\mathbf{v}}_j,$$

which is zero unless $i = j$.

Assume now that we have a symmetric positive-definite matrix $\hat{\mathbf{A}}$ of size $n \times n$. At each iteration $i+1$ we obtain the conjugate direction of a vector

$$\hat{\mathbf{x}}_{i+1} = \hat{\mathbf{x}}_i + \alpha_i \hat{\mathbf{p}}_i.$$

We assume that $\hat{\mathbf{p}}_i$ is a sequence of n mutually conjugate directions. Then the $\hat{\mathbf{p}}_i$ form a basis of R^n and we can expand the solution $\hat{\mathbf{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}$ in this basis, namely

$$\hat{\mathbf{x}} = \sum_{i=1}^{n} \alpha_i \hat{\mathbf{p}}_i.$$

The coefficients are given by

$$\mathbf{A}\mathbf{x} = \sum_{i=1}^{n} \alpha_i \mathbf{A}\mathbf{p}_i = \mathbf{b}.$$

Multiplying with $\hat{\mathbf{p}}_k^T$ from the left gives

$$\hat{\mathbf{p}}_k^T \hat{\mathbf{A}} \hat{\mathbf{x}} = \sum_{i=1}^{n} \alpha_i \hat{\mathbf{p}}_k^T \hat{\mathbf{A}} \hat{\mathbf{p}}_i = \hat{\mathbf{p}}_k^T \hat{\mathbf{b}},$$

and we can define the coefficients α_k as

$$\alpha_k = \frac{\hat{\mathbf{p}}_k^T \hat{\mathbf{b}}}{\hat{\mathbf{p}}_k^T \hat{\mathbf{A}} \hat{\mathbf{p}}_k}$$

If we choose the conjugate vectors $\hat{\mathbf{p}}_k$ carefully, then we may not need all of them to obtain a good approximation to the solution $\hat{\mathbf{x}}$. So, we want to regard the conjugate gradient method as an iterative method. This also allows us to solve systems where n is so large that the direct method would take too much time.

We denote the initial guess for $\hat{\mathbf{x}}$ as $\hat{\mathbf{x}}_0$. We can assume without loss of generality that

$$\hat{\mathbf{x}}_0 = 0,$$

or consider the system

$$\hat{\mathbf{A}}\hat{\mathbf{z}} = \hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}}_0,$$

instead.

One can show that the solution $\hat{\mathbf{x}}$ is also the unique minimizer of the quadratic form

$$f(\hat{\mathbf{x}}) = \frac{1}{2}\hat{\mathbf{x}}^T\hat{\mathbf{A}}\hat{\mathbf{x}} - \hat{\mathbf{x}}^T\hat{\mathbf{x}}, \quad \hat{\mathbf{x}} \in \mathbf{R}^n.$$

This suggests taking the first basis vector $\hat{\mathbf{p}}_1$ to be the gradient of f at $\hat{\mathbf{x}} = \hat{\mathbf{x}}_0$, which equals

$$\hat{\mathbf{A}}\hat{\mathbf{x}}_0 - \hat{\mathbf{b}},$$

and $\hat{\mathbf{x}}_0 = 0$ it is equal $-\hat{\mathbf{b}}$. The other vectors in the basis will be conjugate to the gradient, hence the name conjugate gradient method.

Let $\hat{\mathbf{r}}_k$ be the residual at the k-th step:

$$\hat{\mathbf{r}}_k = \hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}}_k.$$

Note that $\hat{\mathbf{r}}_k$ is the negative gradient of f at $\hat{\mathbf{x}} = \hat{\mathbf{x}}_k$, so the gradient descent method would be to move in the direction $\hat{\mathbf{r}}_k$. Here, we insist that the directions $\hat{\mathbf{p}}_k$ are conjugate to each other, so we take the direction closest to the gradient $\hat{\mathbf{r}}_k$ under the conjugacy constraint. This gives the following expression

$$\hat{\mathbf{p}}_{k+1} = \hat{\mathbf{r}}_k - \frac{\hat{\mathbf{p}}_k^T\hat{\mathbf{A}}\hat{\mathbf{r}}_k}{\hat{\mathbf{p}}_k^T\hat{\mathbf{A}}\hat{\mathbf{p}}_k}\hat{\mathbf{p}}_k.$$

We can also compute the residual iteratively as

$$\hat{\mathbf{r}}_{k+1} = \hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}}_{k+1},$$

which equals

$$\hat{\mathbf{b}} - \hat{\mathbf{A}}(\hat{\mathbf{x}}_k + \alpha_k\hat{\mathbf{p}}_k),$$

or

$$(\hat{\mathbf{b}} - \hat{\mathbf{A}}\hat{\mathbf{x}}_k) - \alpha_k\hat{\mathbf{A}}\hat{\mathbf{p}}_k,$$

which gives

$$\hat{\mathbf{r}}_{k+1} = \hat{\mathbf{r}}_k - \hat{\mathbf{A}}\hat{\mathbf{p}}_k,$$

If we consider finding the minimum of a function f using Newton's method, that implies a search for a zero of the gradient of a function. Near a point x_i we have to second order

$$f(\hat{\mathbf{x}}) = f(\hat{\mathbf{x}}_i) + (\hat{\mathbf{x}} - \hat{\mathbf{x}}_i)\nabla f(\hat{\mathbf{x}}_i)\frac{1}{2}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_i)\hat{\mathbf{A}}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_i)$$

giving

$$\nabla f(\hat{\mathbf{x}}) = \nabla f(\hat{\mathbf{x}}_i) + \hat{\mathbf{A}}(\hat{\mathbf{x}} - \hat{\mathbf{x}}_i).$$

In Newton's method we set $\nabla f = 0$ and we can thus compute the next iteration point

$$\hat{\mathbf{x}} - \hat{\mathbf{x}}_i = \hat{\mathbf{A}}^{-1}\nabla f(\hat{\mathbf{x}}_i).$$

Subtracting this equation from that of $\hat{\mathbf{x}}_{i+1}$ we have

$$\hat{\mathbf{x}}_{i+1} - \hat{\mathbf{x}}_i = \hat{\mathbf{A}}^{-1}(\nabla f(\hat{\mathbf{x}}_{i+1}) - \nabla f(\hat{\mathbf{x}}_i)).$$

6.7 A vector and matrix class

We end this chapter by presenting a class which allows to manipulate one- and two-dimensional arrays. However, before we proceed, we would like to come with some general recommendations. Although it is useful to write your own classes, like the one included here, in general these classes may not be very efficient from a computational point of view. There are several libraries which include many interesting array features that allow us to write more compact code. The latter has the advantage that the code is lost likely easier to debug in case of errors (obviously assuming that the library is functioning correctly). Furthermore, if the proper functionalities are included, the final code may closely resemble the mathematical operations we wish to perform, increasing considerably the readability of our program. And finally, the code is in almost all casesmuch faster than the one we wrote!

In particular, we would like to recommend the C++ linear algebra library Armadillo, see http://arma.sourceforgenet. For those of you who are familiar with compiled programs like Matlab, the syntax is deliberately similar. Integer, floating point and complex numbers are supported, as well as a subset of trigonometric and statistics functions. Various matrix decompositions are provided through optional integration with LAPACK, or one of its high performance drop-in replacements (such as the multi-threaded MKL or ACML libraries). The selected examples included here show some examples on how to declare arrays and rearrange arrays or perform mathematical operations on say vectors or matrices. The first example here defines two random matrices of dimensionality 10×10 and performs a matrix-matrix multiplication using the *dgemm* function of the library BLAS.

Simple matrix-matrix multiplication of two random matrices

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

int main(int argc, char** argv)
  {
  mat A = randu<mat>(10,10);
```

```
 mat B = randu<mat>(10,10);
 // Matrix-matrix multiplication
 cout << A*B << endl;
 return 0;
 }
```

In the next example we compute the determinant of a 5×5 matrix, its inverse and perform thereafter several operations on various matrices.

Determinant and inverse of a matrix

```
#include <iostream>
#include "armadillo"
using namespace arma;
using namespace std;

int main(int argc, char** argv)
 {
 cout << "Armadillo version: " << arma_version::as_string() << endl;
 mat A;
 // Hard coding of the matrix
 // endr indicates "end of row"
 A << 0.165300 << 0.454037 << 0.995795 << 0.124098 << 0.047084 << endr
   << 0.688782 << 0.036549 << 0.552848 << 0.937664 << 0.866401 << endr
   << 0.348740 << 0.479388 << 0.506228 << 0.145673 << 0.491547 << endr
   << 0.148678 << 0.682258 << 0.571154 << 0.874724 << 0.444632 << endr
   << 0.245726 << 0.595218 << 0.409327 << 0.367827 << 0.385736 << endr;
 // .n_rows = number of rows
 // .n_cols = number of columns
 cout << "A.n_rows = " << A.n_rows << endl;
 cout << "A.n_cols = " << A.n_cols << endl;
 // Print the matrix A
 A.print("A =");
 // Computation of the determinant
 cout << "det(A) = " << det(A) << endl;
 // inverse
 cout << "inv(A) = " << endl << inv(A) << endl;
 // save to disk
 A.save("MatrixA.txt", raw_ascii);
 // Define a new matrix B which reads A from file
 mat B;
 B.load("MatrixA.txt");
 B += 5.0*A;
 B.print("The matrix B:");
 // generate the identity matrix
 mat C = eye<mat>(4,4);
 // transpose of B
 cout << "trans(B) =" << endl;
 // maximum from each column (traverse along rows)
 cout << "max(B) =" << endl;
```

```
  cout << max(B) << endl;
  // sum of all elements B
  cout << "sum(sum(B)) = " << sum(sum(B)) << endl;
  cout << "accu(B) = " << accu(B) << endl;
  // trace = sum along diagonal
  cout << "trace(B) = " << trace(B) << endl;
  // random matrix -- values are uniformly distributed in the [0,1] interval
  mat D = randu<mat>(4,4);
  D.print("Matrix D:");
  // sum of four matrices (no temporary matrices are created)
  mat E = A+B + C + D;
  F.print("F:");
  return 0;
}
```

For more examples, please consult the online manual, see http://arma.sourceforgenet.

6.7.1 How to construct your own matrix-vector class

The rest of this section shows how one can build a matrix-vector class. We first give an example of a function which use the header file `Array.h`.

```
#include "Array.h"

#include <iostream>
using namespace std;

int main(){

  // Create an array with (default) nrows = 1, ncols = 1:
  Array<double> v1;

  // Redimension the array to have length n:
  int n1 = 3;
  v1.redim(n1);

  // Extract the length of the array:
  const int length = v1.getLength();

  // Create a narray of specific length:
  int n2 = 5;
  Array<double> v2(n2);

  // Create an array as a copy of another one:
  Array<double> v5(v1);

  // Assign the entries in an array:
  v5(0) = 3.0;
  v5(1) = 2.5;
```

```
v5(2) = 1.0;

for(int i=0; i<3; i++){
  cout << v5(i) << endl;
}

// Extract the ith component of an array:
int i = 2;
double value = v5(1);
cout << "value: " << value << endl;

// Set an array equal another one:
Array<double> v6 = v5;

for(int i=0; i<3; i++){
  v1(i) = 1.0;
  v2(i) = 2.0;
}

// Create a two-dimensional array (matrix):
Array<double> matrix(2, 2);

// Fill the array:
matrix(0,0) = 1;
matrix(0,1) = 2;
matrix(1,0) = 3;
matrix(1,1) = 4;

// Get the entries in the array:
cout << "\nMatrix: " << endl;
for(int i=0; i<2; i++){
  for(int j=0; j<2; j++){
    cout << matrix(i,j) << " ";
  }
  cout << endl;
}

// Assign an entry of the matrix to a variable:
double scalar = matrix(0,0);
const double b = matrix(1,1);


Array<double> vector(2);
vector(0) = 1.0;
vector(1) = 2.0;

Array<double> v = vector;
Array<double> A = matrix;
Array<double> u(2);
```

```
  cout << "\nMatrix: " << endl;
  for(int i=0; i<2; i++){
    for(int j=0; j<2; j++){
      cout << matrix(i,j) << " ";
    }
    cout << endl;
  }

  Array<double> a(2,2);
  a(1,1) = 5.0;

  // Arithmetic operations with arrays using a
  // syntax close to the mathematical language
  Array<double> w = v1 + 2.0*v2;

  // Create multidimensional matrices and assign values to them:
  int N = 3;
  Array<double> multiD; multiD.redim(N,N,N);
  for(int i=0; i<N; i++){
    for(int j=0; j<N; j++){
      for(int k=0; k<N; k++){
        cout << "multD(i,j,k) = " << multiD(i,j,k) << endl;
      }
    }
  }

  multiD(1,2,3) = 4.0;
  cout << "multiD(1,2,3) = " << multiD(1,2,3) << endl;
}
```

The header file follows here

```
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
#include <sstream>
#include <iomanip>
#include <cstdlib>

using namespace std;

template<class T>
class Array{
  private:
    static const int MAXDIM = 6;
    T *data ;      /**> One-dimensional array of data.*/
    int size[MAXDIM];  /**> Size of each dimension.*/
    int ndim;      /**> Number of dimensions occupied. */
int length;     /**> Total number of entries.*/

int dx1, dx2, dx3, dx4, dx5;

void allocate(int ni=0, int nj=0, int nk=0, int nl=0, int nm=0, int nn=0){
 ndim = MAXDIM;

 // Set the number of entries in each dimension.
 size[0]=ni;
 size[1]=nj;
 size[2]=nk;
 size[3]=nl;
 size[4]=nm;
 size[5]=nn;


 // Set the number of dimensions used.
 if(size[5] == 0)
  ndim--;
 if(size[4] == 0)
  ndim--;
 if(size[3] == 0)
  ndim--;
 if(size[2] == 0)
  ndim--;
 if(size[1] == 0)
  ndim--;
 if(size[0] == 0){
  ndim   = 0;
  length  = 0;
  data  = NULL;
 }else{
  try{
   int i;

