Principles of Scientific Computing

Jonathan Goodman
Department of Mathematics
Courant Institute of Mathematical Sciences
New York University
goodman@cims.nyu.edu

Date compiled: January 21, 2003

Preface

Last revised: **January 2, 2003**

This book grew out of a course in Scientific Computing for graduate students at New York University, a first course that covers the basic principles common to most applications. It addresses the question: What are the basics that everyone doing scientific computing needs to know? The answer is: some mathematics, the most basic algorithms, a bit about the workings of a computer, and an idea how to build software for scientific computing applications. Naturally, specific applications (e.g. computing stresses in a bridge) also require more advanced specific material (enginering mechanics, finite elements, etc.).

The principles of scientific computing are a collection of simple, almost obvious, ideas and points of view. The practitioner is hardly a cook relying printed recipies or downoaded software, but a creative problem solver who can devise algorithms and build trustworthy software for new computational challenges. I hope that the reader will come to share my delight in the simplicity and admiration of the power of these simple principles.

This book requires a facility with the mathematics that is common to most quantitative modeling: multivariate calculus, linear algebra, and basic probability. These subjects may be reviewed using books in the *Schaum's Outline* series, particularly, *Multivariate Calculus*, *Linear Algebra*, and *Probability.* The exercises require programming in C or C++. The differences between these are not so important for the small simple programs called for. The book is structured so that an ambitious student can learn programming as he or she goes. See Appendix I. It is possible to do the programming in Fortran, but students are discouraged from using a programming language, such as Java, Visual Basic, or Matlab, not designed for efficient large scale scientific computing.

Visualization and data analysis are an essential part of scientific computing. We recommend Matlab as a simple and reliable system for visualization and examination of computational results. Students familiar with other scientific visualization software are welcome to use it. However, we warn the student that Mathematica is often unreliable. Excel users will need to be sophisticated enough to turn off the default features intended for business presentations, such as shading on bar graphs.

Many students will want to compute on their personal computer. Any current laptop or desktop should be powerful enough. The student will need a C/C++ compiler and visualization software such as Matlab. Some of the exercises require the student to download files and software, but always plain text files or C/C++ source code.

Many of my views on scientific computing were formed during my association with the remarkable group of faculty and graduate students at Serra House, the numerical analysis group of the Computer Science Department of Stanford University, in the early 1980's. I mention in particularly Marsha Berger, Petter Björstad, Bill Coughran, Gene Golub, Bill Gropp, Eric Grosse, Bob Higdon, Randy LeVeque, Steve Nash, Joe Oliger, Michael Overton, Nick Trefethen, and Margaret Wright. Colleagues at the Courant Institute who have influenced this book include Leslie Greengard, Gene Isaacson, Peter Lax, Charlie Peskin, Luis Reyna, Mike Shelley, and Olof Widlund. I also acknowledge the lovely book *Numerical Methods* by Germund Dahlquist and Åke Björk.

Contents

Chapter 1

Introduction

Scientific Computing is truly multidisciplinary.

do question 2.**??**

see figure 2.9

See 2.1 relatively soon.

Chapter 2

Sources of Error

In scientific computing, we never expect to get the exact answer. Inexactness is practically the definition of scientific computing. Getting the exact answer, generally with integers or rational numbers, is *symbolic computing*, an interesting but distinct subject.

To put this technically, suppose we are trying to compute the number A. The computer will produce an approximation, which we call $\widehat{A}$. This notation is borrowed from statistics, where an estimate for a quantity A is often called $\widehat{A}$. Whether A and $\widehat{A}$ agree to 16 decimal places or differ by 30%, we never expect the identity $A = \widehat{A}$ to be true in the mathematical sense, if only because we do not have an exact representation for A. For example, if we need to find x so that $x^2 = 175$, we might get 13 or 13.22876, depending on our computational method, but we cannot represent the exact $\sqrt{175}$ if we only allow finitely many digits.

Since the answer is never exactly right, it can be hard to know whether errors in computed results are due to a poor algorithm, an impossible problem (technically, *ill conditioned*), or a software bug. Understanding the sources of error will help us know their behavior well enough to recognize them and tell them apart. We will use this constantly in devising algorithms and building and testing software.

Of the four sources of error mentioned, we discuss only roundoff error in detail here. The others are discussed at length in later chapters. To explain roundoff error, we pause to discuss computer floating point arithmetic in some detail. Thanks to the IEEE Floating Point Standard [], it we can predict the outcome of inexact computer arithmetic in most cases on most computers.

2.1 Relative and absolute error

The *absolute error* in approximating A by $\widehat{A}$ is $e = \widehat{A} - A$. The *relative error*, which is $\epsilon = e/A$, is usaually more meaningful. Some algebra puts these definitions into the form

$$\widehat{A} = A + e \quad \text{(absolute error)} \ , \qquad \widehat{A} = A \cdot (1 + \epsilon) \quad \text{(relative error)}. \qquad (2.1)$$

For example, the absolute error in approximating $A = \sqrt{175}$ by $\widehat{A} = 13$ is $e = 13.22876 \cdots - 13 \approx .229$. The corresponding relative error is $e/A \approx .229/13.23 \approx .017 < 2\%$. Saying that the error is less than 2% is probably more informative than saying that the error is less than $.25 = 1/4$.

Relative error is a *dimensionless* measure of error. In practical situations, the desired A probably has units, seconds, meters, etc. Knowing that the error is, say, .229 meters does not tell you whether that is large or small. If the correct length is half a meter, then .229 meters is a large error. If the correct length is $13.22876 \cdots$ meters, the approximate answer is off by less than 2%. If we measure lengths in centimeters, then the error becomes 22.9cm. Is 22.9 a large error? It is less than 2% of the exact length, $1,322.876 \cdots$ cm.

2.2 Computer arithmetic

One of the many sources of error in scientific computing is inexact computer arithmetic, which is called *roundoff error.* Roundoff error is inevitable but its consequences vary. Some computations yield nearly exact results, while others give answers that are completely wrong. This is true even for problems that involve no other approximations. For example, solving systems of linear equations using gaussian elimination would give the exact answer if all the computations were performed exactly. When these computations are done in finite precision floating point arithmetic, we might or might not get something close to the right answer.

