Section 1

1 Introduction to the course

These are notes for a one semester course on Scientific Computing at the beginning graduate level. The goal is to help students use mathematical, software, and hardware resources to do numerical computations. Each section focuses on a mathematical topic. Material on software and hardware is included, as long as it can be illustrated with examples connected to the mathematical topic. Scientific computing is a combination of mathematics and computer science.

Prerequisites

The hard prerequisites are linear algebra, multivariable calculus with an appreciation of the role of inequalities and "big Oh" notation, and basic Python. A course in probability (probability densities, PDF for multi-component random variables, conditional and marginal probability) will make it much easier to follow the sections on Monte Carlo and stochastic gradient descent. Newtonian mechanics (freshman physics) will be useful for understanding the sections on dynamics and differential equations. We will make some appeals to material typically in a class called something like "discrete math" or "analysis of algorithms".

2 Introduction to the Section

Thus course explains many things you can do with computer calculation. The first section discusses what basic computations actually do.

The arithmetic in computer number crunching is not done exactly. Consider the `python` commend `z = x + y`. The computer variables `x` and `y` correspond to mathematical numbers x and y. The operation produces a computer variable `z` with a mathematical value z. But computer floating point arithmetic is not exact

`z = x + y` has roundoff error, so $x + y \neq z$.

3 Relative accuracy, condition number

All (almost all) scientific computations are approximations. Quantifying error in computation is necessary but surprisingly subtle. Simple approaches involving *relative error* and *condition number* are very useful, but have drawbacks.

More sophisticated treatments of error are cumbersome and still not completely satisfactory.

There is a distinction between *absolute* and *relative* error. Suppose A is the number being approximated and $\widehat{A}$ is the approximation. One example would be $A = \pi$ and $\widehat{A} = 3.142$. We define absolute and relative errors to mean

$$E_{abs} = \widehat{A} - A \ , \quad E_{rel} = \frac{\widehat{A} - A}{A} \ . \tag{1}$$

For example, the errors for the approximation of $\pi = 3.14159265\cdots$ are

$$E_{abs} = 3.142 - 3.14159265 = .000508$$

$$E_{rel} = \frac{3.142 - 3.14159265}{3.14159265} = .000162$$

Relative error may be more useful than absolute error if A is very large or small. For example, an absolute error $E_{abs} = 1$ is large if $A = 1$ but a tenth of a percent if $A = 1000$. The relative error is *dimensionless*, which means that the number does not change if A is expressed in different units.

The *condition number* of a mathematical problem measures how sensitive the answers is to the data. Suppose the data consists of one number, x, which is perturbed to a nearby number $x + \Delta x$. The corresponding answer changes by $\Delta A = A(x + \Delta x) - A(x)$. The condition number measures the relative change in the answer, related to the relative change in the data. The simple or absolute ratio of changes is

$$\frac{\Delta A}{\Delta x} \approx A'(x) \ .$$

The relative change, or ratio of relative changes, is (using the derivative approximation $\Delta A \approx A'(x)\Delta x$)

$$\frac{\dfrac{\Delta A}{A}}{\dfrac{\Delta x}{x}} \approx \frac{xA'(x)}{A(x)} \ .$$

It is traditional to define *condition number* as the absolute value of the right side, so a problem cannot have a negative condition number:

$$\kappa(x) = \left| \frac{xA'(x)}{A(x)} \right| \ . \tag{2}$$

4 Floating point arithmetic

It is impractical or impossible for the computer to do arithmetic operations (addition, multiplication, division) exactly. Instead, they are done, with a few exceptions, to within *floating point precision*. The *floating point standard*[1] specifies

[1] Technically, it is IEEE standard number 754. The Institute for Electrical and Electronics Engineers, the *IEEE*, issues standards for things to do with electricity, ranging from power

that a floating point operation should either produce an *exception*, or produce error due only to *rounding*.