   // Set the length (total number of entries) of the one-dimensional array.
   length = 1;
   for(i=0; i<ndim; i++)
    length *= size[i];

    data = new T[length];

    dx1 =   size[0];
    dx2 = dx1*size[1];
    dx3 = dx2*size[2];
    dx4 = dx3*size[3];
    dx5 = dx4*size[4];

  }catch(std::bad_alloc&){
        std::cerr << "Array::allocate -- unable to allocate array of length " <<
            length << std::endl;
        exit(1);
      }
    }

  }

public:


  /**
  * @brief Constructor with default arguments.
  *
  * Creates an array with one or two-dimensions.
  *
  * @param int nrows. Number of rows in the array.
  * @param int ncolsd. Number of columns in the array.
  **/
  Array(int ni=0, int nj=0, int nk=0, int nl=0, int nm=0, int nn=0){
    // Allocate memory
    allocate(ni,nj,nk,nl,nm,nn);
  } // end constructor


  //! Constructor
  Array(T* array, int ndim_, int size_[]){
    ndim = ndim_;

    length = 1;
    int i;
    for(i=0; i<ndim; i++){
      size[i] = size_[i]; // Copy only the ndim entries. The rest is zero by default.
      length *= size[i];
    }

    // Now when we known the length, we should not forget to allocate memory!!!!
    data = new T[length];

    // Copy the entries from array to data:
    for(i=0; i<length; i++){
      data[i] = array[i];
    }

  } // End constructor.
```

```
//! Copy constructor
Array(const Array<T>& array);

//! Destructor
~Array();



/**
* @brief Checks the validity of the indexing.
* @param i, an integer for indexing the rows.
* @param j, an integer for indexing the columns.
**/
bool indexOk(int i, int j=0) const;

/**
* @brief Change the dimensions of an array.
* @param ni number of entries in the first dimension.
* @param nj number of entries in the second dimension.
* @param nk number of entries in the third dimension.
* @param nl number of entries in the fourth dimension.
* @param nm number of entries in the fifth dimension.
* @param nn number of entries in the sixth dimension.
**/
bool redim(int ni, int nj=0, int nk=0, int nl=0, int nm=0, int nn=0);

/**
* @return The total number of entries in the array, i.e., the sum of the entries
    in all the dimensions.
**/
int getLength()const{return length;}


/**
* @return The number of rows in a matrix.
**/
int getRows() const {return size[0];}

/**
* @return Returns the number of columns in a matrix.
**/
int getColumns() const {return size[1];}


/** @brief Gives the number of entries in a dimension.
*
* @param i An integer from 0 to 5 indicating the dimension we want to explore.
* @return size[i] An integer for the number of elements in the dimension number i.
**/
int dimension(int i) const{return size[i];}
```

```
/**
* The number of dimensions in the array.
**/
int getNDIM()const{return ndim;}


/**
* @return A constant pointer to the array of data.
* This function can be used to interface C++ with Python/Fortran/C.
**/
const T* getPtr() const;


/**
* @return A pointer to the array of data.
* This function can be used to interface C++ with Python/Fortran/C.
**/
T* getPtr();


/**
* @return A pointer to an array with information on the length of each dimension.
**/
int* getPtrSize();




/*************************************************************/
/*                OPERATORS                  */
/*************************************************************/

//! Assignment operator
Array<T>& operator=(const Array<T>& array);

//! Sum operator
Array<T> operator+(const Array<T>& array);

//! Substraction operator
Array<T> operator-(const Array<T>& array)const; /// w=u-v;


//! Multiplication operator
//Array<T> operator*(const Array<T>& array);


//! Assigment by addition operator
```

```
Array<T>& operator+=(const Array<T>& w);


//! Assignment by substraction operator
Array<T>& operator-=(const Array<T>& w);


//! Assignment by scalar product operator
Array<T>& operator*=(double scalar);

//! Assignment by division operator
Array<T>& operator/=(double scalar);

//! Index operators
const T& operator()(int i)const;
const T& operator()(int i, int j)const;
const T& operator()(int i, int j, int k)const;
const T& operator()(int i, int j, int k, int l)const;
const T& operator()(int i, int j, int k, int l, int m)const;
const T& operator()(int i, int j, int k, int l, int m, int n)const;

T& operator()(int i);
T& operator()(int i, int j);
T& operator()(int i, int j, int k);
T& operator()(int i, int j, int k, int l);
T& operator()(int i, int j, int k, int l, int m);
T& operator()(int i, int j, int k, int l, int m, int n);



/****************************************************************/
/*             FRIEND FUNCTIONS                  */
/****************************************************************/
//! Unary operator +
template <class T2>
friend Array<T> operator+ (const Array<T>&);  // u = + v

//! Unary operator -
template <class T2>
friend Array<T> operator-(const Array<T>&);  // u = - v




/**
* Premultiplication by a floating point number:
* \f$\mathbf{u} = a \mathbf{v}\f$,
* where \f$a\f$ is a scalar and \f$\mathbf{v}\f$ is a array.
**/
```

```
   template <class T2>
   friend Array<T> operator*(double, const Array<T>&); // u = a*v

   /**
   * Postmultiplication by a floating point number:
   * \f$\mathbf{u} = \mathbf{v} a\f$,
   * where \f$a\f$ is a scalar and \f$\mathbf{v}\f$ is a array.
   **/
   template <class T2>
   friend Array<T> operator*(const Array<T>&, double); // u = v*a


   /**
   * Division of the entries of a array by a scalar.
   **/
   template <class T2>
   friend Array<T> operator/(const Array<T>&, double); // u = v/a





};

#include "Array.cpp"


// Destructor
template <class T>
inline Array<T>::~Array(){delete[] data;}

// Index operators
template <class T>
inline const T& Array<T>::operator()(int i)const {
  #if CHECKBOUNDS_ON
    indexOk(i);
  #endif
  return data[i];
}



template <class T>
inline const T& Array<T>::operator()(int i, int j)const {
  #if CHECKBOUNDS_ON
    indexOk(i,j);
  #endif

  return data[i + j*dx1];
```

```
}

template <class T>
inline const T& Array<T>::operator()(int i, int j, int k)const {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k);
  #endif

  return data[i + j*dx1 + k*dx2];
}


template <class T>
inline const T& Array<T>::operator()(int i, int j, int k, int l)const {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k,l);
  #endif

  return data[i + j*dx1 + k*dx2 + l*dx3];
}


template <class T>
inline const T& Array<T>::operator()(int i, int j, int k, int l, int m)const {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k,l, m);
  #endif

  return data[i + j*dx1 + k*dx2 + l*dx3 + m*dx4];
}


template <class T>
inline const T& Array<T>::operator()(int i, int j, int k, int l, int m, int n)const {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k,l,m,n);
  #endif

  return data[i + j*dx1 + k*dx2 + l*dx3 + m*dx4 + n*dx5];
}

template <class T>
inline T& Array<T>::operator()(int i) {
  #if CHECKBOUNDS_ON
    indexOk(i);
  #endif
  return data[i];
}
```

```
template <class T>
inline T& Array<T>::operator()(int i, int j) {
  #if CHECKBOUNDS_ON
    indexOk(i,j);
  #endif

  return data[i + j*dx1];
}


template <class T>
inline T& Array<T>::operator()(int i, int j, int k) {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k);
  #endif

  return data[i + j*dx1 + k*dx2];
}


template <class T>
inline T& Array<T>::operator()(int i, int j, int k, int l) {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k,l);
  #endif

  return data[i + j*dx1 + k*dx2 + l*dx3];
}


template <class T>
inline T& Array<T>::operator()(int i, int j, int k, int l, int m) {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k,l,m);
  #endif

  return data[i + j*dx1 + k*dx2 + l*dx3 + m*dx4];
}


template <class T>
inline T& Array<T>::operator()(int i, int j, int k, int l, int m, int n) {
  #if CHECKBOUNDS_ON
    indexOk(i,j,k,l,m,n);
  #endif

  return data[i + j*dx1 + k*dx2 + l*dx3 + m*dx4 + n*dx5];
}
```

```
template <class T>
inline const T* Array<T>::getPtr() const {return data;}



template <class T>
inline T* Array<T>::getPtr(){return data; }


template <class T>
inline int* Array<T>::getPtrSize(){return size;}


// template <class T>
// inline int Array<T>::dim()const{return ndim;}


/*****************************************************************/
/*          IMPLEMENTATION OF FRIEND FUNCTIONS        */
/*****************************************************************/

/*****************************************************************/
/*         (Arithmetic) Unary operators          */
/*****************************************************************/
//! Unary operator +
template <class T>
inline Array<T> operator+(const Array<T>& v){ // u = + v
  return v;
}


//! Unary operator -
template <class T>
inline Array<T> operator-(const Array<T>& v){ // u = - v
  return Array<T>(v.size[0],v.size[1]) -v;
}


//! Postmultiplication operator
template <class T>
inline Array<T> operator*(const Array<T>& v, double scalar){ // u = v*a
  return Array<T>(v) *= scalar;
}


//! Premultiplication operator.
```

```
template <class T>
inline Array<T> operator*(double scalar, const Array<T>& v){ // u = a*v
 return v*scalar; // Note the call to postmultiplication operator defined above
}


//! Division of the entries in a array by a scalar
template <class T>
inline Array<T> operator/(const Array<T>& v, double scalar){
 if(!scalar) std::cout << "Division by zero!" << std::endl;
 return (1.0/scalar)*v;
}

#endif
```

6.8 Exercises

The aim of this exercise is to write your own Gaussian elimination code.

1. Consider the linear system of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= w_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= w_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= w_3. \end{aligned}$$

   This can be written in matrix form as

$$\mathbf{A}\mathbf{x} = \mathbf{w}.$$

   We specialize here to the following case

$$\begin{aligned} -x_1 + x_2 - 4x_3 &= 0 \\ 2x_1 + 2x_2 &= 1 \\ 3x_1 + 3x_2 + 2x_3 &= \tfrac{1}{2}. \end{aligned}$$

   Obtain the solution (by hand) of this system of equations by doing Gaussian elimination.

2. Write therafter a program which implements Gaussian elimination (with pivoting) and solve the above system of linear equations. How many floating point operations are involved in the solution via Gaussian elimination without pivoting? Can you estimate the number of floating point operations with pivoting?

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)

$$L_{ii} = \left(A_{ii} - \sum_{k=0}^{i-1} L^2_{ik}\right)^{1/2}. \tag{6.38}$$

- within the loop over i, introduce a new loop which goes from $j = i+1$ to $n-1$ and calculate

$$L_{ji} = \frac{1}{L_{ii}}\left(A_{ij} - \sum_{k=0}^{i-1} L_{ik} l_{jk}\right). \tag{6.39}$$

For the Cholesky algorithm we have always that $L_{ii} > 0$ and the problem with exceedingly large matrix elements does not appear and hence there is no need for pivoting. Write a function which performs the Cholesky decomposition. Test your program against the standard LU decomposition by using the matrix

$$\mathbf{A} = \begin{pmatrix} 6 & 3 & 2 \\ 3 & 2 & 1 \\ 2 & 1 & 1 \end{pmatrix} \tag{6.40}$$

Finally, use the Cholesky method to solve

$$\begin{aligned}
0.05x_1 + 0.07x_2 + 0.06x_3 + 0.05x_4 &= 0.23 \\
0.07x_1 + 0.10x_2 + 0.08x_3 + 0.07x_4 &= 0.32 \\
0.06x_1 + 0.08x_2 + 0.10x_3 + 0.09x_4 &= 0.33 \\
0.05x_1 + 0.07x_2 + 0.09x_3 + 0.10x_4 &= 0.31
\end{aligned}$$

You can also use the LU codes for linear equations to check the results.

In this exercise we are going to solve the one-dimensional Poisson equation in terms of linear equations.

1. We are going to solve the one-dimensional Poisson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations.

   The three-dimensional Poisson equation is a partial differential equation,

$$\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}+\frac{\partial^2\phi}{\partial z^2}=-\frac{\rho(x,y,z)}{\varepsilon_0},$$

   whose solution we will discuss in chapter **??**. The function $\rho(x,y,z)$ is the charge density and ϕ is the electrostatic potential. In this project we consider the one-dimensional case since there are a few situations, possessing a high degree of symmetry, where it is possible to find analytic solutions. Let us discuss some of these solutions.

   Suppose, first of all, that there is no variation of the various quantities in the y- and z-directions. In this case, Poisson's equation reduces to an ordinary differential equation in x, the solution of which is relatively straightforward. Consider for example a vacuum diode, in which electrons are emitted from a hot cathode and accelerated towards an anode. The anode is held at a large positive potential V_0 with respect to the cathode. We can think of this as an essentially one-dimensional problem. Suppose that the cathode is at $x=0$ and the anode at $x=d$. Poisson's equation takes the form

$$\frac{d^2\phi}{dx^2}=-\frac{\rho(x)}{\varepsilon_0},$$

   where $\phi(x)$ satisfies the boundary conditions $\phi(0)=0$ and $\phi(d)=V_0$. By energy conservation, an electron emitted from rest at the cathode has an x-velocity $v(x)$ which satisfies

$$\frac{1}{2}m_ev^2(x)-e\phi(x)=0.$$

   Furthermore, we assume that the current I is independent of x between the anode and cathode, otherwise, charge will build up at some points. From electromagnetism one can then show that the current I is given by $I=-\rho(x)v(x)A$, where A is the cross-sectional area of the diode. The previous equations can be combined to give

$$\frac{d^2\phi}{dx^2}=\frac{I}{\varepsilon_0A}\left(\frac{m_e}{2e}\right)^{1/2}\phi^{-1/2}.$$

   The solution of the above equation which satisfies the boundary conditions is

$$\phi=V_0\left(\frac{x}{d}\right)^{4/3},$$

   with

$$I=\frac{4}{9}\frac{\varepsilon_0A}{d^2}\left(\frac{2e}{m_e}\right)^{1/2}V_0^{3/2}.$$

This relationship between the current and the voltage in a vacuum diode is called the Child-Langmuir law.

Another physics example in one dimension is the famous Thomas-Fermi model, widely used as a mean-field model in simulations of quantum mechanical systems [6, 16], see Lieb for a newer and updated discussion [13]. Thomas and Fermi assumed the existence of an energy functional, and derived an expression for the kinetic energy based on the density of electrons, $\rho(r)$ in an infinite potential well. For a large atom or molecule with a large number of electrons. Schrödinger's equation, which would give the exact density and energy, cannot be easily handled for large numbers of interacting particles. Since the Poisson equation connects the electrostatic potential with the charge density, one can derive the following equation for potential V

$$\frac{d^2V}{dx^2} = \frac{V^{3/2}}{\sqrt{x}},$$

with $V(0) = 1$.

In our case we will rewrite Poisson's equation in terms of dimensionless variables. We can then rewrite the equation as

$$-u''(x) = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0.$$

and we define the discretized approximation to u as v_i with grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$. The step length or spacing is defined as $h = 1/(n+1)$. We have then the boundary conditions $v_0 = v_{n+1} = 0$. We approximate the second derivative of u with

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, \dots, n,$$

where $f_i = f(x_i)$. Show that you can rewrite this equation as a linear set of equations of the form

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}},$$

where $\mathbf{A}$ is an $n \times n$ tridiagonal matrix which we rewrite as

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{pmatrix}$$

and $\tilde{b}_i = h^2 f_i$.

In our case we will assume that $f(x) = (3x + x^2)e^x$, and keep the same interval and boundary conditions. Then the above differential equation has an analytic solution given by $u(x) = x(1-x)e^x$ (convince yourself that this is correct by inserting the solution in the Poisson equation). We will compare our numerical solution with this analytic result in the next exercise.

2. We can rewrite our matrix $\mathbf{A}$ in terms of one-dimensional vectors a, b, c of length $1:n$. Our linear equation reads

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{pmatrix}.$$

A tridiagonal matrix is a special form of banded matrix where all the elements are zero except for those on and immediately above and below the leading diagonal. The above tridiagonal system can be written as

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i,$$

for $i = 1, 2, \dots, n$. The algorithm for solving this set of equations is rather simple and requires two steps only, a decomposition and forward substitution and finally a backward substitution.

Your first task is to set up the algorithm for solving this set of linear equations. Find also the number of operations needed to solve the above equations. Show that they behave like $O(n)$ with n the dimensionality of the problem. Compare this with standard Gaussian elimination.

Then you should code the above algorithm and solve the problem for matrices of the size 10×10, 100×100 and 1000×1000. That means that you choose $n = 10$, $n = 100$ and $n = 1000$ grid points.

Compare your results (make plots) with the analytic results for the different number of grid points in the interval $x \in (0, 1)$. The different number of grid points corresponds to different step lengths h.

Compute also the maximal relative error in the data set $i = 1, \dots, n$,by setting up

$$\varepsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right),$$

as function of $log_{10}(h)$ for the function values u_i and v_i. For each step length extract the max value of the relative error. Try to increase n to $n = 10000$ and $n = 10^5$. Comment your results.

3. Compare your results with those from the LU decomposition codes for the matrix of size 1000×1000. Use for example the unix function *time* when you run your codes and compare the time usage between LU decomposition and your tridiagonal solver. Can you run the standard LU decomposition for a matrix of the size $10^5 \times 10^5$? Comment your results.

6.8.1 Solution

The program listed below encodes a possible solution to part b) of the above project. Note that we have employed Blitz++ as library and that the range of the various vectors are now shifted from their default ranges $(0 : n-1)$ to $(1 : n)$ and that we access vector elements as $a(i)$ instead of the standard C++ declaration $a[i]$.

The program reads from screen the name of the ouput file and the dimension of the problem, which in our case corresponds to the number of mesh points as well, in addition to the two endpoints. The function $f(x) = (3x+x^2)\exp(x)$ is included explicitely in the code. An obvious change is to define a separate function, allowing thereby for a generalization to other function $f(x)$.