A single floating point operation almost always produces high relative accuracy. For given B and C, let $A = B \bigcirc C$, with $\bigcirc$ standing for one of the arithmetic operations: addition, subtraction, multiplication, or division. Then, *with the same B and C*, the computer will produce $\widehat{A}$ which satisfies (2.1) with $|\epsilon| \leq \epsilon_{\text{mach}}$, where ϵ_{mach} is the *machine precision.* Normally ϵ_{mach} is $2^{-24} \approx 6 \cdot 10^{-8}$ for single precision and $2^{-53} \approx 10^{-16}$ for double precision. The rough approximation $2^{10} = 1024 \approx 1000 = 10^3$ gives $2^{24} = 2^4 \cdot 2^{20} = 16 \cdot (2^{10})^2 \approx 16 \cdot (10^3)^2 = 16 \cdot 10^6$, so $2^{-24} \approx \frac{1}{16} 10^{-6} = 6.25 \cdot 10^{-8}$. Also, $2^{53} \approx 8 \cdot 10^{15}$ so the double precision machine precision is about $\frac{1}{8} \cdot 10^{-15} = 1.25 \cdot 10^{-16}$.

With uniformly high relative accuracy, how does computer arithmetic ever lead to the wrong answer? The answer is that $\widehat{A}$ may not be computed from the exact B and C, but from computed approximations $\widehat{B}$ and $\widehat{C}$. The relative accuracy of $\widehat{A}$ can be worse than the relative accuracies of $\widehat{B}$ and $\widehat{C}$. High relative accuracy can lost, quickly or slowly, during a multi stage computation. See Section 6, below.

2.2.1 Introducing the standard

The IEEE floating point standard is a set of conventions on computer representation and processing of floating point numbers. Modern computers follow these standards for the most part. The standard has four main goals:

1. To make floating point arithmetic as accurate as possible.

2. To produce sensible outcomes in exceptional situations.

3. To standardize floating point operations across computers.

4. To give the programmer control over exception handling.

The standard specifies exactly how numbers are represented in hardware. The most basic unit of information that a computer stores is a *bit*, a variable whose value may be either 0 or 1. Bits are organized into 32 bit or 64 bit *words* words. A 32 bit word is a *string* of 32 bits. The number of 32 bit words, or *bit strings*, is approximately (recall $2^{10} \approx 10^3$) $2^{32} = 2^2 \cdot 2^{30} \approx 4 \times (10^3)^3 = 4$ billion. A typical computer should take well under a minute to list all 32 bit words. A computer running at 1GHz in theory can perform one billion operations per second, though that is rarely achieved in practice. The number of 64 bit words is about $1.6 \cdot 10^{19}$ 64, which is too many to be listed in a year.

There are two basic computer data types that represent numbers. A fixed point number, or integer, has type `int` or `longint` in C/C++, depending on the number of bits. Floating point numbers have type `float` for *single precision* 32 bit words, and `double` for *double precision*, or 64 bit words. In early C compilers, a `float` by default had 64 bits instead of 32.

Integer arithmetic, also called fixed point, is very simple. For example, with 32 bit integers, the $4 \cdot 10^9$ distinct words represent that many consequtive integers, filling the range from about $-2 \cdot 10^9$ to about $2 \cdot 10^9$. Addition, subtraction, and multiplication are done exactly whenever the answer is within this range. The result is unpredictable when the answer is out of range (*overflow*). Results of integer division are rounded down to the nearest integer below the answer.

2.2.2 Representation of numbers, arithmetic operations

For scientific computing, integer arithmetic has two drawbacks. One is that there is no representation for numbers that are not integers. Equally important is the small range of values. The number of dollars in the US national debt, several trillion (10^{12}), cannot be represented as a 32 bit integer but is easy to approximate in 32 bit floating point.

A floating point number is a binary version of the exponential ("scientific") notation used on calculator displays. Consider the example expression:

$$-.2491\text{E} - 5$$

which is one way a calculator could display the number $-2.491 \cdot 10^{-6}$. This expression consists of a sign bit, $s = -$, a mantissa, $m = 2491$ and an exponent,

$e = -5$. The expression $s.m\mathrm{E}e$ corresponds to the number $s \cdot .m \cdot 10^e$. Scientists like to put the first digit of the mantissa on the left of the decimal point ($-2.491 \cdot 10^{-6}$) while calculators put the whole thing on the right ($-.2491 \cdot 10^{-5}$). In base 2 (binary) arithmetic, the scientists' convention saves a bit, see below.

The IEEE format for floating point numbers replaces the base 10 with base 2, and makes a few other changes. When a 32 bit word (bit string) is interpreted as a floating point number, the first bit is the sign bit, $s = \pm$. The next 8 bits form the "exponent", e, and the remaining 23 bits determine the "fraction", f. There are two possible signs, $2^8 = 256$ possible exponents (ranging from 0 to 255), and $2^{23} \approx 8$ million possible fractions. Normally a floating point number has the value

$$A = \pm 2^{e-127} \cdot (1.f)_2 \quad , \tag{2.2}$$

where f is base 2 and the notation $(1.f)_2$ means that the expression $1.f$ is interpreted in base 2. Note that the mantissa is $1.f$ rather than just the fractional part, f. Any number (except 0) can be normalized so that its base 2 mantissa has the form $1.f$. There is no need to store the "1." explicitly, which saves one bit.

For example, the number $2.752 \cdot 10^3 = 2572$ can be written

$$\begin{aligned} 2752 &= 2^{11} + 2^9 + 2^7 + 2^6 \\ &= 2^{11} \cdot \left(1 + 2^{-2} + 2^{-4} + 2^{-5}\right) \\ &= 2^{11} \cdot \left(1 + (.01)_2 + (.0001)_2 + (.00001)_2\right) \\ &= 2^{11} \cdot (1.01011)_2 \quad . \end{aligned}$$

Altogether, we have, using $11 = (1011)_2$,

$$2752 = +(1.01011)_2^{(1011)_2} \; .$$

Thus, we have sign $s = +$. The exponent is $e - 127 = 11$ so that $e = 138 = (10001010)_2$. The fraction is $f = (01011000000000000000000)_2$. The entire 32 bit string corresponding to $2.752 \cdot 10^3$ then is:

$$\underbrace{1}_{s}\underbrace{10001010}_{e}\underbrace{01011000000000000000000}_{f} \quad .$$

For arithmetic, the standard mandates the rule: *the exact answer, correctly rounded.* For example, suppose `x`, `y`, and `z` are computer variables of type `float`, and the computer executes the statement `x = y / z;`. Let B and C be the numbers that correspond to the 32 bit strings `y` and `z` using the standard (2.2). Let A be the *exact mathematical* quotient, $A = B/C$. Let $\widehat{A}$ be the number closest to A that can be represented exactly in the single floating point format. The computer is supposed to set the bit string `x` equal to the bit string representing that $\widehat{A}$. For exceptions to this rule, see below.

