# Scientific Computing
# Introduction

Jonathan Goodman
Courant Institute of Mathemaical Sciences

started January 18, 2002

Scientific computing means using the computer to solve equations, compute averages, or generally to carry out mathematical computations. This is truly a multdisciplinary job, requiring mathematics, appreciation for computer archetecture, software engineering, and the problem area. These notes, intended for a one semester course, present this landscape in minneature. The mathematics includes use of Taylor series, linear algebra, condition number, the discrete Fourier transform, and basic probability behind Monte Carlo. From basic numerical analysis we discuss order of accuracy, Newton's method, and time stepping methods for ordinary differential equations. Strictly computer related material includes the IEEE standard for floating point arithmetic and something about coding numerical algorithms for modern pipelined and cached computers, material that may be oblolete by the end of the semester. We also explore issues related to constructing large reliable numerical software.

The prerequisites for these notes are linear algebra, multivariate calculus, some familiarity with elementary probability (for the Monte Carlo material), and basic computer literacy. We will write C/C++ programs and use software tools for visualization and debugging. For visualization we strongly recommend Matlab, although Excel is a barely acceptable alternative. A motivated student mitght be able learn C programming during the class. The web site has some help with this.

In numerical computing we never expect to get the *exact* answer.[1] There are four basic sources of error: roundoff error, truncation error, and statistical error, and software error. Roundoff error arises from the fact that computer arithmetic is not exact. If we know $x$ and $y$ and write

```
z = x + y
```

then the computed $z$ will usually not be exactly the sum of $x$ and $y$. Exactly what $z$ will be is discussed when we cover IEEE arithmetic. Truncation error

[1] This is almost the definition of numerical computation. Computations that aim for the exact answer, such as symbolic algebraic manipulation or prime factorization of large integers, are often called "non numerical".

arises from the use of approximate formulas, such as

$$f'(x) \approx (f(x+h) - f(x))/h \ ,$$

that are not exact "even in exact arithmetic". Some computations, particularly in linear algebra, have no truncation error. In most computations that have truncation error, the truncation error is much larger than roundoff error. Also, as will become clear throughout the course, much more can be done to reduce truncation error than roundoff error. For these reasons, most scientific computing courses (including this one) spend more time analyzing truncation error than roundoff error. Statistical error arises only in Monte-Carlo computations.

Often, inaccuracy in a computed answer is very large compared to the size of roundoff, truncation, or statistical error introduced during the computation. This is because error can be amplified during the stages of the computational algorithm. Such error amplification is inevitable if the problem being solved is *ill conditioned.* An ill conditioned problem is one in which the answer is so sensitive to the input parameters that no computational algorithm could would be expected to give an accurate approximation to it. However, even well conditioned problems may have solution algorithms that are *unstable.* To get an accurate answer, the problem being solved must be well conditioned (this may be out of the hands of the person trying to solve the problem) and the solution algorithm must be stable.

It is important to be aware of the possibility of inaccuracy via error amplification because this source of error is hardest to discover by standard debugging techniques. In a large calculation, the error may grow a seemingly negligible amount at each step but grow to swamp the correct answer by the time the computation is finished. One of the main uses of mathematical analysis in scientific computing is in understanding the conditioning of problems and the stability of algorithms.

With all these sources of error, how do we know which errors are due to software bugs? This is particularly serious for numerical computation because we do not know the answer. If the computed numbers are not obviously wrong are they then right? Scientific programmers have strategies, experimentalists would call them "protocols", for building trustworthy codes. Several of the programming exercises focus on this issue.

In scientific computing, "performance" refers to the speed of a computation. To get good performance, one must be aware of how things happen inside the computer. What happens in a procedure call, or when we divide by zero, or execute `sum += a[i++];`? The programmer has more influence over this that he or she might imagine. Being aware of the factors influencing performance can make the code much faster without changing the algorithm or the computer.

The reader will probably not find the everything he or she needs to know here. There is nothing about partial differential equations, no iterative methods or eigenvalue algorithms. Even the topics that are covered, direct factorization mathods, optimization, ordinary differential equations (dynamics), and Monte Carle, are not done in nearly the depth an actual practitioner would need.

I give several excuses for these ommissions. I wanted a short book for a one sementer course suitable to students from science, engineering, applied mathematics, and finance. This book represents the common ground. Finally, the specialized material is easier to come by than this background.