```
/*
   Program to solve the one-dimensional Poisson equation
   -u''(x) = f(x) rewritten as a set of linear equations
   A u = f where A is an n x n matrix, and u and f are 1 x n vectors
   In this problem f(x) = (3x+x*x)exp(x) with solution u(x) = x(1-x)exp(x)
   The program reads from screen the name of the output file.
   Blitz++ is used here, with arrays starting from 1 to n
*/
#include <iomanip>
#include <fstream>
#include <blitz/array.h>
#include <iostream>
using namespace std;
using namespace blitz;

ofstream ofile;
// Main program only, no other functions
int main(int argc, char* argv[])
{
  char *outfilename;
  int i, j, n;
  double h, btemp;
  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
```

Figure 6.4: Numerical solution obtained with $n = 10$ compared with the analytical solution.

```
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  cout << "Read in number of mesh points" << endl;
  cin >> n;
  h = 1.0/( (double) n+1);
  // Use Blitz to allocate arrays
  // Use range to change default arrays from 0:n-1 to 1:n
  Range r(1,n);
  Array<double,1> a(r), b(r), c(r), y(r), f(r), temp(r);
  // set up the matrix defined by three arrays, diagonal, upper and lower diagonal
       band
  b = 2.0; a = -1.0 ; c = -1.0;
  // Then define the value of the right hand side f (multiplied by h*h)
  for(i=1; i <= n; i++){
    // Explicit expression for f, could code as separate function
    f(i) = h*h*(i*h*3.0+(i*h)*(i*h))*exp(i*h);
  }
  // solve the tridiagonal system, first forward substitution
  btemp = b(1);
  for(i = 2; i <= n; i++) {
    temp(i) = c(i-1) / btemp;
    btemp = b(i) - a(i) * temp(i);
    y(i) = (f(i) - a(i) * y(i-1)) / btemp;
  }
  // then backward substitution, the solution is in y()
  for(i = n-1; i >= 1; i--) {
    y(i) -= temp(i+1) * y(i+1);
  }
  // write results to the output file
  for(i = 1; i <= n; i++){
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << setw(15) << setprecision(8) << i*h;
    ofile << setw(15) << setprecision(8) << y(i);
    ofile << setw(15) << setprecision(8) << i*h*(1.0-i*h)*exp(i*h) <<endl;
  }
  ofile.close();
}
```

The program writes also the exact solution to file. In Fig. 6.4 we show the results obtained with $n = 10$. Even with so few points, the numerical solution is very close to the analytic answer. With $n = 100$ it is almost impossible to distinguish the numerical

Figure 6.5: Numerical solution obtained with $n = 10$ compared with the analytical solution.

solution from the analytical one, as shown in Fig. 6.5. It is therefore instructive to study the relative error, which we display in Table 6.2 as function of the step length $h = 1/(n+1)$.

Table 6.2: log_{10} values for the relative error and the step length h computed at $x = 0.5$.

n	$log_{10}(h)$	$\varepsilon_i = log_{10}(\lvert(v_i - u_i)/u_i\rvert)$
10	-1.04	-2.29
100	-2.00	-4.19
1000	-3.00	-6.18
10^4	-4.00	-8.18
10^5	-5.00	-9.19
10^6	-6.00	-6.08

The mathematical truncation we made when computing the second derivative goes like $O(h^2)$. Our results for n from $n = 10$ to somewhere between $n = 10^4$ and $n = 10^5$ result in a slope which is almost exactly equal 2,in good agreement with the mathematical truncation made. Beyond $n = 10^5$ the relative error becomes bigger, telling us that there is no point in increasing n. For most practical application a relative error between 10^{-6} and 10^{-8} is more than sufficient, meaning that $n = 10^4$ may be an acceptable number of mesh points. Beyond $n = 10^5$, numerical round off errors take over, as discussed in the previous chapter as well.

Write your own code for performing the cubic spline interpolation using either Blitz++ or Armadillo. Alternatively you can use the vector-matrix class included in this text.

Write your own code for the LU decomposition using the same libraries as in the previous exercise. Find also the number of floating point operations.

Solve exercise 6.3 by writing a code which implements both the iterative Jacobi method and the Gauss-Seidel method. Study carefully the number of iterations needed to achieve the exact result.

Extend thereafter your code for the iterative Jacobi method to a parallel version and compare with the results from the previous exercise.

Write your own code for the Conjugate gradient method.

Write your own code for matrix-matrix multiplications using Strassen's algorithm discussed in subsection 6.3.3 and compare the speed of your program with the matrix-matrix multiplication provided by the Armadillo library.

Bibliography

[1] *LAPACK – Linear Algebra PACKage*. http://www.netlib.org/lapack/.

[2] J.J. Barton and L.R. Nackman. *Scientific and Engineering C++*. Addison Wesley, 1994.

[3] J.R. Berryhill. *C++ Scientific Programming*. Wiley-Interscience, 2001.

[4] B.N. Datta. *Numerical Linear Algebra and Applications*. Brooks/Cole Publishing Company, 1995.

[5] J.W. Demmel. *Numerical Linear Algebra*. SIAM Publications, 1996.

[6] E. Fermi. Un metodo statistico per la determinazione di alcune proprietà dell'atomo. *Rend. Accad. Naz. Lincei*, 6:602, 1927.

[7] B.H. Flowers. *An Introduction to Numerical Methods in C++*. Oxford University Press, 2000.

[8] F. Franek. *Memory as a Programming Concept in C and C++*. Cambridge University Press, 2004.

[9] G.H. Golub and C.F. Van Loan. *Matrix Computations*. John Hopkins University Press, 1996.

[10] D. Kincaid and W. Cheney. *Numerical Analysis*. Brooks/Gole Publishing Company, 1996.

[11] R. Kress. *Numerical Analysis*. Springer, 1998.

[12] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. *ACM Trans. Math. Soft.*, 5:308, 1979.

[13] Elliott H. Lieb. Thomas-fermi and related theories of atoms and molecules. *Rev. Mod. Phys.*, 53(4):603–641, Oct 1981.

[14] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C++, The art of scientific Computing*. Cambridge University Press, 1999.

[15] K.A. Reek. *Pointers on C*. Addison Wesley, 1998.

[16] L. H. Thomas. The calculation of atomic fields. *Proc. Camb. Phil. Soc.*, 23:542, 1927.

[17] L.N. Trefethen and D. Bau III. *Numerical Linear Algebra*. SIAM Publications, 1997.

[18] T. Veldhuizen. *Blitz++ User's Guide*. http://www.oonumerics.org/blitz/, 2003.

Chapter 7

Eigensystems

7.1 Introduction

We present here two methods for solving directly eigenvalue problems using similarity transformations. One is the familiar Jacobi rotation method while the second method is based on transforming the matrix to tridiagonal form using Householder's algorithm. We discuss also so-called power methods and conclude with a discussion of iterative algorithms. These are particularly interesting for eigenvalue problems of large dimnesionality.

Together with linear equations and least squares, the third major problem in matrix computations deals with the algebraic eigenvalue problem. Here we limit our attention to the symmetric case. We focus in particular on two similarity transformations, the Jacobi method, the famous QR algoritm with Householder's method for obtaining a triangular matrix and Francis' algorithm for the final eigenvalues. Our presentation follows closely that of Golub and Van Loan, see Ref. [1].

7.2 Eigenvalue problems

Let us consider the matrix **A** of dimension n. The eigenvalues of **A** are defined through the matrix equation

$$\mathbf{A}\mathbf{x}^{(\nu)} = \lambda^{(\nu)}\mathbf{x}^{(\nu)}, \tag{7.1}$$

where $\lambda^{(\nu)}$ are the eigenvalues and $\mathbf{x}^{(\nu)}$ the corresponding eigenvectors. Unless otherwise stated, when we use the wording eigenvector we mean the right eigenvector. The left eigenvector is defined as

$$\mathbf{x}^{(\nu)}{}_L\mathbf{A} = \lambda^{(\nu)}\mathbf{x}^{(\nu)}{}_L$$

The above right eigenvector problem is equivalent to a set of n equations with n unknowns x_i

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = & \quad \lambda x_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = & \quad \lambda x_2 \\ \dots & \quad \dots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = & \quad \lambda x_n. \end{aligned}$$

We can rewrite Eq. (7.1) as

$$\left(\mathbf{A} - \lambda^{(\nu)} I\right) \mathbf{x}^{(\nu)} = 0,$$

with I being the unity matrix. This equation provides a solution to the problem if and only if the determinant is zero, namely

$$\left| \mathbf{A} - \lambda^{(\nu)} \mathbf{I} \right| = 0,$$

which in turn means that the determinant is a polynomial of degree n in λ. The eigenvalues of a matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ are thus the n roots of its characteristic polynomial P(λ) = $det(\lambda \mathbf{I} - \mathbf{A}), or P(\lambda) = \prod_{i=1}^{n} (\lambda_i - \lambda). The set of these roots is called the spectrum and is denoted as \lambda(\mathbf{A})$. If $\lambda(\mathbf{A}) = \{\lambda_1, \lambda_2, \dots, \lambda_n\}$ then we have

$$det(\mathbf{A}) = \lambda_1 \lambda_2 \dots \lambda_n,$$

the trace of $\mathbf{A}$ is $Tr(\mathbf{A}) = \lambda_1 + \lambda_2 + \cdots + \lambda_n$.

Procedures based on these ideas can be used if only a small fraction of all eigenvalues and eigenvectors are required or if the matrix is on a tridiagonal form, but the standard approach to solve Eq. (7.1) is to perform a given number of similarity transformations so as to render the original matrix $\mathbf{A}$ in either a diagonal form or as a tridiagonal matrix which then can be be diagonalized by computational very effective procedures.

The first method leads us to Jacobi's method whereas the second one is given by Householder's algorithm for tridiagonal transformations. We will discuss both methods below.

7.3 Similarity transformations

In the present discussion we assume that our matrix is real and symmetric, that is $\mathbf{A} \in \mathbb{R}^{n\times n}$. The matrix $\mathbf{A}$ has n eigenvalues $\lambda_1 \dots \lambda_n$ (distinct or not). Let $\mathbf{D}$ be the diagonal matrix with the eigenvalues on the diagonal

$$\mathbf{D} = \left(\begin{array}{ccccccc} \lambda_1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \lambda_3 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \lambda_{n-1} & \\ 0 & \dots & \dots & \dots & \dots & 0 & \lambda_n \end{array} \right).$$

If $\mathbf{A}$ is real and symmetric then there exists a real orthogonal matrix $\mathbf{S}$ such that

$$\mathbf{S}^T\mathbf{A}\mathbf{S} = \mathrm{diag}(\lambda_1, \lambda_2, \dots, \lambda_n),$$