The floating point standard can be expressed in simple mathematical terms. The *floating point numbers* (of a given *precision*) form a subset of the set of all real numbers. We denote this set

$$\mathcal{F}l \subset \mathbb{R} \, .$$

Rounding a real number x means finding "the" floating point number $\widehat{x} \in \mathcal{F}l$ closest to x. This is also written as $fl(x)$, to represent the "floating point" part of x. That is

$$\widehat{x} = fl(x) = \arg \min_{y \in \mathcal{F}l} |y - x| \ . \tag{3}$$

There are some real numbers x that have two closest numbers $\widehat{x}_{\pm} \in \mathcal{F}l$. The detailed IEEE standard specifies how such ties may be resolved. The *machine precision*, ϵ_{mach}, is a number that characterizes the relative error of rounding. More precisely, there is a small positive number f_{min} and a large positive number f_{max} so that

$$f_{min} \leq |x| \leq f_{max} \implies \frac{|\widehat{x} - x|}{|x|} \leq \epsilon_{mach} \, . \tag{4}$$

The numbers $x \in \mathbb{R}$ with $f_{min} \leq |x| \leq f_{max}$ are "inside the range of normal floating point arithmetic". The rounded number $\widehat{x}$ is said to be *normalized*. The numbers ϵ_{mach}, f_{min} and f_{max} depend on the *precision* of the arithmetic. Greater precision means smaller ϵ_{mach}, and also greater *range*, which means that f_{max} is larger and f_{min} closer to zero. The table below lists the most commonly used precisions.

precision	bits/word	type, in numpy	ϵ_{mach}	f_{min}	f_{min}
double	64	`np.float64`	$2^{-53} = 1.1 \cdot 10^{-16}$	$2^{-1022} = 10^{-123}$	$2^{1023} = 9 \cdot 10^{307}$
single	32	`np.float32`	$2^{-24} = 6 \cdot 10^{-8}$	$2^{-126} = 1.2 \cdot 10^{-38}$	$2^{127} = 1.7 \cdot 10^{38}$
quad	128	`np.float128`	$2^{-113} = 10^{-16}$	$2^{-16382} = 10^{-123}$	$2^{16383} = 10^{123}$
half	16	`np.float16`	$2^{-11} = 4.1 \cdot 10^{-4}$	$2^{-14} = 6.1 \cdot 10^{-5}$	$2^{15} = 3.3 \cdot 10^{4}$

This is most of what a casual scientific computing person needs to know about floating point arithmetic. Python uses 64 bit double precision floating point by default. You need a reason to use something else. You might use 32 bit single precision precision if you need less rounding error, which is rare but can happen if some part of the computation is very ill conditioned. Quad precision also has a greater normalized range, as you can see by comparing f_{min} and f_{max} from the "double" and "quad" rows of the accuracy table.

A floating point number $x \in \mathcal{F}l$ is represented by the bits of a *word* of computer memory. A double precision floating point number is represented by a 64 bit word, and a single precision number is represented a 32 bit word. Half

transmission lines to computer software. The current IEEE floating point standard is 74 pages of exhaustive detail. Most computing hardware (chips, GPUs, etc.) follows the floating point standard closely, if not exactly.

precision and quad precision have 61 bit and 128 bit words respectively. For each precision, there is one *sign* bit and some number of *exponent* bits and *fraction* bits. Double precision has 11 exponent bits and 52 fraction bits. Altogether, counting the sign bit, exponent bits, and fraction bits, a double precision floating point number has $1 + 11 + 52 = 64$ bits. The sign bit, like every other kind of bit, is either 0 or 1. This is interpreted as $\pm$ (I don't know whether 0 is + or $-$, but 1 is the opposite sign from 0.)

The *exponent* of a floating point number is equal to the base 2 integer represented by the exponent bits, minus an exponent *bias*. The exponent bits represent the integer formed by interpreting the bits as "binary digits" (which is the origin of the word "bit"). For example, the 11 exponent bits of a double precision floating point number represent the non-negative integer

$$b_{10}b_9b_8\cdots b_1b_0 \;\longrightarrow\; b_{10}\cdot 2^{10} + b_9\cdot 2^9 + b_8\cdot 2^8 + \cdots + b_1\cdot 2 + b_0$$

5 Recurrence relations

The theory of recurrence relations allows you to see precision being lost, and ill-conditioning arising, not suddenly but gradually over a long sequence of computations. There is nothing absolute such as trying to invert a singular matrix. Instead, several quantities evolve during a computation, with one coming to dominate the others. It takes more and more precision to find increasingly small differences between quantities. The mathematical absolutes of solvability or insolvability are replaced by gradually increasing difficulty of getting answers as accurately as you need them.

