Week 1

Jonathan Goodman, September, 2021

1 About this Scientific Computing course

sec:about

This class is loosely based on the **book (draft)** by David Bindel and me written for an earlier version of this class. I am re-writing parts to bring them up to date and more Python specific. Professor Aleks Donev has prepared **slides** that roughly follow the same topics. These have the advantage of having fewer words, so that the important things can stand out clearly.

These notes are for a one semester beginning graduate course on scientific computing. The course is intended for students with the background to take a beginning graduate class in applied mathematics. The class is supposed to be practical. Whatever theory there is, and there is quite a bit, is there because it is used in practice.

This course does not make you an expert in scientific computing any more than an introductory programming class makes you a software engineer. Hopefully, this class will give you the background to design and understand more specialized methods for more advanced problems.

To do scientific computing well, you have to understand computers and the process of developing scientific computing software. Just as it's not enough to be a good programmer, it's not enough to be a good mathematician. You have to be both. This course has these less mathematical topics in classes and topics where they seem to fit. Homework exercises will involve that material. Code will be graded on the quality of the software, not just basic correctness.

The core mathematical prerequisites are a good class on linear algebra and multi-variate calculus. The reasoning with involve working with inequalities as is done in an undergraduate course on mathematical analysis. We will use "big O" and "little o" notation, both in describing the sizes of errors, but also in "complexity" (computer science for computational work). Students need a basic familiarity with Python 3. In my experience, the best way to learn is from the Python documentation directly. Stack exchange is also helpful. Less helpful, for me, are the many "Python school" and "for dummies" sites that often come up early in web searches. A simple recipe of the form "these lines of code do that" are a less helpful to people who are not dummies than explanations of the principles involved.

Some parts of the class make use of more specialized mathematical background. The part on dynamics makes use of differential equations. We don't use deep theory, just some understanding of what it means to model a dynamical system using a system of differential equations and the idea that if you give "initial conditions" the rest is determined. The part on Monte Carlo methods uses

basic probability. For this, you should understand the relation between probability density and probability (integration), conditional and marginal probability (more integration). It is helpful to know the formula for a multi-variate normal probability density and the multi-variate central limit theorem.

I intended rewrite the Bindel/Goodman notes completely. That will not be practical. Instead, I will just re-write parts that I now see differently or that depend on Python instead of C++.

2 Sources of Error

sec:sources

A scientific computation (usually) does not seek the exact mathematical answer, which we call A. On the contrary, it seeks an approximation $\widehat{A}$ that is acceptably accurate. The *error* (*absolute* error) is $e = \widehat{A} - A$. Much material in this course addresses five basic questions about error:

- How big is it likely to be?
- How do we estimate how big it is?
- Why is it as big as it is?
- How can we make it smaller?
- How do we work with inexact quantities in a computation?

There are four general ways errors are introduced into a computation

Inexact computer arithmetic

If `x` and `y` are computer variables, then `z = x + y` usually does not produce the exact mathematical sum of the numbers represented by `x` and `y`. Errors in computer arithmetic are *roundoff* or *rounding* error. It is helpful to ignore rounding errors when other sources of error are larger, but many calculations have only rounding error and nevertheless are quite inaccurate.

Approximate formulas

If x is small, the approximation $e^x \approx 1 + x$ may be accurate, but it is not exact. Approximations in calculus also are not exact, including

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x} \; . \qquad (1)$$

eq:fd

Errors like this are often called *truncation* error, because they are thought of as truncating (cutting off) an infinite Taylor sum with just a few terms.

Stopping iterations

An *iteration* produces a sequence of *iterates*, that would converge to the exact answer, such as $A_n \to A$ as $n \to \infty$. Newton's method for solving the equation $f(x^*) = y$ is one example:

$$x_{n+1} = x_n - \frac{f(x_n) - y}{f'(x_n)} \; . \tag{2}$$

eq:Nm

We will see (in a future class) that if x_1 is close enough and if $f'(x^*) \neq 0$, then $x_n \to x^*$ as $n \to \infty$. In practice, we *terminate* the iteration and use the approximate answer $\widehat{A} = x_n$ instead of the exact but unobtainable $A = x^*$.

Statistical error

Monte Carlo computations and *stochastic simulations* use computer generated random numbers as computational tools. For example, suppose X is a random variable with a given probability density $f(x)$ and we want to compute

$$A = \mathrm{E}[\, X] = \int x f(x) \, dx \; .$$

If we can *sample* the distribution f, then we can make independent random variables $X_k \sim f$ and use sample averages as approximations

$$A \approx \widehat{A}_N = \frac{1}{N} \sum_{k=1}^{N} X_k \; . \tag{3}$$

eq:sm

The error in such an approximation is *statistical* error.

Error combination

We often end up balancing errors from different sources in a computation. One example involves the *finite difference* approximation (1) [eq:fd]. If Δx is not small, then truncation error is much larger than roundoff error. But if we try to remove truncation error by making Δx very small, then roundoff error is larger. There is an optimal Δx, small but not too small, that balances these. Exercise 3 [ex:fdr] explores this.