and for $j = 1:n$ we have $\mathbf{AS}(:,j) = \lambda_j\mathbf{S}(:,j)$. See chapter 8 of Ref. [1] for proof.

To obtain the eigenvalues of $\mathbf{A} \in \mathbb{R}^{n\times n}$, the strategy is to perform a series of similarity transformations on the original matrix $\mathbf{A}$, in order to reduce it either into a diagonal form as above or into a tridiagonal form.

We say that a matrix $\mathbf{B}$ is a similarity transform of $\mathbf{A}$ if

$$\mathbf{B} = \mathbf{S}^T\mathbf{A}\mathbf{S}, \qquad \text{where} \qquad \mathbf{S}^T\mathbf{S} = \mathbf{S}^{-1}\mathbf{S} = \mathbf{I}.$$

The importance of a similarity transformation lies in the fact that the resulting matrix has the same eigenvalues, but the eigenvectors are in general different. To prove this we start with the eigenvalue problem and a similarity transformed matrix $\mathbf{B}$.

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \qquad \text{and} \qquad \mathbf{B} = \mathbf{S}^T\mathbf{A}\mathbf{S}.$$

We multiply the first equation on the left by $\mathbf{S}^T$ and insert $\mathbf{S}^T\mathbf{S} = \mathbf{I}$ between $\mathbf{A}$ and $\mathbf{x}$. Then we get

$$(\mathbf{S^TAS})(\mathbf{S^Tx}) = \lambda\mathbf{S^Tx}, \tag{7.2}$$

which is the same as

$$\mathbf{B}\left(\mathbf{S^Tx}\right) = \lambda\left(\mathbf{S^Tx}\right).$$

The variable λ is an eigenvalue of $\mathbf{B}$ as well, but with eigenvector $\mathbf{S^Tx}$.

The basic philosophy is to

- either apply subsequent similarity transformations so that

$$\mathbf{S_N^T}\dots\mathbf{S_1^TAS_1}\dots\mathbf{S_N} = \mathbf{D}, \tag{7.3}$$

- or apply subsequent similarity transformations so that $\mathbf{A}$ becomes tridiagonal. Thereafter, techniques for obtaining eigenvalues from tridiagonal matrices can be used.

Let us look at the first method, better known as Jacobi's method or Given's rotations.

7.4 Jacobi's method

Consider an $(n \times n)$ orthogonal transformation matrix

$$\mathbf{S} = \begin{pmatrix} 1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\ 0 & 0 & \dots & cos\theta & 0 & \dots & 0 & sin\theta \\ 0 & 0 & \dots & 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 & \dots \\ 0 & 0 & \dots & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & \dots & -sin\theta & \dots & \dots & 0 & cos\theta \end{pmatrix}$$

with property $\mathbf{S^T} = \mathbf{S^{-1}}$. It performs a plane rotation around an angle θ in the Euclidean $n-$dimensional space. It means that the matrix elements that differ from zero are given by

$$s_{kk} = s_{ll} = cos\theta, s_{kl} = -s_{lk} = -sin\theta, s_{ii} = -s_{ii} = 1 \quad i \neq k \quad i \neq l,$$

A similarity transformation

$$\mathbf{B} = \mathbf{S}^T\mathbf{A}\mathbf{S},$$

results in

$$\begin{aligned} b_{ii} &= a_{ii}, i \neq k, i \neq l \\ b_{ik} &= a_{ik}cos\theta - a_{il}sin\theta, i \neq k, i \neq l \\ b_{il} &= a_{il}cos\theta + a_{ik}sin\theta, i \neq k, i \neq l \\ b_{kk} &= a_{kk}cos^2\theta - 2a_{kl}cos\theta sin\theta + a_{ll}sin^2\theta \\ b_{ll} &= a_{ll}cos^2\theta + 2a_{kl}cos\theta sin\theta + a_{kk}sin^2\theta \\ b_{kl} &= (a_{kk} - a_{ll})cos\theta sin\theta + a_{kl}(cos^2\theta - sin^2\theta) \end{aligned}$$

The angle θ is arbitrary. The recipe is to choose θ so that all non-diagonal matrix elements b_{kl} become zero.

The algorithm is then quite simple. We perform a number of iterations until the sum over the squared non-diagonal matrix elements are less than a prefixed test (ideally equal zero). The algorithm is more or less foolproof for all real symmetric matrices, but becomes much slower than methods based on tridiagonalization for large matrices.

The main idea is thus to reduce systematically the norm of the off-diagonal matrix elements of a matrix $\mathbf{A}$

$$\mathrm{off}(\mathbf{A}) = \sqrt{\sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n} a_{ij}^2}.$$

To demonstrate the algorithm, we consider the simple 2×2 similarity transformation of the full matrix. The matrix is symmetric, we single out $1 \le k < l \le n$ and use the abbreviations $c = \cos\theta$ and $s = \sin\theta$ to obtain

$$\begin{pmatrix} b_{kk} & 0 \\ 0 & b_{ll} \end{pmatrix} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} a_{kk} & a_{kl} \\ a_{lk} & a_{ll} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}.$$

We require that the non-diagonal matrix elements $b_{kl} = b_{lk} = 0$, implying that

$$a_{kl}(c^2 - s^2) + (a_{kk} - a_{ll})cs = b_{kl} = 0.$$

If $a_{kl} = 0$ one sees immediately that $\cos\theta = 1$ and $\sin\theta = 0$.

The Frobenius norm of an orthogonal transformation is always preserved. The Frobenius norm is defined as

$$||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |a_{ij}|^2}.$$

This means that for our 2×2 case we have

$$2a_{kl}^2 + a_{kk}^2 + a_{ll}^2 = b_{kk}^2 + b_{ll}^2,$$

which leads to

$$\mathrm{off}(\mathbf{B})^2 = ||\mathbf{B}||_F^2 - \sum_{i=1}^{n} b_{ii}^2 = \mathrm{off}(\mathbf{A})^2 - 2a_{kl}^2,$$

since

$$||\mathbf{B}||_F^2 - \sum_{i=1}^{n} b_{ii}^2 = ||\mathbf{A}||_F^2 - \sum_{i=1}^{n} a_{ii}^2 + (a_{kk}^2 + a_{ll}^2 - b_{kk}^2 - b_{ll}^2).$$

This result means that the matrix $\mathbf{A}$ moves closer to diagonal form for each transformation.

Defining the quantities $\tan\theta = t = s/c$ and

$$\tau = \frac{a_{ll} - a_{kk}}{2a_{kl}},$$

we obtain the quadratic equation

$$t^2 + 2\tau t - 1 = 0,$$

resulting in

$$t = -\tau \pm \sqrt{1+\tau^2},$$

and c and s are easily obtained via

$$c = \frac{1}{\sqrt{1+t^2}},$$

and $s = tc$. Choosing t to be the smaller of the roots ensures that $|\theta| \leq \pi/4$ and has the effect of minimizing the difference between the matrices **B** and **A** since

$$||\mathbf{B}-\mathbf{A}||_F^2 = 4(1-c)\sum_{i=1,i\neq k,l}^{n}(a_{ik}^2+a_{il}^2)+\frac{2a_{kl}^2}{c^2}.$$

The main idea is thus to reduce systematically the norm of the off-diagonal matrix elements of a matrix **A**

$$\mathrm{off}(\mathbf{A}) = \sqrt{\sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n} a_{ij}^2}.$$

To implement the Jacobi algorithm we can proceed as follows

- Choose a tolerance ε, making it a small number, typically 10^{-8} or smaller.
- Setup a **`while`**-test where one compares the norm of the newly computed off-diagonal matrix elements

  $$\mathrm{off}(\mathbf{A}) = \sqrt{\sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n} a_{ij}^2} > \varepsilon.$$

  This is however a very time-comsuming test which can be replaced by the simpler test

  $$\max(a_{ij}^2) > \varepsilon.$$

- Now choose the matrix elements a_{kl} so that we have those with largest value, that is $|a_{kl}| = \max_{i\neq j}|a_{ij}|$.
- Compute thereafter $\tau = (a_{ll}-a_{kk})/2a_{kl}$, $\tan\theta$, $\cos\theta$ and $\sin\theta$.
- Compute thereafter the similarity transformation for this set of values (k,l), obtaining the new matrix $\mathbf{B} = \mathbf{S}(k,l,\theta)^T\mathbf{A}\mathbf{S}(k,l,\theta)$.
- Continue till

  $$\max(a_{ij}^2) \leq \varepsilon.$$

The convergence rate of the Jacobi method is however poor, one needs typically $3n^2-5n^2$ rotations and each rotation requires $4n$ operations, resulting in a total of $12n^3-20n^3$ operations in order to zero out non-diagonal matrix elements. Although the classical Jacobi algorithm performs badly compared with methods based on tridiagonalization, it is easy to parallelize.

The slow convergence is related to the fact that when a new rotation is performed, matrix elements which were previously zero, may change to non-zero values in the next rotation. To see this, consider the following simple example.

We specialize to a symmetric 3×3 matrix $\mathbf{A}$. We start the process as follows (assuming that $a_{23}=a_{32}$ is the largest non-diagonal matrix element) with $c=\cos\theta$ and $s=\sin\theta$

$$\mathbf{B}=\left(\begin{array}{ccc} 1 & 0 & 0\\ 0 & c & -s\\ 0 & s & c\end{array}\right)\left(\begin{array}{ccc} a_{11} & a_{12} & a_{13}\\ a_{21} & a_{22} & a_{23}\\ a_{31} & a_{32} & a_{33}\end{array}\right)\left(\begin{array}{ccc} 1 & 0 & 0\\ 0 & c & s\\ 0 & -s & c\end{array}\right).$$

We will choose the angle θ in order to have $b_{23}=b_{32}=0$. We get the new symmetric matrix

$$\mathbf{B}=\left(\begin{array}{ccc} a_{11} & a_{12}c-a_{13}s & a_{12}s+a_{13}c\\ a_{12}c-a_{13}s & a_{22}c^2+a_{33}s^2-2a_{23}sc & (a_{22}-a_{33})sc+a_{23}(c^2-s^2)\\ a_{12}s+a_{13}c & (a_{22}-a_{33})sc+a_{23}(c^2-s^2) & a_{22}s^2+a_{33}c^2+2a_{23}sc\end{array}\right).$$

Note that a_{11} is unchanged! As it should.

We have then

$$\begin{array}{rcl} b_{11} & = & a_{11}\\ b_{12} & = & a_{12}cos\theta-a_{13}sin\theta, 1\neq 2, 1\neq 3\\ b_{13} & = & a_{13}cos\theta+a_{12}sin\theta, 1\neq 2, 1\neq 3\\ b_{22} & = & a_{22}cos^2\theta-2a_{23}cos\theta sin\theta+a_{33}sin^2\theta\\ b_{33} & = & a_{33}cos^2\theta+2a_{23}cos\theta sin\theta+a_{22}sin^2\theta\\ b_{23} & = & (a_{22}-a_{33})cos\theta sin\theta+a_{23}(cos^2\theta-sin^2\theta)\end{array}$$

We will fix the angle θ so that $b_{23}=0$.

We get then a new matrix

$$\mathbf{B}=\left(\begin{array}{ccc} b_{11} & b_{12} & b_{13}\\ b_{12} & b_{22} & 0\\ b_{13} & 0 & a_{33}\end{array}\right).$$

We repeat assuming that b_{12} is the largest non-diagonal matrix element and get a new matrix

$$\mathbf{C}=\left(\begin{array}{ccc} c & -s & 0\\ s & c & 0\\ 0 & 0 & 1\end{array}\right)\left(\begin{array}{ccc} b_{11} & b_{12} & b_{13}\\ b_{12} & b_{22} & 0\\ b_{13} & 0 & b_{33}\end{array}\right)\left(\begin{array}{ccc} c & s & 0\\ -s & c & 0\\ 0 & 0 & 1\end{array}\right).$$

We continue this process till all non-diagonal matrix elements are zero. It is easy to convince oneself that when performing the above operations, the matrix element b_{23} which was previously set to zero may become different from zero. This is one of the problems which slows down the Jacobi procedure. We leave this experience to the reader in form of a large numerical project at the end of this chapter.

An implementation of the above algorithm, normally referred to as the classical Jacobi algorithm, is exposed partially in the code here.

http://folk.uio.no/compphys/programs/chapter07/cpp/jacobi.cpp

```
/*
   Jacobi's method for finding eigenvalues
   eigenvectors of the symetric matrix A.

   The eigenvalues of A will be on the diagonal
   of A, with eigenvalue i being A[i][i].
   The j-th component of the i-th eigenvector
   is stored in R[i][j].