The *exact answer correctly rounded* rule implies that the only error in floating point arithmetic comes from rounding the exact answer, A, to the nearest floating point number, $\widehat{A}$. Clearly, this rounding error is determined by the distance between floating point numbers. The worst case would be to put A in the middle of the largest gap between neighboring floating point numbers, B and B'. For a floating point number of the form $B = (1.f)_2 \cdot 2^p$, the next larger floating point number is usually $B' = (1.f')_2 \cdot 2^p$, where we get f' from f by adding the smallest possible fraction, which is 2^{-23} for 23 bit single precision fractions. The *relative* size of the gap between B and B' is, after some algebra,

$$\epsilon = \frac{B' - B}{B} = \frac{(1.f')_2 - (1.f)_2}{(1.f)_2} = \frac{2^{-23}}{(1.f)_2} .$$

The largest ϵ is given by the smallest denominator, which is $(1.0 \cdots 0)_2 = 1$, which gives $\epsilon_{max} = 2^{-23}$. The largest rounding error is half the gap size, which gives the single precision machine precision $\epsilon_{mach} = 2^{-24}$ stated above.

The 64 bit double precision floating point format allocates one bit for the sign, 11 bits for the exponent, and the remaining 52 bits for the fraction. Therefore its floating point precision is given by $\epsilon_{\text{mach}} = 2^{-53}$. Double precision arithmetic gives roughly 16 decimal digits of accuracy instead of 7 for single precision. There are 2^{11} possible exponents in double precision, ranging from 1023 to -1022. The largest double precision number is of the order of $2^{1023} \approx 10^{307}$. The largest single precision number is about $2^{126} \approx 10^{38}$. Not only is double precision arithmetic more accurate than single precision, but the range of numbers is far greater.

2.2.3 Exceptions

The extreme exponents, $e = 0$ and $e = 255$ ($e = 2^{11} - 1 = 2047$ for double precision) do not correspond to numbers. Instead, they have carefully engineered interpretations that make the IEEE standard distinctive. If $e = 0$, the value is

$$A = \pm 0.f \cdot 2^{-126} \text{ (single precision)}, \qquad A = \pm 0.f \cdot 2^{-2046} \text{ (double precision)}.$$

This feature is called *gradual underflow*. *Underflow* is the situation in which the result of an operation is not zero but is closer to zero than any floating point number. The corresponding numbers are called *denormalized*. Gradual underflow has the consequence that two floating point numbers are equal, $x = y$, if and only if subtracting one from the other gives exactly zero.

Introducing denormalized numbers makes sense when you consider the spacing between floating point numbers. If we exclude denormalized numbers then the smallest positive floating pont number (in single precision) is $A = 2^{-126}$ (corresponding to $e = 1$ and $f = 00 \cdots 00$ (23 zeros)) but the next positive floating point number larger than A is B, which also has $e = 1$ but now has $f = 00 \cdots 01$ (22 zeros and a 1). Because of the implicit leading 1 in $1.f$, the difference between B and A is 2^{22} times smaller than the difference between A and zero. That is, without gradual underflow, there is a large and unnecessary gap between 0 and the nearest nonzero floating point number.

The other extreme case, $e = 255$, has two subcases, `inf` (for infinity) if $f = 0$ and `NaN` (for Not a Number) if $f \neq 0$. C/C++ prints[1] "`inf`" and "`NaN`" when you print out a variable in floating point format that has one of these values. An arithmetic operation produces `inf` if the exact answer is larger than the largest floating point number, or `1/x` if $x = \pm 0$. (Actually $1/+0 = +$`inf` and $1/-0 =$ `-inf`). Invalid operations such as `sqrt(-1.)`, `log(-4.)`, produce `NaN`. Any operation involving a `NaN` produces another `NaN`. It is planned that f will contain information about how or where in the program the `NaN` was created but this is not standardized yet. Operations with `inf` are common sense: `inf` $+$ *finite* $=$ `inf`, `inf/inf` $=$ `NaN`, *finite*/`inf` $= 0.$, `inf` $-$ `inf` $=$ `NaN`.

A floating point arithmetic operation is an *exeption* if the result is not a normalized number in floating point format. The standard mandates that a hardware *flag* (a binary bit of memory in the processor) should be *set* (given the value 1) when an exception occurs. There should be a separate flag for the underflow, `inf`, and `NaN` exceptions. The programmer should be able to specify what happens when an exception flag is set. Either the program exection continues without interruption or an *exeption handler* procedure is called. The programmer should be able to write procedures that interface with the exception handler to find out what happened and take appropriate action. Only the most advanced and determined programmer will be able to do this. The rest of us have the worst of both: the exception handler is called, which slows the program execution but does nothing useful.

Many features of IEEE arithmetic are illustrated in Figure 2.1. Note that e^{204} gives `inf` in single precision but not in double precision because the range of values is larger in double precision. We see that `inf` and `NaN` work as promised. The main rule, "exact answer correctly rounded", explains why adding pairs of floating point numbers is commutative: the mathematical sums are equal so they round to the same floating point number. This does not force addition to be associative, which it is not. Multiplication is also commutitative but not associative. The division operator gives integer or floating point division depending on the types of the operands. Integer arithmetic *truncates* the result to the next lower integer rather than rounding it to the nearest integer.

```
// A program that explores floating point arithmetic in the IEEE
// floating point standard.  The source code is SourcesOfError.C.

#include <iostream.h>
#include <math.h>

int main() {

   float  xs, ys, zs, ws;  // Some single precision variables.
   double yd;              // A double precision variable.
```

[1] In keeping with the Microsoft pattern of maximizing incompatibility, the its compiler prints something different.

```
xs = 204.;         // Take an exponential that is out of range.
ys = exp(xs);
cout << "The single precision exponential of " << xs <<
        " is " << ys << endl;
yd = exp ( xs );   // In double precision, it is in range.
cout << "The double precision exponential of " << xs <<
        " is " << yd << endl;


zs = xs / ys;      // Divide a normal number by infinity.
cout << xs << " divided by " << ys  << " gives " << zs << endl;

ws = ys;           // Divide infinity by infinity.
zs = ws / ys;
cout << ws << " divided by " << ys << " gives " << zs << endl;

zs = sqrt( -1.) ;  // sqrt(-1) should be NaN.
cout << "sqrt(-1.) is " << zs << endl;

ws = xs + zs;      // Add NaN to a normal number.
cout << xs << " + " << zs << " gives " << ws << endl;

xs =        sin(1.); // Some generic single precision numbers.
ys = 100.  *sin(2.);
zs = 10000.*sin(3.);
float xsPys, ysPxs, xsPzs, zsPxs;  // xsPzx holds xs + zs, etc.
xsPys = xs + ys;
ysPxs = ys + xs;   // Try commuting pairs.
xsPzs = xs + zs;
zsPxs = zs + xs;
if ( ( xsPys == ysPxs ) && ( xsPzs == zsPxs ) )
   cout << "Adding " << xs << "  " << ys << " and "<< zs <<
           " in pairs commutes." << endl;
  else
    cout << "Adding " << xs << "  " << ys << " and "<< zs <<
            " in pairs does not commute." << endl;

float xsPysPzs, ysPzsPxs;     // Test for associativity.
xsPysPzs = ( xs + ys ) + zs;
ysPzsPxs = ( ys + zs ) + xs;
if ( xsPysPzs == ysPzsPxs )
   cout << "Adding " << xs << "  " << ys << " and "<< zs <<
           " is associative." << endl;
  else
```

```
    cout << "Adding " << xs << "  " << ys << " and "<< zs <<
            " is not associative." << endl;

int xi, yi; // Some integer variables.
xi =  9;    // Compute the quotient using integer
yi = 10;    // and floating point arithmetic.
zs = xi/yi;
ws = ( (float) xi ) / ( (float) yi ); // Convert, then divide.
cout << "Integer division of " << xi << " by " << yi << " gives " <<
        zs << ".  Floating point gives " << ws << endl;

return(0);

}
```