A *recurrence relation* of length r is a formula that determines a new x from from the $r-1$ most recent ones. Elements x_k of the sequence are called *iterates*, because recurrence relations could be called *iterations* or *iteration relations*. A length r recurrence relation determines x_{k+1} from x_k, x_{k-1} and down to x_{k-r+2}. For example, a three term recurrence relation ($r = 3$) determines x_{k+1} from x_k and x_{k-1}. If the iterates x_k have one component, then it is a *scalar* recurrence relation. If x_k has n components with $n > 1$ then it is a *vector* recurrence relation. If the formula is the same for each k, then it is a *homogeneous* or *stationary* recurrence relation. A recurrence relation is *linear* if x_{k+1} is a linear function of the previous iterates. A scalar linear homogeneous recurrence, has the form

$$x_{k+1} = a_0x_k + \cdots + a_{r-2}x_{k-r+2} = \sum_{j=0}^{r-2} a_jx_{k-j}\ . \tag{5}$$

The second form on the right makes clear the structure in which j is the *lag* and a_j is the coefficient of lag j.

An important example is the *Fibonacci* recurrence

$$x_{k+1} = x_k + x_{k-1}\ . \tag{6}$$

This three term recurrence determines x_{k+1} from two lagged variables x_k (lag $j = 0$) and x_{k-1} (lag $j = 1$). If $x_0 = 1$ and $x_1 = 1$, the numbers in the resulting

sequence are called the *Fibonacci numbers*, $f_0 = 1$, $f_1 = 1$, $f_2 = 2$, $f_3 = 3$, $f_4 = 5$, $f_5 = 8$, etc.

The Fibonacci relation (6) has a surprising origin. Fibonacci, the person, was born in 1170 in Pisa (now part of Italy). He was a merchant and learned to keep records using *Roman numerals*, a clumsy number system left over from the long past Roman empire. Fibonacci travelled to Béjaïa (now in Algeria) and met Arab merchants who were using a number system we now call *Arabic*. Compare the "old" (Roman) to the "new" (Arabic) expression of some numbers.

Roman:	$XLVIII$	$XLIX$	L	LI	LII	$LIII$	LIV	LV
Arabic:	48	49	50	51	52	53	54	55

Fibonacci wrote a book, in 1202, explaining Arabic numerals and arithmetic to Italians. The Fibonacci recurrence and the sequence 1, 1, 2, 4, 5, 8, 13, 21, 34, 55, 89, 144, $\cdots$ were given just to illustrate how much easier the Arabic way is. Here is $144 = 55 + 89$, found the old way and the new way:

$$LV + LXXXIX = CXLIV \qquad \begin{pmatrix} & {\scriptstyle 1} \\ & 55 \\ + & 89 \\ \hline & 144 \end{pmatrix} .$$

Fibonacci did not know how interesting this accidental sequence would turn out to be.

The solution of a homogeneous scalar linear recurrence may be expressed in terms of *roots* of the *characteristic polynomial*.

6 An example, computing an integral

An algorithm is *stable* if it gives accurate answers to well conditioned problems, and *unstable* if it gives inaccurate answers to well conditioned problems. We will not try to distinguish stable from unstable when solving ill conditioned problems because no algorithm should give accurate results. These are not rigid binary properties. Problems can have varying degrees of ill conditioning (unavoidable loss of accuracy), and varying degrees of instability (possibly avoidable loss of accuracy).

An algorithm may be unstable because it relies on solving an ill conditioned sub-problem. Here is an example. The problem is to compute a one variable integral

$$I = \int_a^b f(x)\,dx \ .$$

There are many simple algorithms that compute accurate approximations to I as long as the integrand is *smooth*, which means f is several times differentiable. More specifically, we consider algorithms that evaluate f at several points x_j then use an approximate formula for the integral in terms of those

values. Section 5 has a systematic discussion and comparison of some of these methods.

It seems natural to approximate I by the integral of a polynomial chosen to approximate f. We experiment with using the *interpolating* polynomial at uniformly spaced points. Throughout the class we use Δx to denote the distance between points. In this case, $x_{j+1} - x_j = \Delta x$, the same Δx for each j. We use n such points in the interval $[a, b]$ with $x_1 = a$ and $x_n = b$. There are $n - 1$ interior intervals $[x_j, x_{j+1}]$, so $(n-1)\Delta x = b - a$. This leads to

$$x_j = a + (j-1)\Delta x \ , \quad \Delta x = \frac{b-a}{n-1} \ .$$

You can check that these definitions imply that $x_n = b$. We write the values of f at the sample points as $f_j = f(x_j)$. The integration algorithm learns about the integrand f by evaluating f_j.