   A: input matrix (n x n)
   R: empty matrix for eigenvectors (n x n)
   n: dimention of matrices
*/
#include <iostream>
#include <cmath>
#include "jacobi.h"

void jacobi_method ( double ** A, double ** R, int n )
{
// Setting up the eigenvector matrix
  for ( int i = 0; i < n; i++ ) {
    for ( int j = 0; j < n; j++ ) {
      if ( i == j ) {
  R[i][j] = 1.0;
      } else {
  R[i][j] = 0.0;
      }
    }
  }

  int k, l;
  double epsilon = 1.0e-8;
  double max_number_terations = (double) n * (double) n * (double) n;
  int iterations = 0;
  double max_offdiag = maxoffdiag ( A, &k, &l, n );

  while ( fabs(max_offdiag) > epsilon && (double) iterations < max_number_iterations
     ) {
    max:offdiag = maxoffdiag ( A, &k, &l, n );
    rotate ( A, R, k, l, n );
    iterations++;
  }
  std::cout << "Number of iterations: " << iterations << "\n";
  return;
}
// Function to find the maximum matrix element. Can you figure out a more
// elegant algorithm?
double maxoffdiag ( double ** A, int * k, int * l, int n )
```

```
{
  double max = 0.0;

  for ( int i = 0; i < n; i++ ) {
    for ( int j = i + 1; j < n; j++ ) {
      if ( fabs(A[i][j]) > max ) {
  max = fabs(A[i][j]);
  *l = i;
  *k = j;
      }
    }
  }
  return max;
}
// Function to find the values of cos and sin
void rotate ( double ** A, double ** R, int k, int l, int n )
{
  double s, c;
  if ( A[k][l] != 0.0 ) {
    double t, tau;
    tau = (A[l][l] - A[k][k])/(2*A[k][l]);
    if ( tau > 0 ) {
      t = 1.0/(tau + sqrt(1.0 + tau*tau);
    } else {
      t = -1.0/( -tau + sqrt(1.0 + tau*tau);
    }

    c = 1/sqrt(1+t*t);
    s = c*t;
  } else {
    c = 1.0;
    s = 0.0;
  }
  double a_kk, a_ll, a_ik, a_il, r_ik, r_il;
  a_kk = A[k][k];
  a_ll = A[l][l];
  // changing the matrix elements with indices k and l
  A[k][k] = c*c*a_kk - 2.0*c*s*A[k][l] + s*s*a_ll;
  A[l][l] = s*s*a_kk + 2.0*c*s*A[k][l] + c*c*a_ll;
  A[k][l] = 0.0; // hard-coding of the zeros
  A[l][k] = 0.0;
  // and then we change the remaining elements
  for ( int i = 0; i < n; i++ ) {
    if ( i != k && i != l ) {
      a_ik = A[i][k];
      a_il = A[i][l];
      A[i][k] = c*a_ik - s*a_il;
      A[k][i] = A[i][k];
      A[i][l] = c*a_il + s*a_ik;
      A[l][i] = A[i][l];
    }
    // Finally, we compute the new eigenvectors
    r_ik = R[i][k];
    r_il = R[i][l];
    R[i][k] = c*r_ik - s*r_il;
    R[i][l] = c*r_il + s*r_ik;
  }
  return;
}
```

7.5 Similarity Transformations with Householder's method

In this case the diagonalization is performed in two steps: First, the matrix is transformed into tridiagonal form by the Householder similarity transformation. Secondly, the tridiagonal matrix is then diagonalized. The reason for this two-step process is that diagonalizing a tridiagonal matrix is computational much faster than the corresponding diagonalization of a general symmetric matrix. Let us discuss the two steps in more detail.

7.5.1 The Householder's method for tridiagonalization

The first step consists in finding an orthogonal matrix $\mathbf{S}$ which is the product of $(n-2)$ orthogonal matrices

$$\mathbf{S} = \mathbf{S}_1\mathbf{S}_2\dots\mathbf{S}_{n-2},$$

each of which successively transforms one row and one column of $\mathbf{A}$ into the required tridiagonal form. Only $n-2$ transformations are required, since the last two elements are already in tridiagonal form. In order to determine each $\mathbf{S_i}$ let us see what happens after the first multiplication, namely,

$$\mathbf{S}_1^T\mathbf{A}\mathbf{S}_1 = \begin{pmatrix} a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & a'_{22} & a'_{23} & \dots & \dots & \dots & a'_{2n} \\ 0 & a'_{32} & a'_{33} & \dots & \dots & \dots & a'_{3n} \\ 0 & \dots & \dots & \dots & \dots & \dots & \\ 0 & a'_{n2} & a'_{n3} & \dots & \dots & \dots & a'_{nn} \end{pmatrix}$$

where the primed quantities represent a matrix $\mathbf{A}'$ of dimension $n-1$ which will subsequentely be transformed by $\mathbf{S_2}$. The factor e_1 is a possibly non-vanishing element. The next transformation produced by $\mathbf{S_2}$ has the same effect as $\mathbf{S_1}$ but now on the submatirx $\mathbf{A}^{'}$ only

$$(\mathbf{S}_1\mathbf{S}_2)^T\mathbf{A}\mathbf{S}_1\mathbf{S}_2 = \begin{pmatrix} a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & a'_{22} & e_2 & 0 & \dots & \dots & 0 \\ 0 & e_2 & a''_{33} & \dots & \dots & \dots & a''_{3n} \\ 0 & \dots & \dots & \dots & \dots & \dots & \\ 0 & 0 & a''_{n3} & \dots & \dots & \dots & a''_{nn} \end{pmatrix}$$

Note that the effective size of the matrix on which we apply the transformation reduces for every new step. In the previous Jacobi method each similarity transformation is in principle performed on the full size of the original matrix.

After a series of such transformations, we end with a set of diagonal matrix elements

$$a_{11}, a'_{22}, a''_{33} \dots a_{nn}^{n-1},$$

and off-diagonal matrix elements

$$e_1, e_2, e_3, \dots, e_{n-1}.$$

The resulting matrix reads

$$\mathbf{S}^T\mathbf{A}\mathbf{S} = \begin{pmatrix} a_{11} & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & a'_{22} & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & a''_{33} & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & a_{n-1n-1}^{(n-1)} & e_{n-1} \\ 0 & \dots & \dots & \dots & \dots & e_{n-1} & a_{nn}^{(n-1)} \end{pmatrix}.$$

It remains to find a recipe for determining the transformation $\mathbf{S}_n$. We illustrate the method for $\mathbf{S}_1$ which we assume takes the form

$$\mathbf{S_1} = \begin{pmatrix} 1 & \mathbf{0^T} \\ \mathbf{0} & \mathbf{P} \end{pmatrix},$$

with $\mathbf{0^T}$ being a zero row vector, $\mathbf{0^T} = \{0, 0, \cdots\}$ of dimension $(n-1)$. The matrix $\mathbf{P}$ is symmetric with dimension $((n-1)\times(n-1))$ satisfying $\mathbf{P}^2 = \mathbf{I}$ and $\mathbf{P}^T = \mathbf{P}$. A possible choice which fulfills the latter two requirements is

$$\mathbf{P} = \mathbf{I} - 2\mathbf{u}\mathbf{u}^T,$$

where $\mathbf{I}$ is the $(n-1)$ unity matrix and $\mathbf{u}$ is an $n-1$ column vector with norm $\mathbf{u}^T\mathbf{u} = 1$, that is its inner product.

Note that $\mathbf{u}\mathbf{u}^T$ is an outer product giving a dimension $((n-1)\times(n-1))$. Each matrix element of $\mathbf{P}$ then reads

$$P_{ij} = \delta_{ij} - 2u_iu_j,$$

where i and j range from 1 to $n-1$. Applying the transformation $\mathbf{S}_1$ results in

$$\mathbf{S}_1^T\mathbf{A}\mathbf{S}_1 = \begin{pmatrix} a_{11} & (\mathbf{Pv})^T \\ \mathbf{Pv} & \mathbf{A}' \end{pmatrix},$$

where $\mathbf{v^T} = \{a_{21}, a_{31}, \cdots, a_{n1}\}$ and $\mathbf{P}$ must satisfy $(\mathbf{Pv})^T = \{k, 0, 0, \cdots\}$. Then **Pv = v** - 2**u**($\mathbf{u}^T\mathbf{v}) = k\mathbf{e}, with \mathbf{e}^T = \{1, 0, 0, \ldots 0\}$**. Solving the latter equation gives us u and thus the needed transformation P. We do first however need to compute the scalar** k **by taking the scalar product of the last equation with its transpose and using the fact that** $\mathbf{P}^2 = \mathbf{I}$**. We get then**

$$(\mathbf{Pv})^T\mathbf{Pv} = k^2 = \mathbf{v}^T\mathbf{v} = |\mathbf{v}|^2 = \sum_{i=2}^{n} a_{i1}^2,$$

which determines the constant $k = \pm v$**. Now we can rewrite Eq. (7.5.1) as**

$$\mathbf{v} - k\mathbf{e} = 2\mathbf{u}(\mathbf{u}^T\mathbf{v}),$$

and taking the scalar product of this equation with itself and obtain 2($\mathbf{u}^T\mathbf{v})^2 = (v^2 \pm a_{21}v), which finally determines \mathbf{u} = \frac{\mathbf{v}-k\mathbf{e}}{2(\mathbf{u}^T\mathbf{v})}$.In solving Eq. (7.5.1) great care has to be exercised so as to choose those values which make the right-hand largest in order to avoid loss of numerical precision. The above steps are then repeated for every transformations till we have a tridiagonal matrix suitable for obtaining the eigenvalues. It is not so difficult to implement Householder's algorithm, as demonstrated by the following code.

http://folk.uio.no/compphys/programs/chapter07/cpp/householder.cpp

```
  /*
  ** The function
  **           householder()
  ** perform a Housholder reduction of a real symmetric matrix
  ** a[][]. On output a[][] is replaced by the orthogonal matrix
  ** effecting the transformation. d[] returns the diagonal elements
  ** of the tri-diagonal matrix, and e[] the off-diagonal elements,
  ** with e[0] = 0.
  */
void householder(double **a, int n, double *d, double *e)
{
  register int l,k,j,i;
  double      scale,hh,h,g,f;

  for(i = n - 1; i > 0; i--) {
    l = i-1;
    h = scale= 0.0;
    if(l > 0) {
      for(k = 0; k <= l; k++)
```

```
          scale += fabs(a[i][k]);
          if(scale == 0.0)       // skip transformation
            e[i] = a[i][l];
          else {
          for(k = 0; k <= l; k++) {
            a[i][k] /= scale;   // used scaled a's for transformation
            h     += a[i][k]*a[i][k];
          }
          f     = a[i][l];
          g     = (f >= 0.0 ? -sqrt(h) : sqrt(h));
          e[i]  = scale*g;
          h    -= f * g;
          a[i][l] = f - g;
          f     = 0.0;

          for(j = 0;j <= l;j++) {
            a[j][i] = a[i][j]/h; // can be omitted if eigenvector not wanted
            g     = 0.0;
            for(k = 0; k <= j; k++) {
              g += a[j][k]*a[i][k];
            }
            for(k = j+1; k <= l; k++)
              g += a[k][j]*a[i][k];
            e[j]=g/h;
            f += e[j]*a[i][j];
          }
          hh=f/(h+h);
          for(j = 0; j <= l;j++) {
            f = a[i][j];
            e[j]=g=e[j]-hh*f;
            for(k = 0; k <= j; k++)
              a[j][k] -= (f*e[k]+g*a[i][k]);
          }
        } // end k-loop
      } // end if-loop for l > 1
      else {
        e[i]=a[i][l];
      }
      d[i]=h;
    } // end i-loop
    d[0] = 0.0;
    e[0] = 0.0;

        /* Contents of this loop can be omitted if eigenvectors not
    ** wanted except for statement d[i]=a[i][i];
        */