Figure 2.1: *A program that illustrates some of the features of arithmetic using the IEEE floating point standard.*

2.3 Truncation error

Truncation error is the error in analytical approximations such as

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} \, . \tag{2.3}$$

This is not an exact formula, but it can be a useful approximation. We often think of truncation error as arising from truncating a Taylor series. In this case, the Taylor series formula,

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \cdots \, ,$$

is truncated by neglecting all the terms after the first two on the right. This leaves the approximation

$$f(x+h) \approx f(x) + hf'(x) \, ,$$

which can be rearranged to give (2.3). Usually, truncation is the main source of error in numerical integration or solution of differential equations. The analysis of truncation error using Taylor series will occupy the next two chapters.

As an example, we take $f(x) = xe^x$, $x = 1$, and a variety of h values. The results are in Figure 2.3. In Chapter 3 we will see that in exact arithmetic (i.e. without rounding), the error would be roughly proportional th h for small h. These numbers were computed in double precision arithmetic and the effect of roundoff error is appearent in for the smallest h values. It is rare in a practical application that h would be so small that roundoff error would be a significant factor in the overall error.

h	.3	.01	10^{-5}	10^{-8}	10^{-10}
$\widehat{f'}$	6.84	5.48	5.4366	5.436564	5.436562
e_{tot}	1.40	$4.10 \cdot 10^{-2}$	$4.08 \cdot 10^{-5}$	$-5.76 \cdot 10^{-8}$	$-1.35 \cdot 10^{-6}$

Figure 2.2: Estimates of $f'(x)$ using (2.3). The error is e_{tot}, which is a combination of truncation and roundoff error. Roundoff error is apparent only in the last two estimates.

n	1	3	6	10	20	50
x_n	1	1.46	1.80	1.751	1.74555	1.745528
e_n	$-.745$	$-.277$	$5.5 \cdot 10^{-2}$	$5.9 \cdot 10^{-3}$	$2.3 \cdot 10^{-5}$	$1.2 \cdot 10^{-12}$

Figure 2.3: Iterates of $x_{n+1} = \ln(y) - \ln(x_n)$ illustrating convergence to a limit that satisfies the equation $xe^x = y$. The error is $e_n = x_n - x$. Here, $y = 10$.

2.4 Iterative Methods

Suppose we want to find a number, A, by solving an equation. Often it is impossible to find a formula for the solution. Instead, *iterative methods* construct a sequence of approximate solutions, A_n, for $n = 1, 2, \ldots$. Hopefully, the approximations *converge* to the right answer: $A_n \to A$ as $n \to \infty$. In practice, we must stop the iteration process for some large but finite n and accept that A_n as the approximate answer.

For example, suppose we have a $y > 0$ and we want to find x with $xe^x = y$. There is not a formula for x, but we can write a program to carry out the iteration: $x_1 = 1$, $x_{n+1} = \ln(y) - \ln(x_n)$. The numbers x_n are the *iterates*. If the limit of the iterates exists, $x_n \to x$ exists as $n \to \infty$, then that x should be a *fixed point* of the iteration, i.e. $x = \ln(y) - \ln(x)$. Figure 2.4 demonstrates the convergence of the iterates in this case. If we stop after 10 iterations, the 20^{th} iterate has an error $e_{20} \approx 2.3 \cdot 10^{-5}$, which might be small enough, depending on the application.

2.5 Statistical error in Monte Carlo

Monte Carlo is a computational technique that uses random numbers as a computational tool. For example, we may be able to use the computer random number generator to create a sequence of random variables X_1, X_2, $\ldots$, with the same distribution. We could use these samples to estimate the average, A, by the computer generated sample mean

$$A \approx \widehat{A} = \frac{1}{n} \sum_{k=1}^{n} X_k \; .$$

Statistical error is the difference between the estimate $\widehat{A}$ and the correct answer, A. Monte Carlo statistical errors typically would be larger than roundoff or

n	10	100	10^4	10^6	10^6	10^6
$\widehat{A}$	.603	.518	.511	.5004	.4996	.4991
error	.103	$1.8 \cdot 10^{-2}$	$1.1 \cdot 10^{-2}$	$4.4 \cdot 10^{-4}$	$-4.0 \cdot 10^{-4}$	$-8.7 \cdot 10^{-4}$

Figure 2.4: Statistical errors in a demonstration Monte Carlo computation.

truncation errors. This makes Monte Carlo a method of last resort, to be used only when other methods are not available.

We illustrate the behavior of statistical error with a simple example. Here the X_k are independent random variables with the same probability distribution that have average value $A = .5$. Figure 2.5 gives the approximations and statistical errors for several values of n. The value $n = 10^6$ is repeated several times to illustrate the fact that statistical error is random (see Chapter mc*mc for a clarification of this). Note the size of the errors even with a million samples and compare these to the errors in Figures **??** and **??**.

2.6 Error amplification and unstable algorithms

Errors can be amplified during sequence of computational steps. For example, suppose we are using the divided difference (2.3) to approximate $f'(x)$ using approximate values $\widehat{f}_1 = f(x) + e_1$ and $\widehat{f}_2 = f(x+h) + e_2$. This is practically inevitable, since we cannot evaluate f exactly. Using these values gives

$$\begin{aligned} \widehat{f'} &= \frac{f(x+h) - f(x)}{h} + \frac{e_2 - e_1}{h} + e_r \\ &= f'(x) + e_{tr} + e_{am} + e_r \; . \end{aligned}$$

Here e_{tr} is the truncation error

$$e_{tr} = \frac{f(x+h) - f(x)}{h} - f'(x) \; ,$$

e_r is the roundoff error in evaluating $(\widehat{f}_2 - \widehat{f}_1)/h$, in floating point, given the floating point values $\widehat{f}_1$ and $\widehat{f}_1$. The remaining error, and often the largest, is the amplification error

$$e_{am} = \frac{e_2 - e_1}{h} \quad .$$

We see that the errors e_1 and e_2 are *amplified* by the factor $1/h$, which is quite large if h is small. For example, if $h = .01$, then the error in evaluating f' is likely to be a hundred times larger than the error in approximating f.

