

PHY 329

Introduction to Computational Physics

a college level course given by

Richard Fitzpatrick

ASSOCIATE PROFESSOR OF PHYSICS

The University of Texas at Austin

Fall 2002

Tel.: 512-471-9439

Office: RLM 11.324

Email: rfitzp@farside.ph.utexas.edu

Course homepage: <http://farside.ph.utexas.edu/teaching/329/329.html>

1 Introduction

1.1 Major sources

The sources which I have consulted most frequently whilst developing course material are as follows:

Linux operating system:

Unix notes, M.W. Choptuik (http://laplace.physics.ubc.ca/People/-matt/Teaching/98Fall/Phy329/Notes_unix.html), 1998.

C/C++ programming:

Software engineering in C, P.A. Darnell, and P.E. Margolis (Springer-Verlag, New York NY, 1988).

The C++ programming language, 2nd edition, B. Stroustrup (Addison-Wesley, Reading MA, 1991).

Schaum's outline: Programming with C, 2nd edition, B. Gottfried (McGraw-Hill, New York NY, 1996).

Schaum's outline: Programming with C++, 2nd edition, J.R. Hubbard (McGraw-Hill, New York NY, 2000).

Numerical methods and computational physics:

Computational physics, D. Potter (Wiley, New York NY, 1973).

Numerical recipes in C: the art of scientific computing, W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.R. Flannery (Cambridge University Press, Cambridge UK, 1992).

Computational physics, N.J. Giordano (Prentice-Hall, Upper Saddle River NJ, 1997).

Numerical methods for physics, 2nd edition, A.L. Garcia (Prentice-Hall, Upper Saddle River NJ, 2000).

Physics of baseball:

The physics of baseball, R.K. Adair (Harper & Row, New York NY, 1990).

The physics of sports, A.A. Armenti, Jr., Ed. (American Institute of Physics, New York NY, 1992).

Note that much of the material appearing in Sect. 2 is adapted from notes written by Prof. M.W. Choptuik (Dept. of Physics and Astronomy, Univ. of British Columbia, Canada) whilst teaching at the University of Texas at Austin. Prof. Choptuik's original notes are available at the URL listed above.

1.2 Purpose of course

The primary purpose of this course is demonstrate to students how computers can enable us to both broaden and deepen our understanding of physics by vastly increasing the range of mathematical calculations which we can conveniently perform.

The secondary purpose of this course is to introduce students to computing and numerical methods from the perspective of a research physicist. Students will be given the opportunity to tackle interesting numerical problems using the type of tools (both hardware and software), and the type of techniques, employed by professional physicists.

1.3 Course philosophy

My approach to computational physics is to write self-contained programs in a high-level scientific language—*i.e.*, either FORTRAN or C/C++—on a Linux platform. Of course, there are many other possible approaches, each with their own peculiar advantages and disadvantages. It is instructive to briefly examine the available options.

1.4 Hardware platforms

The choice of hardware platform is, perhaps, the least contentious choice of all. Nowadays, the overwhelming majority of physicists choose to work within UNIX environments. With the advent of Linux—a hugely popular, free clone of UNIX which runs on personal computers—there seems little prospect of this changing in the near future.

1.5 Programming methodologies

Basically, there are *three* possible methods by which we could perform the numerical calculations which are going to crop up during this course.

Firstly, we could use a mathematical software package, such as MATHEMATICA¹, MAPLE² or MATLAB.³ The main advantage of these packages is that they facilitate the very rapid coding up of numerical problems. The main disadvantage is that they produce executable code which is *interpreted*, rather than *compiled*. Compiled code is translated directly from a high-level language into machine code instructions, which, by definition, are platform dependent—after all, an Intel x86 chip has a completely different instruction set to a Power-PC chip. Interpreted code is translated from a high-level language into a set of meta-code instructions which are platform independent. Each meta-code instruction is then translated into a fixed set of machine code instructions which is peculiar to the particular hardware platform on which the code is being run. In general, interpreted code is nowhere near as efficient, in terms of computer resource utilization, as compiled code: *i.e.*, interpreted code run *a lot slower* than equivalent compiled code. Thus, although MATHEMATICA, MAPLE, and MATLAB are ideal environments in which to perform relatively *small* calculations, they are not suitable for full-blown research projects, since the code which they produce generally runs far too slowly.