Let $p(x)$ be a polynomial of degree d given in terms of its coefficients as

$$p(x) = c_d x^d + \cdots + c_1 x + c_0 \ . \tag{7}$$

There are $d+1$ coefficients $c_0, \cdots, c_d$ defining a degree d polynomial. We say p interpolates f at the points x_j if

$$p(x_j) = f_j \ , \ \ j = 1, \ldots, n \ . \tag{8}$$

Polynomial interpolation means finding an *interpolating polynomial* that satisfies these *interpolation conditions*.

One way to find the interpolating polynomial is to find the coefficients c_k by solving a system of linear equations. The equations are the interpolation conditions (8). These may be expressed in terms of the coefficients c_k

$$\begin{aligned} j = 1: \quad & c_0 + x_1 c_1 + \cdots + x_1^d c_d = f_1 \\ \vdots \quad & \qquad \vdots \\ j = n: \quad & c_0 + x_n c_1 + \cdots + x_n^d c_d = f_n \end{aligned}$$

These linear equations may be put in matrix/vector form as

$$\begin{pmatrix} 1 & x_1 & \cdots & x_1^d \\ 1 & x_2 & \cdots & x_2^d \\ \vdots & \vdots & \cdots & \vdots \\ 1 & x_n & & x_n^d \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_d \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix} \ .$$

This equation may be written abstractly as

$$Vc = f \ .$$

The matrix and vectors involved are:

$$V = \begin{pmatrix} 1 & x_1 & \cdots & x_1^d \\ 1 & x_2 & \cdots & x_2^d \\ \vdots & \vdots & \cdots & \vdots \\ 1 & x_n & & x_n^d \end{pmatrix} \ , \quad c = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_d \end{pmatrix} \ , \quad f = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix} \ .$$

The matrix V is a *Vandermonde* matrix

7 Exercises

1.

2.

3. This exercise explores the effect of roundoff on evaluating sums. The analysis will use *Stirling's "formula"*, which is an approximation for $n!$

$$n! \approx \sqrt{2\pi}\, n^n e^{-n} \ .$$

This approximation has good relative accuracy when n is large in the sense that

$$\frac{\left|\sqrt{2\pi}\, n^n e^{-n} - n!\right|}{n!} = O(\tfrac{1}{n}) \ .$$

It is common, if you're in a hurry, to use simplified versions such as $n! \approx n^n e^{-n}$ or even $n! \approx n^n$. What you can get away with depends on what you're trying to do.

   (a) For a large x (either positive or negative, but not close to zero) show that the largest (in magnitude) Taylor series term $T_n = \frac{1}{n!}x^n$ occurs with $n = x$ (or nearly, since n is an integer and x does not have to be). One way to do this is to set $|A_n| = |A_{n+1}|$ and solve for n. Then show the magnitudes are increasing for smaller n and decreasing for larger n. Another way is to use Sterling's formula and maximize using calculus. For this, you can get away with the n^n version.

4. We want to evaluate $f(x) = e^x - 1$ to high relative accuracy when x is close to zero. The goal is that if $|x| < 1$ and x is within the range of normalized double precision floating point, then the relative accuracy satisfies

$$\frac{\left|\widehat{f} - f\right|}{|f|} \leq \text{tol} = 10^{-6} \ .$$

   (a) Show that the condition number of the problem allows this accuracy in double precision arithmetic.

   (b) Design a hybrid algorithm that uses

```
f = np.exp(x)-1.
```

   if $|x|$ is larger than some threshold. For smaller x, it should use the Taylor series

$$e^x - 1 \approx x + \frac{1}{2}x^2 + \cdots + \frac{1}{n!}x^n \ .$$

Use the error approximation that replaces the tail of the Taylor series with the first neglected term:

$$\left| e^x - 1 - \left(x + \cdots + \frac{1}{n!} x^n \right) \right| \approx \frac{1}{(n+1)!} |x|^{n+1} \ .$$

Determine the smallest number of terms needed. Find a threshold to minimize the number of terms needed.

5. Write an SDE to model