This point may be put in a more technical form: addition or subtraction with cancellation can amplify relative errors in the operands. This is only true for relative error, which is a major reason relative error is important. Consider the operations $A = B + C$ and $\widehat{A} = \widehat{B} + \widehat{C}$. We suppose for simplicity that the sum is done exactly, so that all the error in $\widehat{A}$ is due to error in the operands, $\widehat{B}$ and $\widehat{C}$. The absolute and relative errors, e_A, ϵ_A, etc. are defined by (see (2.1):

$$\widehat{A} = A + e_A = (1+\epsilon_A)A \;\; , \quad \widehat{B} = B + e_B = (1+\epsilon_B)B \;\; , \quad \widehat{C} = C + e_C = (1+\epsilon_C)C \;\; .$$

With absolute errors there are no surprises: $e_A = e_B + e_C$. We do not know the signs of e_B and e_C, but we can write $|e_A| \leq |e_B| + |e_C|$. This says that the magnitude of the absolute error of the result is at most the sum of the absolute errors in the operands. If floating point arithmetic were to produce high absolute accuracy, which it does not, we could perform tens of thousands of single precision operations always come out with high absolute accuracy.

With relative error, *amplification* through *cancellation* is possible. The relative error of A may be larger or much larger than sum of the relative errors in the operands. Some algebra leads to the formula

$$|\epsilon_A| = |\epsilon_B W_B + \epsilon_C W_C| \leq \max(|\epsilon_B|\,, |\epsilon_C|)(|W_B| + |W_C|) \;\; , \tag{2.4}$$

where

$$W_B = \frac{B}{B+C} \;\; , \quad W_C = \frac{C}{B+C} \;\; , \quad W_B + W_C = 1 \; .$$

If B and C have the same sign, then $|W_B| = |W_C| = 1$, and (2.4) shows that $|\epsilon_A|$ is less than both $|\epsilon_A|$ and $|\epsilon_A|$: the relative error has not been amplified. On the other hand, if $B = 1.01$ and $C = -1$, then $|W_B| \approx |W_C| = 100$: the relative error may be amplified by a factor of 100.

Cancellation is the situation $A = B + C$ (or $A = B - C$) and $|A| < |B| + |C|$. Addition has cancellation if the operands B and C have opposite signs. The previous paragraph shows that addition and subtraction amplify relative errors only if there is cancellation. A similar argument shows that multiplication and division produce high relative accuracy in (almost) all cases, no cancellation there. Performing $1,000$ arithmetic operations will generate a relative error not larger than $1000\epsilon_{mach}$, and probably much smaller, if there is no cancellation in the additions and subtractions.

It is possible to lose many decimal places of accuracy in a single subtraction. This is called *catistrophic cancellation*. More insideous is the possiblilty of small error amplification at each stage of a long calculation. For example, if we start with a relative error of 10^{-16} characteristic of double precision arithmetic and amplify by 5% in each of 1000 steps, then we have well over 100% relative error at the end. A computational algorithm is called *unstable* if it loses accuracy because of cancellation, either catastrophically or, which is worse, gradually. One of the main uses of mathematical analysis in scientific computing is in understanding the stability of multistage computational algorithms.

Inaccuracy because of an unstable algorithm very is hard 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.

2.7 Condition number and ill conditioned problems

The *condition number* of a computational problem measures the sensitivity of the answer to small changes in the data, in relative terms. The condition number limits the accuracy we can expect to achieve in a computation even with the best algorithm. If a problem is sufficiently *ill conditioned*, the condition number is too large, then even inevativle rounding errors in the input may lead to unacceptable errors in A.

We write $A(x)$ to indicate that the answer, A, depends on the data, x. In the simplest case, both A and x are single numbers, and $A(x)$ is a differentiable function of x. A perturbation in the data, Δx, will lead to a change in the answer, $\Delta A = A(x + \Delta x) - A(x)$. If Δx is small enough, $\Delta A/\Delta x \approx a'(x)$, so that

$$\Delta A \approx A'(x)\Delta x \ .$$

The absolute change in the answer is related to the absolute change in the data by the derivative, A'.

The relative change of the answer is related to the relative change in the data by the condition number, κ:

$$\frac{\Delta A}{A} \approx \kappa \frac{\Delta x}{x} \quad . \tag{2.5}$$

For small Δx, we may use the derivative approximation to get, after a bit of algebra,

$$\kappa = \frac{\Delta A/A}{\Delta x/x} \approx A' \cdot \frac{x}{A} \quad .$$

Everybody's definition of κ differs from this one in that nobody allows the condition number to be negative; they use the absolute value of this κ:

$$\kappa = \left| A'(x) \frac{x}{A(x))} \right| \quad . \tag{2.6}$$

This has the property that the ratio of the relative size of $|\Delta A|$ to the relative size of Δx is not very sensitive to Δx, provided Δx is small enough:

$$\frac{|\Delta A| / |A|}{|\Delta x| / |x|} \approx \kappa \ , \quad \text{for small } \Delta x. \tag{2.7}$$

In most real problems, both the answer and the data are more complex than a single number. We still write Δx and ΔA for perturbations in the data and answer, but now norms $\|\Delta x\|$ and $\|\Delta A\|$ measure the size of the perturbation. We will discuss norms in Chapter **??**. A measure of the relative size of Δx is $\|\Delta x\| \,/\, \|x\|$. Unlike the univariate case (2.7), here $\|\Delta A\|$ depends on the direction of Δx, not just its size. The condition number is taken to be the worst case sensitivity:

$$\kappa = \max_{\Delta x \text{ small}} \frac{\|\Delta A\| \,/\, \|A\|}{\|\Delta x\| \,/\, \|x\|} \quad .$$

```
double x = ( double(1) ) / 7, y1, y2;
y1 = x + x + x + x + x;
y2 = 5*x;
if ( y1 == y2 ) cout << "Equal"     << endl;
else            cout << "Not equal" << endl;
```

Figure 2.5: *A code fragment in which roundoff error can lead to numbers that should be equal in exact arithmetic not being equal in floating point.*

We rewrite this to resemble (2.5) by saying that κ is the smallest number so that

$$\frac{\|\Delta A\|}{\|A\|} \leq\approx \kappa \frac{\|\Delta x\|}{\|x\|} \quad , \quad \text{for all small enough } \Delta x. \tag{2.8}$$

Floating point rounding makes it is inevatible that $|\Delta x/x|$ is of the order of ϵ_{mach}. Therefore, if $\kappa > 1/\epsilon_{mach}$, than (2.5) or (2.8) show that ΔA will be as large as A, total error. Unfortunately, condition numbers as large as 10^7 or 10^{16} occur in practice.