    for(i = 0; i < n; i++) {
      l = i-1;
      if(d[i]) {
        for(j = 0; j <= l; j++) {
          g= 0.0;
          for(k = 0; k <= l; k++) {
             g += a[i][k] * a[k][j];
          }
          for (k = 0; k <= l; k++) {
             a[k][j] -= g * a[k][i];
          }
        }
      }
      d[i]  = a[i][i];
      a[i][i] = 1.0;
      for(j = 0; j <= l; j++) {
        a[j][i]=a[i][j] = 0.0;
      }
    }
  } // End: function householder()
```

7.5.2 Diagonalization of a Tridiagonal Matrix via Francis' Algorithm

The matrix is now transformed into tridiagonal form and the last step is to transform it into a diagonal matrix giving the eigenvalues on the diagonal[1].

Before we discuss the algorithms, we note that the eigenvalues of a tridiagonal matrix can be obtained using the characteristic polynomial

$$P(\lambda) = det(\lambda\mathbf{I}-\mathbf{A}) = \prod_{i=1}^{n}(\lambda_i-\lambda),$$

with the matrix

$$\mathbf{A}-\lambda\mathbf{I} = \left(\begin{array}{cccccccc} & d_1-\lambda & e_1 & 0 & 0 & \dots & 0 & 0 \\ & e_1 & d_2-\lambda & e_2 & 0 & \dots & 0 & 0 \\ det & 0 & e_2 & d_3-\lambda & e_3 & 0 & \dots & 0 \\ & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ & 0 & \dots & \dots & \dots & \dots & d_{N_{\mathrm{step}}-2}-\lambda & e_{N_{\mathrm{step}}-1} \\ & 0 & \dots & \dots & \dots & \dots & e_{N_{\mathrm{step}}-1} & d_{N_{\mathrm{step}}-1}-\lambda \end{array}\right)$$

We can solve this equation in a recursive manner. We let $P_k(\lambda)$ be the value of k subdeterminant of the above matrix of dimension $n\times n$. The polynomial $P_k(\lambda)$ is clearly a polynomial of degree k. Starting with $P_1(\lambda)$ we have $P_1(\lambda) = d_1-\lambda$. The

[1]This section is not complete it will be finished end of fall 2009.

next polynomial reads $P_2(\lambda) = (d_2 - \lambda)P_1(\lambda) - e_1^2$. By expanding the determinant for $P_k(\lambda)$ in terms of the minors of the nth column we arrive at the recursion relation

$$P_k(\lambda) = (d_k - \lambda)P_{k-1}(\lambda) - e_{k-1}^2 P_{k-2}(\lambda).$$

Together with the starting values $P_1(\lambda)$ and $P_2(\lambda)$ and good root searching methods we arrive at an efficient computational scheme for finding the roots of $P_n(\lambda)$. However, for large matrices this algorithm is rather inefficient and time-consuming.

The programs which performs these transformations are matrix $\mathbf{A} \longrightarrow$ tridiagonal matrix $\longrightarrow$ diagonal matrix

C:	void householder(double $**$a, int n, double d[], double e[])
	void francis(double d[], double[], int n, double $**$z)
Fortran:	CALL householder(a, n, d, e)
	CALL francis(d, e, n, z)

The last step through the function *francis()* involves several technical details. Let us describe the basic idea in terms of a four-dimensional example. For more details, see Ref. [1], in particular chapters seven and eight.

The current tridiagonal matrix takes the form

$$\mathbf{A} = \begin{pmatrix} d_1 & e_1 & 0 & 0 \\ e_1 & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e_3 \\ 0 & 0 & e_3 & d_4 \end{pmatrix}.$$

As a first observation, if any of the elements e_i are zero the matrix can be separated into smaller pieces before diagonalization. Specifically, if $e_1 = 0$ then d_1 is an eigenvalue. Thus, let us introduce a transformation $\mathbf{S_1}$

$$\mathbf{S_1} = \begin{pmatrix} \cos\theta & 0 & 0 & \sin\theta \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -\sin\theta & 0 & 0 & \cos\theta \end{pmatrix}$$

Then the similarity transformation

$$\mathbf{S_1^T A S_1} = \mathbf{A}' = \begin{pmatrix} d_1' & e_1' & 0 & 0 \\ e_1' & d_2 & e_2 & 0 \\ 0 & e_2 & d_3 & e'3 \\ 0 & 0 & e_3' & d_4' \end{pmatrix}$$

produces a matrix where the primed elements in $\mathbf{A}'$ have been changed by the transformation whereas the unprimed elements are unchanged. If we now choose θ to give the element $a_{21}^{'} = e^{'} = 0$ then we have the first eigenvalue $= a_{11}^{'} = d_1^{'}$.

This procedure can be continued on the remaining three-dimensional submatrix for the next eigenvalue. Thus after four transformations we have the wanted diagonal form.

7.6 Power Methods

We assume $\hat{A}$ can be diagonalized. Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the n eigenvalues (counted with multiplicity) of $\hat{A}$ and let $v_1, v_2, \ldots, v_n$ be the corresponding eigenvectors. We assume that λ_1 is the dominant eigenvalue, so that $|\lambda_1| > |\lambda_j|$ for $j > 1$.

The initial vector b_0 can be written:

$$b_0 = c_1 v_1 + c_2 v_2 + \cdots + c_m v_m.$$

If b_0 is chosen randomly (with uniform probability), then c_1 âĽă 0 with probability 1. Now,

$$\begin{aligned} A^k b_0 &= c_1 A^k v_1 + c_2 A^k v_2 + \cdots + c_m A^k v_m \\ &= c_1 \lambda_1^k v_1 + c_2 \lambda_2^k v_2 + \cdots + c_m \lambda_m^k v_m \\ &= c_1 \lambda_1^k \left(v_1 + \frac{c_2}{c_1} \left(\frac{\lambda_2}{\lambda_1} \right)^k v_2 + \cdots + \frac{c_m}{c_1} \left(\frac{\lambda_m}{\lambda_1} \right)^k v_m \right). \end{aligned}$$

The expression within parentheses converges to v_1 because $|\lambda_j/\lambda_1| < 1$ for $j > 1$. On the other hand, we have

$$b_k = \frac{A^k b_0}{\|A^k b_0\|}.$$

Therefore, b_k converges to (a multiple of) the eigenvector v_1. The convergence is geometric, with ratio

$$\left| \frac{\lambda_2}{\lambda_1} \right|,$$

where λ_2 denotes the second dominant eigenvalue. Thus, the method converges slowly if there is an eigenvalue close in magnitude to the dominant eigenvalue.

Under the assumptions:

- A has an eigenvalue that is strictly greater in magnitude than its other eigenvalues
- The starting vector b_0 has a nonzero component in the direction of an eigenvector associated with the dominant eigenvalue.

then:

- A subsequence of (b_k) converges to an eigenvector associated with the dominant eigenvalue

Note that the sequence (b_k) does not necessarily converge. It can be shown that $b_k = e^{i\phi_k} v_1 + r_k$ where: v_1 is an eigenvector associated with the dominant eigenvalue, and $\|r_k\| \to 0$. The presence of the term $e^{i\phi_k}$ implies that (b_k) does not converge unless $e^{i\phi_k} = 1$. Under the two assumptions listed above, the sequence (μ_k) defined by $\mu_k = \frac{b_k^* A b_k}{b_k^* b_k}$ converges to the dominant eigenvalue.

Power iteration is not used very much because it can find only the dominant eigenvalue.

The algorithm is however very useful in some specific case. For instance, Google uses it to calculate the page rank of documents in their search engine. For matrices that are well-conditioned and as sparse as the web matrix, the power iteration method can be more efficient than other methods of finding the dominant eigenvector.

Some of the more advanced eigenvalue algorithms can be understood as variations of the power iteration. For instance, the inverse iteration method applies power iteration to the matrix $\hat{A}^{-1}$. Other algorithms look at the whole subspace generated by the vectors b_k. This subspace is known as the Krylov subspace. It can be computed by Arnoldi iteration or Lanczos iteration. The latter is method of choice for diagonalizing symmetric matrices with huge dimensionalities. We discuss the Lanczos algorithm in the next section.

7.7 Iterative methods: Lanczos' algorithm

The Lanczos algorithm is applied to symmetric eigenvalue problems. The basic features with a real symmetric matrix (and normally huge $n > 10^6$ and sparse) $\hat{A}$ of dimension $n \times n$ are

- The Lanczos' algorithm generates a sequence of real tridiagonal matrices T_k of dimension $k \times k$ with $k \leq n$, with the property that the extremal eigenvalues of T_k are progressively better estimates of $\hat{A}$' extremal eigenvalues.
- The method converges to the extremal eigenvalues.
- The similarity transformation is

$$\hat{T} = \hat{Q}^T \hat{A} \hat{Q},$$

  with the first vector $\hat{Q}\hat{e}_1 = \hat{q}_1$.

We are going to solve iteratively

$$\hat{T} = \hat{Q}^T \hat{A} \hat{Q},$$

with the first vector $\hat{Q}\hat{e}_1 = \hat{q}_1$. We can then write out the matrix $\hat{Q}$ in terms of its column vectors

$$\hat{Q} = [\hat{q}_1 \hat{q}_2 \ldots \hat{q}_n].$$

The matrix

$$\hat{T} = \hat{Q}^T \hat{A} \hat{Q},$$

can be written as

$$\hat{T} = \begin{pmatrix} \alpha_1 & \beta_1 & 0 & \dots & \dots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & 0 & \dots & 0 \\ 0 & \beta_2 & \alpha_3 & \beta_3 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & 0 \\ \dots & & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ 0 & \dots & \dots & 0 & \beta_{n-1} & \alpha_n \end{pmatrix}$$

Using the fact that $\hat{Q}\hat{Q}^T = \hat{I}$, we can rewrite

$$\hat{T} = \hat{Q}^T \hat{A} \hat{Q},$$

as

$$\hat{Q}\hat{T} = \hat{A}\hat{Q},$$

and if we equate columns (recall from the previous slide)

$$\hat{T} = \begin{pmatrix} \alpha_1 & \beta_1 & 0 & \dots & \dots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & 0 & \dots & 0 \\ 0 & \beta_2 & \alpha_3 & \beta_3 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & 0 \\ \dots & & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ 0 & \dots & \dots & 0 & \beta_{n-1} & \alpha_n \end{pmatrix}$$

we obtain

$$\hat{A}\hat{q}_k = \beta_{k-1}\hat{q}_{k-1} + \alpha_k\hat{q}_k + \beta_k\hat{q}_{k+1}.$$

We have thus

$$\hat{A}\hat{q}_k = \beta_{k-1}\hat{q}_{k-1} + \alpha_k\hat{q}_k + \beta_k\hat{q}_{k+1},$$

with $\beta_0\hat{q}_0 = 0$ for $k = 1 : n-1$. Remember that the vectors $\hat{q}_k$ are orthornormal and this implies

$$\alpha_k = \hat{q}_k^T \hat{A}\hat{q}_k,$$

and these vectors are called Lanczos vectors. We have thus

$$\hat{A}\hat{q}_k = \beta_{k-1}\hat{q}_{k-1} + \alpha_k\hat{q}_k + \beta_k\hat{q}_{k+1},$$

with $\beta_0\hat{q}_0 = 0$ for $k = 1 : n-1$ and

$$\alpha_k = \hat{q}_k^T \hat{A}\hat{q}_k.$$

If

$$\hat{r}_k = (\hat{A} - \alpha_k\hat{I})\hat{q}_k - \beta_{k-1}\hat{q}_{k-1},$$

is non-zero, then

$$\hat{q}_{k+1} = \hat{r}_k/\beta_k,$$

with $\beta_k = \pm||\hat{r}_k||_2$. These steps can then be written in terms of the following simple algorithm:

```
r_0 = q_1; beta_0=1; q_0=0; int k = 0;
while (beta_k != 0)
   q_{k+1} = r_k/beta_k
   k = k+1
   alpha_k = q_k^T A q_k
   r_k = (A-alpha_k I) q_k -beta_{k-1}q_{k-1}
   beta_k = || r_k||_2
end while
```

7.8 Schrödinger's Equation Through Diagonalization

Instead of solving the Schrödinger equation as a differential equation, we will solve it through diagonalization of a large matrix. However, in both cases we need to deal with a problem with boundary conditions, viz., the wave function goes to zero at the endpoints.

To solve the Schrödinger equation as a matrix diagonalization problem, let us study the radial part of the Schrödinger equation. The radial part of the wave function, $R(r)$, is a solution to

$$-\frac{\hbar^2}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr}-\frac{l(l+1)}{r^2}\right)R(r)+V(r)R(r)=ER(r).$$

Then we substitute $R(r)=(1/r)u(r)$ and obtain

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r)+\left(V(r)+\frac{l(l+1)}{r^2}\frac{\hbar^2}{2m}\right)u(r)=Eu(r).$$

We introduce a dimensionless variable $\rho=(1/\alpha)r$ where α is a constant with dimension length and get

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(r)+\left(V(\rho)+\frac{l(l+1)}{\rho^2}\frac{\hbar^2}{2m\alpha^2}\right)u(\rho)=Eu(\rho).$$

In the example below, we will replace the latter equation with that for the one-dimensional harmonic oscillator. Note however that the procedure which we give below applies equally well to the case of e.g., the hydrogen atom. We replace ρ with x, take away the centrifugal barrier term and set the potential equal to

$$V(x)=\frac{1}{2}kx^2,$$

with k being a constant. In our solution we will use units so that $k=\hbar=m=\alpha=1$ and the Schrödinger equation for the one-dimensional harmonic oscillator becomes

$$-\frac{d^2}{dx^2}u(x)+x^2u(x)=2Eu(x).$$

Let us now see how we can rewrite this equation as a matrix eigenvalue problem. First we need to compute the second derivative. We use here the following expression for the second derivative of a function f

$$f'' = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2),$$

where h is our step. Next we define minimum and maximum values for the variable x, R_{min} and R_{max}, respectively. With a given number of steps, N_{step}, we then define the step h as

$$h = \frac{R_{\mathrm{max}} - R_{\mathrm{min}}}{N_{\mathrm{step}}}.$$

If we now define an arbitrary value of x as

$$x_i = R_{\mathrm{min}} + ih \qquad i = 1, 2, \dots, N_{\mathrm{step}} - 1$$

we can rewrite the Schrödinger equation for x_i as

$$-\frac{u(x_k + h) - 2u(x_k) + u(x_k - h)}{h^2} + x_k^2 u(x_k) = 2Eu(x_k),$$

or in a more compact way

$$-\frac{u_{k+1} - 2u_k + u_{k-1}}{h^2} + x_k^2 u_k = -\frac{u_{k+1} - 2u_k + u_{k-1}}{h^2} + V_k u_k = 2Eu_k,$$

where $u_k = u(x_k)$, $u_{k\pm 1} = u(x_k \pm h)$ and $V_k = x_k^2$, the given potential. Let us see how this recipe may lead to a matrix reformulation of the Schrödinger equation. Define first the diagonal matrix element

$$d_k = \frac{2}{h^2} + V_k,$$

and the non-diagonal matrix element

$$e_k = -\frac{1}{h^2}.$$

In this case the non-diagonal matrix elements are given by a mere constant. *All non-diagonal matrix elements are equal.* With these definitions the Schrödinger equation takes the following form

$$d_k u_k + e_{k-1} u_{k-1} + e_{k+1} u_{k+1} = 2Eu_k,$$

where u_k is unknown. Since we have $N_{\mathrm{step}} - 1$ values of k we can write the latter equation as a matrix eigenvalue problem

$$\begin{pmatrix} d_1 & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & d_2 & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & d_3 & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & d_{N_{\mathrm{step}}-2} & e_{N_{\mathrm{step}}-1} \\ 0 & \dots & \dots & \dots & \dots & e_{N_{\mathrm{step}}-1} & d_{N_{\mathrm{step}}-1} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{N_{\mathrm{step}}-1} \end{pmatrix} = 2E \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{N_{\mathrm{step}}-1} \end{pmatrix} \tag{7.4}$$

or if we wish to be more detailed, we can write the tridiagonal matrix as

$$\left(\begin{array}{ccccccc} \frac{2}{h^2}+V_1 & -\frac{1}{h^2} & 0 & 0 & \dots & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2}+V_2 & -\frac{1}{h^2} & 0 & \dots & 0 & 0 \\ 0 & -\frac{1}{h^2} & \frac{2}{h^2}+V_3 & -\frac{1}{h^2} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & \frac{2}{h^2}+V_{N_{\text{step}}-2} & -\frac{1}{h^2} \\ 0 & \dots & \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2}+V_{N_{\text{step}}-1} \end{array}\right) \tag{7.5}$$

This is a matrix problem with a tridiagonal matrix of dimension $N_{\text{step}}-1\times N_{\text{step}}-1$ and will thus yield $N_{\text{step}}-1$ eigenvalues. It is important to notice that we do not set up a matrix of dimension $N_{\text{step}}\times N_{\text{step}}$ since we can fix the value of the wave function at $k=N_{\text{step}}$. Similarly, we know the wave function at the other end point, that is for x_0.

The above equation represents an alternative to the numerical solution of the differential equation for the Schrödinger equation discussed in chapter **??**.

The eigenvalues of the harmonic oscillator in one dimension are well known. In our case, with all constants set equal to 1, we have

$$E_n = n+\frac{1}{2},$$

with the ground state being $E_0 = 1/2$. Note however that we have rewritten the Schrödinger equation so that a constant 2 stands in front of the energy. Our program will then yield twice the value, that is we will obtain the eigenvalues $1,3,5,7......$

In the next subsection we will try to delineate how to solve the above equation.

7.8.1 Numerical solution of the Schrödinger equation by diagonalization

The algorithm for solving Eq. (7.4) may take the following form

- Define values for N_{step}, R_{min} and R_{max}. These values define in turn the step size h. Typical values for R_{max} and R_{min} could be 10 and -10 respectively for the lowest-lying states. The number of mesh points N_{step} could be in the range 100 to some thousands. You can check the stability of the results as functions of $N_{\text{step}}-1$ and R_{max} and R_{min} against the exact solutions.

- Construct then two one-dimensional arrays which contain all values of x_k and the potential V_k. For the latter it can be convenient to write a small function which sets up the potential as function of x_k. For the three-dimensional case you may also need to include the centrifugal potential. The dimension of these two arrays should go from 0 up to N_{step}.

- Construct thereafter the one-dimensional vectors d and e, where d stands for the diagonal matrix elements and e the non-diagonal ones. Note that the dimension of these two arrays runs from 1 up to $N_{\text{step}} - 1$, 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 $n = N_{\text{step}} - 1$ and a matrix z of dimension $N_{\text{step}} - 1 \times N_{\text{step}} - 1$ which returns the eigenfunctions. On return, the array d contains the eigenvalues. If z is given as the unity matrix on input, it returns the eigenvectors. For a given eigenvalue k, the eigenvector is given by the column k in z, that is z[][k] in C, or z(:,k) in Fortran.

- TQLI does however not return an ordered sequence of eigenvalues. You may then need to sort them as e.g., an ascending series of numbers. The program we provide includes a sorting function as well.

- Finally, you may perhaps need to plot the eigenfunctions as well, or calculate some other expectation values. Or, you would like to compare the eigenfunctions with the analytical answers for the harmonic oscillator or the hydrogen atom. We provide a function *plot* which has as input one eigenvalue chosen from the output of *tqli*. This function gives you a normalized wave function u where the norm is calculated as

$$\int_{R_{\text{min}}}^{R_{\text{max}}} |u(x)|^2 dx \rightarrow h \sum_{i=0}^{N_{\text{step}}} u_i^2 = 1,$$

  and we have used the trapezoidal rule for integration discussed in chapter 5.

7.8.2 Program example and results for the one-dimensional harmonic oscillator

We present here a program example which encodes the above algorithm.

http://folk.uio.no/compphys/programs/chapter07/cpp/program1.cpp

