Chapter 1
Introduction

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

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

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

1. The Monte Carlo method or Metropolis algorithm, devised by John von Neumann, Stanislaw Ulam, and Nicholas Metropolis, discussed in chapters 11-14.
2. The simplex method of linear programming, developed by George Dantzig.
3. Krylov Subspace Iteration method for large eigenvalue problems in particular, developed by Magnus Hestenes, Eduard Stiefel, and Cornelius Lanczos, discussed in chapter 7.
4. The Householder matrix decomposition, developed by Alston Householder and discussed in chapter 7.
5. The Fortran compiler, developed by a team lead by John Backus, codes used throughout this text.
6. The QR algorithm for eigenvalue calculation, developed by Joe Francis, discussed in chapter 7
7. The Quicksort algorithm, developed by Anthony Hoare.
8. Fast Fourier Transform, developed by James Cooley and John Tukey.
9. The Integer Relation Detection Algorithm, developed by Helaman Ferguson and Rodney
10. The fast Multipole algorithm, developed by Leslie Greengard and Vladimir Rokhlin; (to calculate gravitational forces in an N-body problem normally requires N^2 calculations. The fast multipole method uses order N calculations, by approximating the effects of groups of distant particles using multipole expansions)

The topics we cover start with an introduction to C++ and Fortran programming (with digressions to Python as well) combining it with a discussion on numerical precision, a point

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

Therafter we switch to numerical integration for integrals with few dimensions, typically less than three, in chapter 5. The numerical integration chapter serves also to justify the introduction of Monte-Carlo methods discussed in chapters 11 and 12. There, a variety of applications are presented, from integration of multidimensional integrals to problems in statistical physics such as random walks and the derivation of the diffusion equation from Brownian motion. Chapter 13 continues this discussion by extending to studies of phase transitions in statistical physics. Chapter 14 deals with Monte-Carlo studies of quantal systems, with an emphasis on variational Monte Carlo methods and diffusion Monte Carlo methods. In chapter 7 we deal with eigensystems and applications to e.g., the Schrödinger equation rewritten as a matrix diagonalization problem. Problems from scattering theory are also discussed, together with the most used solution methods for systems of linear equations. Finally, we discuss various methods for solving differential equations and partial differential equations in chapters 8-10 with examples ranging from harmonic oscillations, equations for heat conduction and the time dependent Schrödinger equation. The emphasis is on various finite difference methods.

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

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

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

1.1 Choice of programming language

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

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

- C++ is now the dominating language in Unix and Windows environments. It is widely available and is the language of choice for system programmers. It is very widespread for developments of non-numerical software
- The C++ syntax has inspired lots of popular languages, such as Perl, Python and Java.
- It is an extremely portable language, all Linux and Unix operated machines have a C++ compiler.
- In the last years there has been an enormous effort towards developing numerical libraries for C++. Numerous tools (numerical libraries such as MPI [15–17]) are written in C++ and interfacing them requires knowledge of C++. Most C++ and Fortran compilers compare fairly well when it comes to speed and numerical efficiency. Although Fortran 77 and C are regarded as slightly faster than C++ or Fortran, compiler improvements during the last few years have diminshed such differences. The Java numerics project has lost some of its steam recently, and Java is therefore normally slower than C++ or Fortran.
- Complex variables, one of Fortran's strongholds, can also be defined in the new ANSI C++ standard.
- C++ is a language which catches most of the errors as early as possible, typically at compilation time. Fortran has some of these features if one omits implicit variable declarations.
- C++ is also an object-oriented language, to be contrasted with C and Fortran. This means that it supports three fundamental ideas, namely objects, class hierarchies and polymorphism. Fortran has, through the `MODULE` declaration the capability of defining classes, but lacks inheritance, although polymorphism is possible. Fortran is then considered as an object-based programming language, to be contrasted with C++ which has the capability of relating classes to each other in a hierarchical way.

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

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

The efficiency of C++ codes are close to those provided by Fortran. This means often that a code written in Fortran 77 can be faster, however for large numerical projects C++ and

Fortran are to be preferred. If speed is an issue, one could port critical parts of the code to Fortran 77.

1.1.0.1 Future plans

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

1.2 Designing programs

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

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

First about designing a program.

- Before writing a single line, have the algorithm clarified and understood. It is crucial to have a logical structure of e.g., the flow and organization of data before one starts writing.
- Always try to choose the simplest algorithm. Computational speed can be improved upon later.
- Try to write a as clear program as possible. Such programs are easier to debug, and although it may take more time, in the long run it may save you time. If you collaborate with other people, it reduces spending time on debugging and trying to understand what the codes do. A clear program will also allow you to remember better what the program really does!
- Implement a working code with emphasis on design for extensions, maintenance etc. Focus on the design of your code in the beginning and don't think too much about efficiency before you have a thoroughly debugged and verified program. A rule of thumb is the so-called $80-20$ rule, 80 % of the CPU time is spent in 20 % of the code and you will experience that typically only a small part of your code is responsible for most of the CPU expenditure. Therefore, spend most of your time in devising a good algorithm.
- The planning of the program should be from top down to bottom, trying to keep the flow as linear as possible. Avoid jumping back and forth in the program. First you need to arrange

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

the major tasks to be achieved. Then try to break the major tasks into subtasks. These can be represented by functions or subprograms. They should accomplish limited tasks and as far as possible be independent of each other. That will allow you to use them in other programs as well.

- Try always to find some cases where an analytical solution exists or where simple test cases can be applied. If possible, devise different algorithms for solving the same problem. If you get the same answers, you may have coded things correctly or made the same error twice.
- When you have a working code, you should start thinking of the efficiency. Analyze the efficiency with a tool (profiler) to predict the CPU-intensive parts. Attack then the CPU-intensive parts after the program reproduces benchmark results.

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

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

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

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