Computational errors caused by large condition number cannot be cured by changing the computational algorithm. None of our discussion of condition number referred to any particular computational algorithm. If a problem has

Anyone engaged in scientific computing should keep conditioning in mind. He or she should be aware that a problem may be ill conditioned, and try to estimate the condition number, at least roughly. High quality numerical software often includes estimators of condition number with the results, which should be checked. Even in complex situations, it may be possible to find an analogue of (2.6) by differentiation. Chapter **??** has some examples.

2.8 Software tips

2.8.1 Floating point numbers are (almost) never equal

Because of inexact floating point arithmetic, two numbers that should be equal in exact arithmetic often are not equal in the computer. For example, the code in Figure 2.5 prints `Not equal`. In general, an equality test between two variables of type `float` or `double` is a mistake. Even worse, if we have `y1 = y2;` and then do not reassign either `y1` or `y2`, many lines later the test `( y1 == y2 )` may evaluate to `false`. For example, on a Pentium 4 chip there are 80 bit registers for holding variables of type `double`. The computer often holds a variable in a register rather than returning it to memory to save time. If the variable is returned to memory, only the 64 bits of the IEEE standard are stored, the remaining 16 bits being lost. If `y1` but not `y2` was returned to memory, then they may no longer be equal.

```
double tStart, tFinal, t, dt;
int    n;
tStart = . . .  ;   // Some code that determines the start
tFinal = . . .  ;   // and ending time and the number of
n      = . . .  ;   // equal size steps.
dt     = ( tFinal - tStart ) / n;  // The size of each step.
while ( t = tStart, t < tFinal, t+= dt )
   { . . . }        // Body of the loop does not assign t.
```

Figure 2.6: *A code fragment illustrating a pitfall of using a floating point variable to regulate a* `while` *loop.*

```
double tStart, tFinal, t, dt;
int    n,    , i;
tStart = . . .  ;   // Some code that determines the start
tFinal = . . .  ;   // and ending time and the number of
n      = . . .  ;   // equal size steps.
dt     = ( tFinal - tStart ) / n;  // The size of each step.
while ( i = 0, i < n, i++ )
   { t = tStart + i*dt;  // In case the value of t is needed
          . . . }        // in the loop body.
```

Figure 2.7: *A code fragment using an integer variable to regulate the loop of Figure 2.6.*

A common mistake in this regard is to use floating point comparisons to terminate a loop. Figure 2.6 illustrates this. In exact aritmetic this would give the desired n iterations. Because of floating point arithmetic, after the n^{th} iteration, the variable `t` may be equal to `tFinal` but is much more likely to be above or below roundoff error. Unfortunately, it is impossible to predict which way the roundoff error will go. Therefore, we do not know whether this code will execute the `while` loop body n or $n+1$ times.

To guarantee n executions of the `while` loop body, use integer (fixed point) variables, as illustrated in Figure 2.7

2.8.2 Plotting data curves

Careful visualization is a key step in understanding any data, particularly the results of a numerical computation. Time spent making plots carefully and thoughtfully is usually rewarded. The first rule is that a plot, like a piece of

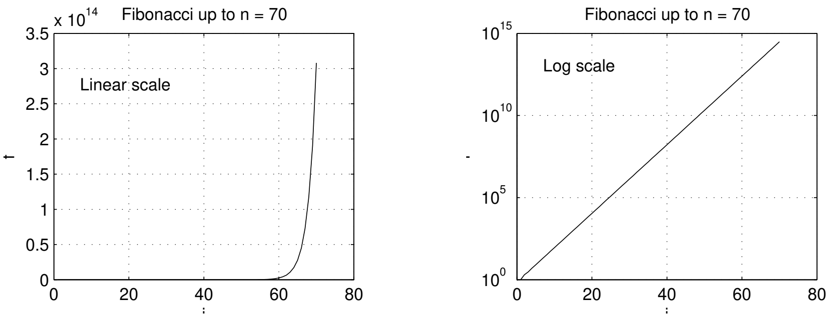

Figure 2.8: *Plots of the first n Fibonacci numbers, linear scale on the left, log scale on the right*

code, should be self documenting. In coding, we take the time to type comments, format carefully, etc. In plots, we take the time to label axes and print titles with relevant plot parameters. Otherwise, a week later nobody will know which plot is which.

Careful scaling can make the data more clear. If you have a function $f(x)$ with values in the range from 1.2 to 1.22, the Matlab `plot` function will lable the vertical axis from 0 to 2 and plot the data as a nearly horizontal line. This is illustrated in Figure 2.8, where the first 70 Fibonacci numbers, are plotted on a linear scale and a log scale. The Fibonacci numbers, f_i, are defined by $f_0 = f_1 = 1$, and $f_{i+1} = f_i + f_{i-1}$, for $i \geq 1$. On the linear scale, f_1 through f_{57} sit on the horizontal axis. The log plot lets us see how big each of the 70 numbers is. It also makes it clear that $\log(f_i)$ is nearly proportional to i. If $\log(f_i) \approx a + bi$, then $f_i \approx cd^i$. It is common to understand the data by plotting it in various ways.

The Matlab script that made the plots of Figure 2.8 is in Figure 2.9. Normally, one need not comment up throwaway scripts like this. The only real parameters are n, the largest i value, and whether the plot is on a linear or log scale. Both of those are recorded in the plot. Note the convenience and clarity of not hard wiring $n = 70$. It would take just a moment to make plots up to $n = 100$.