Secondly, we could write our own programs in a high-level language, but use calls to pre-written, pre-compiled routines in commonly available subroutine

¹See <http://www.wolfram.com>

²See <http://www.maplesoft.com>

³See <http://www.mathworks.com>

libraries, such as NAG,⁴ LINPACK,⁵ and ODEPACK,⁶ to perform all of the real numerical work. This is the approach used by the majority of research physicists.

Thirdly, we could write our own programs—completely from scratch—in a high-level language. This is the approach used in this course. I have opted *not* to use pre-written subroutine libraries, simply because I want students to develop the ability to think for themselves about scientific programming and numerical techniques. Students should, however, realize that, in many cases, pre-written library routines offer solutions to numerical problems which are pretty hard to improve upon.

1.6 Scientific programming languages

What is the best high-level language to use for scientific programming? This, unfortunately, is a highly contentious question. Over the years, literally hundreds of high-level languages have been developed. However, few have stood the test of time. Many languages (*e.g.*, Algol, Pascal, Haskell) can be dismissed as ephemeral computer science fads. Others (*e.g.*, Cobol, Lisp, Ada) are too specialized to adapt for scientific use. Let us examine the remaining options:

FORTRAN 77: FORTRAN was the first high-level programming language to be developed: in fact, it predates the languages listed below by decades. Before the advent of FORTRAN, all programming was done in assembler code! Moreover, FORTRAN was specifically designed for scientific computing. Indeed, in the early days of computers *all* computing was scientific in nature—*i.e.*, physicists and mathematicians were the original computer scientists!. FORTRAN's main advantages are that it is very straightforward, and it interfaces well with most commonly available, pre-written subroutine libraries (since these libraries generally consist of compiled FORTRAN code). FORTRAN's main disadvantages are all associated with its relative

⁴See <http://www.nag.com>

⁵See <http://www.netlib.org>

⁶*ibid.*

antiquity. For instance. FORTRAN's control statements are fairly rudimentary, whereas its input/output facilities are positively paleolithic.

FORTRAN 90: This language is a major extension to FORTRAN 77 which does away with many of the latter language's objectionable features. In addition, many "modern" features, such as dynamic memory allocation, are included in the language for the first time. The major disadvantage of this language is the absence of a free compiler for Linux platforms. There seems little prospect of this situation changing in the near future.

C: This language was originally developed by computer scientists to write operating systems. Indeed, all UNIX operating systems, including Linux, are written in C. C is, consequently, an extremely flexible and powerful language. Amongst its major advantages are its good control statements and excellent input/output facilities. C's main disadvantage is that, since it was not specifically written to be a scientific language, some important scientific features (*e.g.*, complex arithmetic) are missing. Although C is a high-level language, it incorporates many comparatively low-level features, such as pointers (this is hardly surprisingly, since C was originally designed to write operating systems). The low-level features of C—in particular, the rather primitive implementation of arrays—sometimes make scientific programming more complicated than need be the case, and undoubtedly facilitate programming errors. On the other hand, these features allow scientific programmers to write *extremely* efficient code. Since efficiency is generally the most important concern in scientific computing, the low-level features of C are, on balance, advantageous.

C++: This language is a major extension of C whose main aim is to facilitate object-orientated programming. Object-orientation is a completely different approach to programming than the more traditional procedural approach: it is particularly well suited to large projects involving many people who are each writing different segments of the same code. However, object-orientation represents a large, and somewhat unnecessary, overhead for the type of straightforward, single person programming tasks considered in this course. Note, however, that C++ incorporates some non-object-orientated extensions to C which are extremely useful.

JAVA: This is currently the most popular object-orientated programming language. Its main advantage is its relative simplicity—at least, compared to C++. Its main disadvantage is that it is interpreted, rather than compiled—this accounts for its much-touted “platform-independence” feature.

Of the above languages, we can immediately rule out JAVA, because it is interpreted, C++, because object-orientation is an unnecessary complication (at least, for our purposes), and FORTRAN 90, because there is no suitable compiler available for Linux platforms. The remaining options are FORTRAN 77 and C. I have chosen to use C (augmented by some of the useful, non-object-orientated features of C++) in this course, simply because I find the archaic features of FORTRAN 77 too embarrassing to teach students in the 21st century.