Chapter 1

Introduction

The principle focus of this course is the discipline of scientific programming. The aim is to acquaint the students with how to program computers to solve analytically (pencil and paper approach) difficult but numerically tractable mathematical problems.

1.1 Justification for scientific computing

The study of natural physical phenomena must be approached in a variety of ways. They include observations, measurements, experiments, and last but not least building abstract physical/mathematical models of the phenomena of interest. These abstract models are necessary for the purpose of elucidating the underlying physical principle, and for design and forecasting purposes. An example include understanding and predicting atmospheric flows like hurricane formation, steering and intensification. Fluid flows for example are governed by the Navier-Stokes equations which are derived based on the principle of momentum, mass and energy conservation. This constitutes for example the abstraction phase into a mathematical model. Unfortunately, these equations are very hard to solve analytically: they are nonlinear so we cannot superpose simple solutions to build complex ones, they occur in geometrically complicated regions like ocean basins, or flow around cars and airplanes, or around the Earth, and are subject to complicated forcing mechanisms. These equations must be solved numerically which requires an algorithm suitable for computers, and the implementation of that algorithm on a computer, i.e. programming. We are mostly concerned with the last step here. However, to provide examples of programming we will study several numerical algorithms that one encounters in science and engineering. The emphasis however, will not be on studying these algorithms and their stability but on programming them.
1.2 Types of programming tasks

Programming encompasses a wide variety of subcategories that can be classified according to the target applications. These include data base programming, i.e. organizing, storing and storing vast amount of dynamically changing data, for example to handle a company’s inventory, or bank transaction. Other programming disciplines revolve around graphics programming (for games), or system programming to create operating systems and graphical user interfaces. The emphasis on these tasks is on creating efficient data structures, managing algorithm complexity, etc. The task at hand is substantially simpler from a programming point of view, and it involves chiefly numerical programming, i.e. manipulating data and input to come with numerical answers and occasionally plotting these results on a computer screens. Although the programming tasks can be easy, the art and the skill lies in the selection of the “optimal” numerical algorithms to solve a particular problem. To do this we need to draw on a number of disciplines and intuitions that include physics, mathematics, and computer science.

A simple example consists of solving for example the dispersion relationship of free surface gravity waves, and which relates the frequency of waves in time to their wavelength in space:

\[ \sigma^2 = gk \tanh kh, \]  

(1.1)\]

where \( \sigma = 2\pi/T \) is the wave frequency and \( T \) is its time periodicity, \( g \) is the gravitational acceleration, \( h \) is the local water depth, \( k = 2\pi/\lambda \) is the wavenumber and \( \lambda \) is the wavelength. The task is to solve for \( k \) (or \( \lambda \) knowing \( \sigma \). The problem is complicated because the equation is transcendental and does not lend itself to algebraic manipulation.

The first order of business is to non-dimensionalize the equation so that units are not an issue. We thus treat \( kh \) as a single variable and rewrite the equation as:

\[ \frac{\sigma^2h}{g} = kh \tanh kh, \]  

(1.2)\]

One solution is to solve the equation graphically as shown in figure 1.1: the roots are the intersections of the curves: \( \tanh kh \) with the curve \( \frac{\sigma^2h}{g}(kh) \), three of which are drawn for different values of the parameter \( \frac{\sigma^2h}{g} \).

A second solution is to use analytical approximations. A mathematically inclined scientist would notice that for small values of \( \frac{\sigma^2h}{g} \), we can approximate \( \tanh kh \) with \( kh \) (the two curves overlap near 0 and separate gradually); the solution is then easy to write as \( k \approx \sigma/\sqrt{gh} \). On the other side of the spectrum for large \( kh \) (i.e. when the depth is much larger then the wavelength), \( \tanh kh \) is almost one and the solution is simply \( kh \approx \sigma^2h/g \). In the intermediate regime, neither approximation is good and one must use a numerical method to solve for the wavenumber \( k \). Algorithms for solving nonlinear algebraic equations will be discussed later in this course.
1.3 Why fortran

There are a number of programming languages available to implement algorithms on computers. These languages are often optimized with a particular kind of programming/applications in mind. These include Java for web-enabled and networked applications; C++ for managing large complex projects; C for system programming; COBOL for commercial transactions; LISP for artificial intelligence programming. Fortran is one of the older languages and was designed specifically for numerical applications. The language is often disparaged for its lack of power and expressiveness; but its strength lies in the fact that it effectively shields the programmer from the hardware, is easy to learn and use, produces optimal machine code that allow it to run fast, and has built-in support for the most widely used data structures in numerical computing, namely multidimensional arrays to represent matrices and vectors. The newer variants of the language, (e.g. FORTRAN 90, FORTRAN 95, and FORTRAN 2000) expand the language substantially and allow more elaborate code to be written easily. Moreover, there is a large body of excellent time-tested software libraries written in fortran. These are tremendous assets to anybody who is building an application but does not want to right every single line of code from scratch.

We will not study every single aspect of the programming language. This is akin to rote learning and is not very educational in my opinion, since we are likely to forget things we seldom use. Most scientists, however, use a small subset of the language to complete their tasks, and here we will focus on the most useful aspects.
to accomplish our tasks. This is of course biased by my experience as a numerical programmer. I will steer away from the old fortran statements that are dangerous and have been superseded by newer more expressive ones.

1.4 Software Tools

A number of software tools need to be learned to practice programming. Some of them are essential to operating a computer, while others are tools to simplify the maintenance and validation of computer codes.

1.4.1 Operating System

Most large scientific computing problems are run on machines running the Linux operating system. What is required is actually a minimal understanding of how to manipulate files in UNIX, create directory, etc...

1.4.2 Editor

Writing program requires an editor to write the text of the source code. In contrast to text processors (like WORD and WORD PERFECT) that format text, an editor plays a humbler role of simply storing alphanumeric characters in the computer. A number of editors are available, some are more visually oriented then others. We will focus on using xemacs a popular editor equipped with a GUI.

1.4.3 Compiler

Computers understand only machine language, i.e. 0’s and 1’s. A compiler is thus needed to translate the high level programming language into machine instructions. We will use the Portland Group F90 compiler, pgf90, for that. Compilers have lots of options to help in validating code or enhance its performance. We will learn about those in due course.

1.4.4 Debugger

Although it would be nice to code a program and have it run the first time, things seldom work out that way. Errors and bugs are often introduced while writing the source code. Some of them are caught at compile time with, usually, a warning message from the compiler. Others are not and only manifest themselves at run time with an unexpected termination (crash) of the code. Perusing the code is often fruitless in tracking the bugs. Inserting print statements can help. However, nothing beats following the code execution line by line using a visual debugger. Luckily the pgf90 compiler comes with a debugger, pgdbg, that will come in handy.
1.4. SOFTWARE TOOLS

1.4.5 Make

As the programming tasks become more complicated, it became imperative to divide the code among different files. These files need not be compiled simultaneously or need be compiled in a specific order. A make utility is available on many UNIX platform to perform this task.

1.4.6 Visualization Package

Since the output of many computations is a function defined on a discrete set of points, we will often need to draw the output using some graphics package. We will do that in matlab.