```
/*
  Solves the one-particle Schrodinger equation
  for a potential specified in function
  potential(). This example is for the harmonic oscillator
*/
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
```

```
#include "lib.h"
using namespace std;
// output file as global variable
ofstream ofile;

// function declarations

void initialise(double&, double&, int&, int&) ;
double potential(double);
int comp(const double *, const double *);
void output(double, double, int, double *);

int main(int argc, char* argv[])
{
  int     i, j, max_step, orb_l;
  double  r_min, r_max, step, const_1, const_2, orb_factor,
          *e, *d, *w, *r, **z;
  char *outfilename;
  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  //  Read in data
  initialise(r_min, r_max, orb_l, max_step);
  // initialise constants
  step  = (r_max - r_min) / max_step;
  const_2 = -1.0 / (step * step);
  const_1 = - 2.0 * const_2;
  orb_factor = orb_l * (orb_l + 1);

  // local memory for r and the potential w[r]
  r = new double[max_step + 1];
  w = new double[max_step + 1];
  for(i = 0; i <= max_step; i++) {
    r[i] = r_min + i * step;
    w[i] = potential(r[i]) + orb_factor / (r[i] * r[i]);
  }
  // local memory for the diagonalization process
  d = new double[max_step]; // diagonal elements
  e = new double[max_step]; // tridiagonal off-diagonal elements
  z = (double **) matrix(max_step, max_step, sizeof(double));
  for(i = 0; i < max_step; i++) {
    d[i]  = const_1 + w[i + 1];
    e[i]  = const_2;
    z[i][i] = 1.0;
    for(j = i + 1; j < max_step; j++) {
      z[i][j] = 0.0;
    }
  }
  // diagonalize and obtain eigenvalues
  tqli(d, e, max_step - 1, z);
  // Sort eigenvalues as an ascending series
  qsort(d,(UL) max_step - 1,sizeof(double),
       (int(*)(const void *,const void *))comp);
  // send results to ouput file
  output(r_min , r_max, max_step, d);
  delete [] r; delete [] w; delete [] e; delete [] d;
  free_matrix((void **) z); // free memory
  ofile.close(); // close output file
  return 0;
} // End: function main()

/*
  The function potential()
  calculates and return the value of the
  potential for a given argument x.
  The potential here is for the 1-dim harmonic oscillator
*/

double potential(double x)
{
   return x*x;

} // End: function potential()

/*
  The function int comp()
  is a utility function for the library function qsort()
  to sort double numbers after increasing values.
*/

int comp(const double *val_1, const double *val_2)
{
  if((*val_1) <= (*val_2)) return -1;
  else if((*val_1) > (*val_2)) return +1;
  else                 return 0;
} // End: function comp()

// read in min and max radius, number of mesh points and l
void initialise(double& r_min, double& r_max, int& orb_l, int& max_step)
{
  cout << "Min vakues of R = ";
  cin >> r_min;
  cout << "Max value of R = ";
  cin >> r_max;
  cout << "Orbital momentum = ";
  cin >> orb_l;
  cout << "Number of steps = ";
  cin >> max_step;
} // end of function initialise
// output of results
void output(double r_min , double r_max, int max_step, double *d)
{
  int i;
  ofile << "RESULTS:" << endl;
  ofile << setiosflags(ios::showpoint | ios::uppercase);
  ofile <<"R_min = " << setw(15) << setprecision(8) << r_min << endl;
  ofile <<"R_max = " << setw(15) << setprecision(8) << r_max << endl;
  ofile <<"Number of steps = " << setw(15) << max_step << endl;
  ofile << "Five lowest eigenvalues:" << endl;
  for(i = 0; i < 5; i++) {
    ofile << setw(15) << setprecision(8) << d[i] << endl;
  }
} // end of function output
```

There are several features to be noted in this program.

The main program calls the function *initialise*, which reads in the minimum and maximum values of r, the number of steps and the orbital angular momentum l. Thereafter we allocate place for the vectors containing r and the potential, given by the variables $r[i]$ and $w[i]$, respectively. We also set up the vectors $d[i]$ and $e[i]$ containing the diagonal and non-diagonal matrix elements. Calling the function $tqli$ we obtain in turn the unsorted eigenvalues. The latter are sorted by the intrinsic C-function $qsort$.

The calculaton of the wave function for the lowest eigenvalue is done in the function $plot$, while all output of the calculations is directed to the fuction $output$.

The included table exhibits the precision achieved as function of the number of mesh points N. The exact values are $1,3,5,7,9$.

The agreement with the exact solution improves with increasing numbers of mesh points. However, the agreement for the excited states is by no means impressive. Moreover, as the dimensionality increases, the time consumption increases dramatically. Matrix diagonalization scales typically as $\approx N^3$. In addition, there is a maximum size of a matrix which can be stored in RAM.

The obvious question which then arises is whether this scheme is nothing but a mere example of matrix diagonalization, with few practical applications of interest. In chapter 3, where we dealt with interpolation and extrapolation, we discussed also called Richardson's deferred extrapolation scheme. Applied to this particualr case, the philosophy of this scheme would be to diagonalize the above matrix for a set of

Table 7.1: Five lowest eigenvalues as functions of the number of mesh points N with $r_{\min} = -10$ and $r_{\max} = 10$.

N	E_0	E_1	E_2	E_3	E_4
50	9.898985E-01	2.949052E+00	4.866223E+00	6.739916E+00	8.568442E+00
100	9.974893E-01	2.987442E+00	4.967277E+00	6.936913E+00	8.896282E+00
200	9.993715E-01	2.996864E+00	4.991877E+00	6.984335E+00	8.974301E+00
400	9.998464E-01	2.999219E+00	4.997976E+00	6.996094E+00	8.993599E+00
1000	1.000053E+00	2.999917E+00	4.999723E+00	6.999353E+00	8.999016E+00

values of N and thereby the step length h. Thereafter, an extrapolation is made to $h \to 0$. The obtained eigenvalues agree then with a remarkable precision with the exact solution. The algorithm is then as follows

- Perform a series of diagonalizations of the matrix in Eq. (7.5) for different values of the step size h. We obtain then a series of eigenvalues $E(h/2^k)$ with $k = 0, 1, 2, \dots$. That will give us an array of 'x-values' $h, h/2, h/4, \dots$ and an array of 'y-values' $E(h), E(h/2), E(h/4), \dots$. Note that you will have such a set for each eigenvalue.

- Use these values to perform an extrapolation calling e.g., the function POLINT with the point where we wish to extrapolate to given by $h = 0$.

- End the iteration over k when the error returned by POLINT is smaller than a fixed test.

The results for the 10 lowest-lying eigenstates for the one-dimensional harmonic oscillator are listed below after just 3 iterations, i.e., the step size has been reduced to $h/8$ only. The exact results are $1, 3, 5, \dots, 19$ and we see that the agreement is just excellent for the extrapolated results. The results after diagonalization differ already at the fourth-fifth digit.

Parts of a Fortran program which includes Richardson's extrapolation scheme is included here. It performs five diagonalizations and establishes results for various step lengths and interpolates using the function `POLINT`.

```
! start loop over interpolations, here we set max interpolations to 5
    DO interpol=1, 5
      IF ( interpol == 1) THEN
```

Table 7.2: Result for numerically calculated eigenvalues of the one-dimensional harmonic oscillator after three iterations starting with a matrix of size 100×100 and ending with a matrix of dimension 800×800. These four values are then used to extrapolate the 10 lowest-lying eigenvalues to $h = 0.$. The values of x span from -10 to 10, that means that the starting step was $h = 20/100 = 0.2$. We list here only the results after three iterations. The error test was set equal 10^{-6}.

Extrapolation	Diagonalization	Error
0.100000D+01	0.999931D+00	0.206825D-10
0.300000D+01	0.299965D+01	0.312617D-09
0.500000D+01	0.499910D+01	0.174602D-08
0.700000D+01	0.699826D+01	0.605671D-08
0.900000D+01	0.899715D+01	0.159170D-07
0.110000D+02	0.109958D+02	0.349902D-07
0.130000D+02	0.129941D+02	0.679884D-07
0.150000D+02	0.149921D+02	0.120735D-06
0.170000D+02	0.169899D+02	0.200229D-06
0.190000D+02	0.189874D+02	0.314718D-06

```
          max_step=start_step
        ELSE
          max_step=(interpol-1)*2*start_step
        ENDIF
        n=max_step-1
        ALLOCATE ( e(n) , d(n) )
        ALLOCATE ( w(0:max_step), r(0:max_step))
        d=0. ; e =0.
! define the step size
        step=(rmax-rmin)/FLOAT(max_step)
        hh(interpol)=step*step
! define constants for the matrix to be diagonalized
        const1=2./(step*step)
        const2=-1./(step*step)
!   set up r, the distance from the nucleus and the function w for energy =0
!   w corresponds then to the potential
!   values at
        DO i=0, max_step
          r(i) = rmin+i*step
          w(i) = potential(r(i))
        ENDDO
!   setup the diagonal d and the non-diagonal part e of
!   the tridiagonal matrix matrix to be diagonalized
        d(1:n)=const1+w(1:n) ; e(1:n)=const2
```