2.9 Further reading

The idea for starting a book on computing with a discussion of sources of error comes from the book *Numerical Methods and Software* by David Hahaner, Cleve Moler, and Steve Nash. Another interesting version is in *Scientific Computing* by Michael Heath. My colleague, Michael Overton, has written a nice short book *IEEE floating point arithmetic.*

```
  % Matlab code to generate and plot Fibonacci numbers.

clear f     % If you decrease the value of n, the plots still work.
n    = 70;  % The number of Fibonacci numbers to compute.
fi   = 1;   % Start with f0 = f1 = 1, as usual.
fim1 = 1;
f(1) = fi;  % Record f(1) = f1.
for i = 2:n
   fip1 = fi + fim1;  % f(i+1) = f(i) + f(i-1) is the recurrence
   fim1 = fi;         % relation that defines the Fibonacci numbers.
   fi   = fip1;
   f(i) = fi;         % Record f(i) for plotting.
end

plot(f)
xlabel('i')           % The horizontal and vertical axes are
ylabel('f')           % i and f respectively.
topTitle = sprintf('Fibonacci up to n = %d',n); % Put n into the title.
title(topTitle)
text(n/10, .9*f(n), 'Linear scale');
grid                  % Make it easier to read values in the plot.
set ( gcf, 'PaperPosition', [.25 2.5 3.2 2.5]);
                   % Print a tiny image of the plot for the book.
print -dps FibLinear_se
```

Figure 2.9: *A code fragment using an integer variable to regulate the loop of Figure 2.6.*

2.10 Exercises

1. For each of the following, indicate whether the statement is true or false and explain your answer.

   a. The `for` loop in line 4 below is will be executed 20 times no matter what x is generated by `rand()`. Here `float rand()` generates random numbers uniformly distributed in the interval $[0, 1]$.

```
float  x = 100*rand() + 2;               // line 1
int    n = 20;                           // line 2
float dy = x/n;                          // line 3
for ( float y = 0; y < x; y += dy; ) {   // line 4
     body does not change x, y, or dy }
```

   b. The `for` loop in line 4 always will always be executed at least 20 times.

   c. The `for` loop in line 4 never will be executed more than 20 times.

   d. There is a function, $f(x)$ that we wish to compute numerically. We know that for x values around 10^{-3}, f is about 10^5 and f' is about 10^{10}. This function is too ill conditioned to compute in single precision.

2. Show that in the IEEE floating point standard with any number of fraction bits, ϵ_{mach} is the largest floating point number, ϵ, so that $1 + \epsilon$ gives 1 in floating point arithmetic. Whether this is mathematically equal to the definition in the text depends on how ties are broken in rounding, but the difference between the two definitions is irrelevent (show this).

3. Starting with the declarations

```
      float x, y, z, w;
const float oneThird = 1/ (float) 3;  // const means these can never
const float oneHalf  = 1/ (float) 2;  // be reassigned
```

   we do lots of arithmetic on the variables `x, y, z, w`. In each case below, determine whether the two arithmetic expressions result in the same floating point number (down to the last bit) as long as no `NaN` or `inf` values or denormalized numbers are produced.

   a.

```
( x * y ) + ( z - w )
( z - w ) + ( y * x )
```

   b.

```
( x +   y ) + z
  x + ( y +   z )
```

   c.

```
  x * oneHalf + y * oneHalf
( x + y ) * oneHalf
```

d.
```
  x * oneThird + y * oneThird
( x + y ) * oneThird
```

4. The *fibonacci numbers*, f_k, are defined by $f_0 = 1$, $f_1 = 1$, and

$$f_{k+1} = f_k + f_{k-1} \tag{2.9}$$

for any integer $k > 1$. A small perturbation of them, the *pib numbers* ("p" instead of "f" to indicate a perturbation), p_k, are defined by $p_0 = 1$, $p_1 = 1$, and

$$p_{k+1} = c \cdot p_k + p_{k-1}$$

for any integer $k > 1$, where $c = 1 + \sqrt{3}/100$.

a. Make one plot of $\log(f_n)$ and $\log(p_n)$, or plot the f_n and p_n together on a log scale, as a function of n. On the plot, mark $1/\epsilon_{mach}$ for single and double precision arithmetic. This can be useful in answering the questions below.

b. Rewrite (2.9) to express f_{k-1} in terms of f_k and f_{k+1}. Use the computed f_n and f_{n-1} to recompute f_k for $k = n-2, n-3, \ldots, 0$. Make a plot of the difference between the original $f_0 = 1$ and the recomputed $\widehat{f}_0$ as a function of n. What n values result in no accuracy for the recomputed f_0? How do the results in single and double precision differ?

c. Repeat b. for the pib numbers. Comment on the striking difference in the way precision is lost in these two cases. Which is more typical? *Extra credit*: predict the order of magnitude of the error in recomputing p_0 using what you may know about recurrence relations and what you should know about computer arithmetic.

5. The binomial coefficients, $a_{n,k}$, are defined by

$$a_{n,k} = \begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!(n-k)!}$$

To compute the $a_{n,k}$, for a given n, start with $a_{n,0} = 1$ and then use the recurrence relation $a_{n,k+1} = \frac{n-k}{k+1} a_{n,k}$.

a. For a fixed of n, compute the $a_{n,k}$ this way, noting the largest $a_{n,k}$ and the accuracy with which $a_{n,n} = 1$ is computed. Do this in single and double precision. Why is it that roundoff is not a problem here as it was in problem 4?

b. Use the algorithm of part (a) to compute

$$E(k) = \frac{1}{2^n} \sum_{k=0}^{n} k a_{n,k} = \frac{n}{2} \quad . \tag{2.10}$$

In this equation, we think of k as a random variable, the number of heads in n tosses of a fair coin, with $E(k)$ being the expected value of k. This depends on n. Write a program without any safeguards against overflow or zero divide (*this time only!*)[2]. Show (both in single and double precision) that the computed answer has high accuracy as long as the intermediate results are within the range of floating point numbers. As with (a), explain how the computer gets an accurate, small, answer when the intermediate numbers have such a wide range of values. Why is cancellation not a problem? Note the advantage of a wider range of values: we can compute $E(k)$ for much larger n in double precision. Print $E(k)$ as computed by (2.10) and $M_n = \max_k a_{n,k}$. For large n, one should be `inf` and the other `NaN`. Why?

c. For fairly large n, plot $a_{n,k}$ as a function of k for a range of k chosen to illuminate the interesting "bell shaped" behavior of the $a_{n,k}$ near their maximum.

d. (*Extra credit, and lots of it!*) Find a way to compute

$$S(k) = \sum_{k=0}^{n} (-1)^k \sin(2\pi \sin(k/n)) a_{n,k}$$

with good relative accuracy for large n. This is very hard, so don't spend too much time on it.

6. What day is it

[2] One of the purposes of the IEEE floating point standard was to *allow* a program with overflow or zero divide to run and print results

Chapter 3

NumericalAnalysis

Manipulation of Taylor series expansions is one of the most common techniques in scientific computing. Most computational methods for differentiation, integration, and solution of differential equations are based directly on Taylor series. Series expansions not only lead to computational algorithms, but they also tell us how accurate these algorithms should be and predict properties of the error that are the basis for code validation and sophisticated adaptive computational software.

In this chapter we apply the Taylor series method for three specific problems, differentiation, integration, and interpolation. Numerical differentiation is the problem of finding (as accurately as necessary) a derivative of a function, given several values of the function itself. Numerical integration is the problem of computing the integral. Interpolation is the problem of evaluating a function at some value of its arguments given function values at other values of the arguments.