```
! allocate space for eigenvector info
      ALLOCATE ( z(n,n) )
! obtain the eigenvalues
      CALL tqli(d,e,n,z)
! sort eigenvalues as an ascending series
      CALL eigenvalue_sort(d,n)
      DEALLOCATE (z)
      err1=0.
! the interpolation part starts here
      DO l=1,20
        err2=0.
        value(interpol,l)=d(l)
        inp=d(l)
        IF ( interpol > 1 ) THEN
          CALL polint(hh,value(:,l),interpol,0.d0 ,inp,err2)
          err1=MAX(err1,err2)
          WRITE(6,'(D12.6,2X,D12.6,2X,D12.6)') inp, d(l), err1
        ELSE
          WRITE(6,'(D12.6,2X,D12.6,2X,D12.6)') d(l), d(l), err1
        ENDIF
      ENDDO
      DEALLOCATE ( w, r, d, e)
    ENDDO
```

7.9 Exercises

The aim of this problem is to solve Schrödinger's equation for two electrons in a three-dimensional harmonic oscillator well with and without a repulsive Coulomb interaction. Your task is to solve this equation by reformulating it in a discretized form as an eigenvalue equation to be solved with Jacobi's method. To achieve this you will have to write your own code which implements Jacobi's method.

Electrons confined in small areas in semiconductors, so-called quantum dots, form a hot research area in modern solid-state physics, with applications spanning from such diverse fields as quantum nano-medicine to the contruction of quantum gates.

Here we will assume that these electrons move in a three-dimensional harmonic oscillator potential (they are confined by for example quadrupole fields) and repel each other via the static Colulomb interaction. We assume spherical symmetry.

We are first interested in the solution of the radial part of Schrödinger's equation for one electron. This equation reads

$$-\frac{\hbar^2}{2m}\left(\frac{1}{r^2}\frac{d}{dr}r^2\frac{d}{dr}-\frac{l(l+1)}{r^2}\right)R(r)+V(r)R(r)=ER(r).$$

In our case $V(r)$ is the harmonic oscillator potential $(1/2)kr^2$ with $k = m\omega^2$ and E is the energy of the harmonic oscillator in three dimensions. The oscillator frequency is ω and the energies are

$$E_{nl} = \hbar\omega\left(2n + l + \frac{3}{2}\right),$$

with $n = 0, 1, 2, \dots$ and $l = 0, 1, 2, \dots$.

Since we have made a transformation to spherical coordinates it means that $r \in [0, \infty)$. The quantum number l is the orbital momentum of the electron. Then we substitute $R(r) = (1/r)u(r)$ and obtain

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r) + \left(V(r) + \frac{l(l+1)}{r^2}\frac{\hbar^2}{2m}\right)u(r) = Eu(r).$$

The boundary conditions are $u(0) = 0$ and $u(\infty) = 0$.

We introduce a dimensionless variable $\rho = (1/\alpha)r$ where α is a constant with dimension length and get

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(\rho) + \left(V(\rho) + \frac{l(l+1)}{\rho^2}\frac{\hbar^2}{2m\alpha^2}\right)u(\rho) = Eu(\rho).$$

We will set in this project $l = 0$. Inserting $V(\rho) = (1/2)k\alpha^2\rho^2$ we end up with

$$-\frac{\hbar^2}{2m\alpha^2}\frac{d^2}{d\rho^2}u(\rho) + \frac{k}{2}\alpha^2\rho^2u(\rho) = Eu(\rho).$$

We multiply thereafter with $2m\alpha^2/\hbar^2$ on both sides and obtain

$$-\frac{d^2}{d\rho^2}u(\rho) + \frac{mk}{\hbar^2}\alpha^4\rho^2u(\rho) = \frac{2m\alpha^2}{\hbar^2}Eu(\rho).$$

The constant α can now be fixed so that

$$\frac{mk}{\hbar^2}\alpha^4 = 1,$$

or

$$\alpha = \left(\frac{\hbar^2}{mk}\right)^{1/4}.$$

Defining

$$\lambda = \frac{2m\alpha^2}{\hbar^2}E,$$

we can rewrite Schrödinger's equation as

$$-\frac{d^2}{d\rho^2}u(\rho) + \rho^2u(\rho) = \lambda u(\rho).$$

This is the first equation to solve numerically. In three dimensions the eigenvalues for $l = 0$ are $\lambda_0 = 3, \lambda_1 = 7, \lambda_2 = 11, \dots$.

We use the by now standard expression for the second derivative of a function u

$$u'' = \frac{u(\rho+h) - 2u(\rho) + u(\rho-h)}{h^2} + O(h^2),$$

where h is our step. Next we define minimum and maximum values for the variable ρ, $\rho_{\text{min}} = 0$ and ρ_{max}, respectively. You need to check your results for the energies against different values ρ_{max}, since we cannot set $\rho_{\text{max}} = \infty$.

With a given number of steps, n_{step}, we then define the step h as

$$h = \frac{\rho_{\text{max}} - \rho_{\text{min}}}{n_{\text{step}}}.$$

Define an arbitrary value of ρ as

$$\rho_i = \rho_{\text{min}} + ih \qquad i = 0, 1, 2, \dots, n_{\text{step}}$$

we can rewrite the Schrödinger equation for ρ_i as

$$-\frac{u(\rho_i + h) - 2u(\rho_i) + u(\rho_i - h)}{h^2} + \rho_i^2 u(\rho_i) = \lambda u(\rho_i),$$

or in a more compact way

$$-\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + \rho_i^2 u_i = -\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + V_i u_i = \lambda u_i,$$

where $V_i = \rho_i^2$ is the harmonic oscillator potential. Define first the diagonal matrix element

$$d_i = \frac{2}{h^2} + V_i,$$

and the non-diagonal matrix element

$$e_i = -\frac{1}{h^2}.$$

In this case the non-diagonal matrix elements are given by a mere constant. *All non-diagonal matrix elements are equal.* With these definitions the Schrödinger equation takes the following form

$$d_i u_i + e_{i-1} u_{i-1} + e_{i+1} u_{i+1} = \lambda u_i,$$

where u_i is unknown. We can write the latter equation as a matrix eigenvalue problem

$$\left(\begin{array}{ccccccc} d_1 & e_1 & 0 & 0 & \dots & 0 & 0 \\ e_1 & d_2 & e_2 & 0 & \dots & 0 & 0 \\ 0 & e_2 & d_3 & e_3 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & d_{n_{\text{step}}-2} & e_{n_{\text{step}}-1} \\ 0 & \dots & \dots & \dots & \dots & e_{n_{\text{step}}-1} & d_{n_{\text{step}}} \end{array} \right) \left(\begin{array}{c} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{n_{\text{step}}-1} \end{array} \right) = \lambda \left(\begin{array}{c} u_1 \\ u_2 \\ \dots \\ \dots \\ \dots \\ u_{n_{\text{step}}-1} \end{array} \right) \tag{7.6}$$

or if we wish to be more detailed, we can write the tridiagonal matrix as

$$\left(\begin{array}{ccccccc}
\frac{2}{h^2}+V_1 & -\frac{1}{h^2} & 0 & 0 & \dots & 0 & 0 \\
-\frac{1}{h^2} & \frac{2}{h^2}+V_2 & -\frac{1}{h^2} & 0 & \dots & 0 & 0 \\
0 & -\frac{1}{h^2} & \frac{2}{h^2}+V_3 & -\frac{1}{h^2} & 0 & \dots & 0 \\
\dots & \dots & \dots & \dots & \dots & \dots & \dots \\
0 & \dots & \dots & \dots & \dots & \frac{2}{h^2}+V_{n_{\mathrm{step}}-2} & -\frac{1}{h^2} \\
0 & \dots & \dots & \dots & \dots & -\frac{1}{h^2} & \frac{2}{h^2}+V_{n_{\mathrm{step}}-1}
\end{array}\right) \quad (7.7)$$

Recall that the solutions are known via the boundary conditions at $i = n_{\mathrm{step}}$ and at the other end point, that is for ρ_0. The solution is zero in both cases.

a) Your task here is to write a function which implements Jacobi's rotation algorithm in order to solve Eq. (7.6).

   We Define the quantities $\tan\theta = t = s/c$, with $s = \sin\theta$ and $c = \cos\theta$ and

$$\cot 2\theta = \tau = \frac{a_{ll} - a_{kk}}{2a_{kl}}.$$

   We can then define the angle θ so that the non-diagonal matrix elements of the transformed matrix a_{kl} become non-zero and we obtain the quadratic equation (using $\cot 2\theta = 1/2(\cot\theta - \tan\theta)$)

$$t^2 + 2\tau t - 1 = 0,$$

   resulting in

$$t = -\tau \pm \sqrt{1+\tau^2},$$

   and c and s are easily obtained via

$$c = \frac{1}{\sqrt{1+t^2}},$$

   and $s = tc$. Explain why we should choose t to be the smaller of the roots. Show that these choice ensures that $|\theta| \leq \pi/4$) and has the effect of minimizing the difference between the matrices **B** and **A** since

$$||\mathbf{B} - \mathbf{A}||_F^2 = 4(1-c)\sum_{i=1,i\neq k,l}^{n}(a_{ik}^2 + a_{il}^2) + \frac{2a_{kl}^2}{c^2}.$$

b) How many points n_{step} do you need in order to get the lowest three eigenvalues with four leading digits? Remember to check the eigenvalues for the dependency on the choice of $\rho_{\max}$.

   How many similarity transformations are needed before you reach a result where all non-diagonal matrix elements are essentially zero? Try to estimate the number of transformations and extract a behavior as function of the dimensionality of the matrix.

You can check your results against the code based on Householder's algorithm, *tqli* in the file lib.cpp.

Comment your results (here you could for example compute the time needed for both algorithms for a given dimensionality of the matrix).

c) We will now study two electrons in a harmonic oscillator well which also interact via a repulsive Coulomb interaction. Let us start with the single-electron equation written as

$$-\frac{\hbar^2}{2m}\frac{d^2}{dr^2}u(r)+\frac{1}{2}kr^2u(r)=E^{(1)}u(r),$$

where $E^{(1)}$ stands for the energy with one electron only. For two electrons with no repulsive Coulomb interaction, we have the following Schrödinger equation

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dr_1^2}-\frac{\hbar^2}{2m}\frac{d^2}{dr_2^2}+\frac{1}{2}kr_1^2+\frac{1}{2}kr_2^2\right)u(r_1,r_2)=E^{(2)}u(r_1,r_2).$$

Note that we deal with a two-electron wave function $u(r_1,r_2)$ and two-electron energy $E^{(2)}$.

With no interaction this can be written out as the product of two single-electron wave functions, that is we have a solution on closed form.

We introduce the relative coordinate $\mathbf{r}=\mathbf{r}_1-\mathbf{r}_2$ and the center-of-mass coordinate $\mathbf{R}=1/2(\mathbf{r}_1+\mathbf{r}_2)$. With these new coordinates, the radial Schrödinger equation reads

$$\left(-\frac{\hbar^2}{m}\frac{d^2}{dr^2}-\frac{\hbar^2}{4m}\frac{d^2}{dR^2}+\frac{1}{4}kr^2+kR^2\right)u(r,R)=E^{(2)}u(r,R).$$

The equations for r and R can be separated via the ansatz for the wave function $u(r,R)=\psi(r)\phi(R)$ and the energy is given by the sum of the relative energy E_r and the center-of-mass energy E_R, that is

$$E^{(2)}=E_r+E_R.$$

We add then the repulsive Coulomb interaction between two electrons, namely a term

$$V(r_1,r_2)=\frac{\beta e^2}{|\mathbf{r}_1-\mathbf{r}_2|}=\frac{\beta e^2}{r},$$

with $\beta e^2 = 1.44$ eVnm.

Adding this term, the r-dependent Schrödinger equation becomes

$$\left(-\frac{\hbar^2}{m}\frac{d^2}{dr^2} + \frac{1}{4}kr^2 + \frac{\beta e^2}{r}\right)\psi(r) = E_r\psi(r).$$

This equation is similar to the one we had previously in (a) and we introduce again a dimensionless variable $\rho = r/\alpha$. Repeating the same steps as in (a), we arrive at

$$-\frac{d^2}{d\rho^2}\psi(\rho) + \frac{mk}{\hbar^2}\alpha^4\rho^2\psi(\rho) + \frac{m\alpha\beta e^2}{\rho\hbar^2}\psi(\rho) = \frac{m\alpha^2}{\hbar^2}E_r\psi(\rho).$$

We want to manipulate this equation further to make it as similar to that in (a) as possible. We define $k_r = 1/4k$ The constant α is then again fixed so that

$$\frac{mk_r}{\hbar^2}\alpha^4 = 1,$$

or

$$\alpha = \left(\frac{\hbar^2}{mk_r}\right)^{1/4}.$$

Defining

$$\lambda = \frac{m\alpha^2}{\hbar^2}E,$$

we can rewrite Schrödinger's equation as

$$-\frac{d^2}{d\rho^2}\psi(\rho) + \rho^2\psi(\rho) + \frac{\gamma}{\rho} = \lambda\psi(\rho),$$

with

$$\gamma = \frac{m\alpha\beta e^2}{\hbar^2}.$$

We treat γ as a parameter which reflects the strength of the oscillator potential.

Here we will study the cases $\gamma = 0$, $\gamma = 0.5$, $\gamma = 1$, $\gamma = 2$ and $\gamma = 4$. for the ground state only, that is the lowest-lying state.

For $\gamma = 0$ you should get a result which corresponds to the relative energy of a non-interacting system. The way we have written the equations means you get the same as in (a) for $\gamma = 0$. Make sure your results are stable as functions of $\rho_{\max}$ and the number of steps.

We are only interested in the ground state with $l = 0$. We omit the center-of-mass energy.

You can reuse the code you wrote for (a), but you need to change the potential from ρ^2 to $\rho^2+\gamma/\rho$.

Comment the results for the lowest state (ground state) as function of varying strengths of γ.

For specific oscillator frequencies, the above equation has analytic answers, see the article by M. Taut, Phys. Rev. A 48, 3561 - 3566 (1993). The article can be retrieved from the following web address http://prola.aps.org/abstract/PRA/v48/i5/p3561

d) In this exercise we want to plot the wave function for two electrons as functions of the relative coordinate r and different values of γ. For $\gamma = 0$ your wave function should correspond to that of a harmonic oscillator. Varying γ, the shape of the wave function will change.

We are only interested in the wave function for the ground state with $l = 0$ and omit again the center-of-mass motion.

You can choose between two approaches; the first is to use the existing *tqli* function. Here the eigenvectors are obtained from the matrix $z[i][j]$, where the index j refers to eigenvalue j. The index i points to the value of the wave function in position ρ_j. That is, $u^{(\lambda_j)}(\rho_i) = z[i][j]$.

The eigenvectors are normalized. Plot then the normalized wave functions for different values of γ and comment the results.

The other alternative is to add a piece to your Jacobi routine which also returns the eigenvectors. This is the more difficult part. You will need to normalize the eigenvectors.

Bibliography

[1] G.H. Golub and C.F. Van Loan. *Matrix Computations*. John Hopkins University Press, 1996.