For all these purposes, we use Taylor series as *asymptotic expansions*. To see what this means, consider the expansion

$$e^x = 1 + x + \frac{1}{2}x^2 + \cdots + \frac{1}{n!}x^n + \cdots \quad . \tag{3.1}$$

One could use this formula to compute e^2 by taking $x = 2$ and adding up enough terms to get the desired accuracy. That is not what we do here. Instead, we fix the number of terms, for example

$$e^x \approx 1 + x + \frac{1}{2}x^2 \; , \tag{3.2}$$

and ask about the range of x values for which this approximation is accurate enough. In real problems, Taylor series beyond the first few may be difficult or impossible to compute. Fortunately, most of the methods in this chapter apply to any function that has a Taylor series expansion, in the asymptotic sense explained below. We rarely need explicit expressions for the coefficients.

As we stated in Chapter 2, errors that arise from taking just a few terms of an infinite Taylor series are called *truncation errors*. For example, we truncate the infinite series (3.1) to get the three term approximation (3.2). This chapter is about truncation error. We generally neglect roundoff error, since truncation error is usually larger. We will have to revisit this assumption if our problem is ill posed, if our computational algorithm is unstable, or if the step size is too small.

For each of the problems discussed we will start with simple approximations of modest accuracy and proceed to methods of increasing complexity and accuracy. The simplest methods may be adequate for casual computing. Why spend more time optimizing a code than possibly could be saved in a computation that takes the computer less than a second? However, the basic operations of integration and differentiation are often at the hearts of computational algorithms whose running times are a serious concern; time discovering and implementing more complex and accurate approximations may be repaid amply.

We work with a *step size*, which is generically called h but may have other names in specific contexts. The approximations become more accurate as h becomes smaller. The rate of improvement is the *order of accuracy*. More accurate approximations generally lead to faster computations. For example, in estimating $\int_a^b f(x)dx$, the region of integration, $[a, b]$ is divided into *panels* of size h. The smaller h, the more panels and longer it takes the computer to process them all. If a sophisticated integration method achieves 1% error with $h = .1$ while the simple one requires $h = .01$, the sophisticated method will be, maybe, five times faster – twice the cost per panel but ten times fewer panels. Whether this factor of five speedup is worth days of extra analysis and programming depends on the situation.

We will often use the phrase "for sufficiently small h" without clarifying how small is small enough. One answer might be that h should be small compared to "natural length scales" in the problem. Unfortunately, this question is too problem dependent to settle in a simple general way. When h is small enough, we say that h is in the *asymptotic range*, the range in which an asymptotic expansion gives useful information. It is often clear from the problem roughly how large the asymptotic range is likely to be, and that range is usually large enough to cover practical h values. If a code is completely unable to produce results consistent with an asymptotic error analysis, it is generally not because the asymptotic range is inaccessible. Look instead for a bug in the code, in the method, or in the error analysis. See Software tip **??** for more on this point.

3.1 Software tips

3.1.1 Write flexible and verifiable codes

The main way to verify correctness of a code it to check that it gets the right answer. For this reason, it is helpful to build codes general enough to calculate a variety of answers, some of which we already know.

3.1.2 Report failures

A code should not fail silently. If a procedure is unable to do what it is asked to do, it must report this in some way. Most procedures used in a computation can fail on some problems they would be exposed to. Most user want know that the answer is wrong if it is. As a principle of programming practice, every procedure should have some way to communicate failure, either to the calling procedure (preferred for serious codes) or directly to the user.

There are several ways to report failure. The simplest is just to print an error message, such as:

```
cout << "Procedure Integrate failed because n = " << n
     << " was larger than N_MAX = " << N_MAX << endl;
```

Notice that the message told the user the name of the routine, the reason for the problem, and the offending number.

3.2 Exercises

1. We want to study the function

$$f(x) = \int_0^1 \cos\left(xt^2\right) dt \quad . \tag{3.3}$$

Step 1: Write a procedure (or "method") to estimate $f(x)$ using a panel integration method with uniformly spaced points. The procedure should be well documented, robust, and clean. Robust will mean many things in future assignments. Here is means: (i) that it should use the correct number of panels even though the points t_k are computed in inexact floating point arithmetic, and (ii) that the procedure will return an error code and possibly print an error message if one of the calling arguments is out of range (here, probably just $n \leq 0$). It should take as inputs x and $n = 1/\Delta t$ and returns the approximate integral with that x and Δt value. This routine should be written so that another person could easily substitute a different panel method or a different integrand by changing a few lines of code.

Step 2: Verify the correctness of this procedure by checking that it gives the right answer for small x. We can estimate $f(x)$ for small x using a few terms of its Taylor series. This series can be computed by integrating the Taylor series for $\cos(xt^2)$ term by term. This will require you to write a "driver" that calls the integration procedure with some reasonable but not huge values of n and compares the returned values with the Taylor series approximation.

Step 3: With $x = 1$, do a convergence study to verify the second order accuracy of the trapezoid rule and the fourth order accuracy of Simpson's rule. This requires you to write a different driver to call the integration procedure with several values of n and compare the answers in the manner of a convergence study. Once you have done this for the trapezoid rule, it should take less than a minute to redo it for Simpson's rule. This is how you can tell whether you have done Step 1 well.

Step 4: Write a procedure that uses the basic integration procedure from Step 1, together with Richardson error estimation to find an n that gives $f(x)$ to within a specified error tolerance. The procedure should work by repeatedly doubling n until the estimated error, based on comparing approximations, is less than the tolerence given. This routine should be robust enough to quit and report failure if it is unable to achieve the requested accuracy. The input should be x and the desired error bound. The output should be the estimated value of f, the number of points used, and an error flag to report failure. Before applying this procedure to the panel integration procedure, apply it to the fake procedure `fakeInt.c` or `fakeInt.C`. Note that these testers have options to make the Richardson program fail or succeed. You should try it both ways, to make sure the robustness feature of your Richardson procedure works. Include with your homework, output illustrating the behavior of your Richardson procedure when it fails.

Step 5: Here is the "science" part of the problem, what you have been doing all this coding for. We want to test an approximation to f that is supposed to be valid for large x. The supposed approximation is:

$$f(x) \sim \sqrt{\frac{\pi}{8x}} + \frac{1}{2x}\sin(x) - \frac{1}{16x^2}\cos(x) + \cdots \quad . \tag{3.4}$$

Make a few plots showing f and its approximations using one, two and all three terms on the right side of (2) for x in the range $1 \leq x \leq 1000$. In all cases we want to evaluate f so accurately that the error in our f value is much less than the error of the approximation (2). Note that even for a fixed level of accuracy, more points are needed for large x. Why?