Another combined error example involves the *Robbins Monro* algorithm for stochastic optimization. This is closely related to *stochastic gradient descent* from machine learning. In this problem, you have a function defined as the expected value of a function of a random variable and some parameters

$$v(y) = \mathrm{E}[\, U(X,y)] = \int U(x,y) f(x) \, dx \; .$$

The problem is to find y^* that minimizes $v(y)$, which we write

$$y^* = \arg\, \min_y v(y) \; .$$

Gradient descent is an iteration for finding y^* by going "downhill" using the gradient

$$x_{n+1} = x_n - \alpha \nabla v(y) \; .$$

The parameter α is the *learning rate* (formerly called *step size*). If v is defined as an expected value, then v may be hard to evaluate. You could choose a large N and use a sample mean approximation like (3)

$$\nabla v(x_n) \approx \frac{1}{N} \sum_{k=1}^{N} \nabla_y U(X_{k,n}, y_n) \; . \tag{4}$$

eq:sag

The notation $X_{k,n}$ suggests that you need N independent samples of X for each iteration n, which may be a lot of samples and a lot of work doing the big sum. Robbins and Monro suggested using just one sample per iteration, which corresponds to taking $N = 1$ in (4):

$$y_{n+1} = y_n - \alpha \nabla_y U(X_n, y_n) \; . \tag{5}$$

eq:RM

Either way, (4) or (5), there is statistical error and termination error. If you stop at a small n, the early termination error may be larger than statistical error. But even if $n \to \infty$ in the Robbins Monro iteration (5), there is still statistical error. A smaller learning rate α makes the iteration converge more slowly (need larger n), but reduces statistical error.

3 Relative and absolute error

sec:rae

The *absolute* error is is defined by a formula or by the *absolute error equation*:

$$e = \widehat{A} - A \; , \;\; \widehat{A} = A + e \; . \tag{6}$$

eq:ae

The *relative error* has its formula and *relative error equation*:

$$\epsilon = \frac{\widehat{A} - A}{A} \; , \;\; \widehat{A} = A(1 + \epsilon) \; . \tag{7}$$

eq:re

Relative error is usually a more useful error measure than absolute error. One reason is that relative error is *dimensionless*. You can think of it as a percentage of the answer, as in "one percent accuracy". For example, suppose the exact answer is $A = 12345.67$ and the computed answer is $\widehat{A} = 12249.52$. The absolute error is almost a humdred, which may seem like a lot, but the relative error is

$$\epsilon = \frac{\widehat{A} - A}{A} = \frac{12249.52 - 12249.52}{12249.52} = .0078 \cdots \; .$$

Floating point computer arithmetic (see Section 5) has uniformly small relative error over a wide range of number size. That is, the relative error in computing $x + y$ is almost the same as the relative error in $100x + 100y$, unless y and y are near the edge of the range of floating point numbers. This range

seems very large at first, but simple practical calculations often reach the edge of it. Exercise 6 gives an example.

Catastrophic cancellation is a form of *error amplification* that may be explained in terms of relative error. Suppose x and y are computed to high relative accuracy. This means that the computer has

$$\widehat{x} = x(1+\epsilon_x)$$
$$\widehat{y} = y(1+\epsilon_y)$$

We simplify by imagining that $z = x + y$ is computed exactly:

$$\widehat{z} = \widehat{x} + \widehat{y} \; .$$

For absolute error, we just have (hoping the notation is clear)

$$|e_z| \le |e_x| + |e_y| \; .$$

From the point of view of absolute error addition does not seem to be a problem.

The relative error story can be seriously different. We start with the relative error equation

$$z(1+\epsilon_z) = z + x\epsilon_x + y\epsilon_y \; .$$

This leads to

$$\epsilon_z = \frac{x\epsilon_x + y\epsilon_y}{z} \; .$$

This shows that ϵ_z can be much larger than ϵ_x and ϵ_y if $|z|$ is much smaller than $|x|$ and $|y|$. Of course $x+y$ is much smaller than x or y (in absolute value) if $x \approx -y$. This is *cancellation* – subtracting numbers that are nearly equal or adding numbers that are nearly opposite. Cancellation can lead to a large increase in relative error. *Catastrophic cancellation* means losing all accuracy in one subtraction or addition. More gradual *error amplification* comes from losing a little relative accuracy each operation in a long computation. Exercise 3 illustrates catastrophic cancellation arising from the finite difference formula (1). This happens because $f(x+\Delta x)$ is approximately equal to $f(x)$ when Δx is small. Exercise 5 is an example of gradual cancellation in a sequence of operations.

4 Condition number

sec:cn

The *condition number* of a problem is a fundamental barrier to accurate computer solution of a problem. The condition number of a problem does not depend on the algorithm used to solve it. A problem with a large condition number is *ill conditioned* and may be intrinsically impossible to solve accurately. A problem with a moderate condition number is *well conditioned*.

Condition number measures the relative change in the answer to a problem caused by a small relative change in the input. There are mathematical

definitions of condition number that capture this idea in various ways. One simple definition involves a problem $A(x)$ (the answer to a problem) that depends on a single parameter, x. If x changes by Δx, then the answer changes by $\Delta A = A(x + \Delta x) - A(x)$. The ratio of the relative changes is

$$\frac{\text{relative change in } A}{\text{relative change in } x} = \frac{\frac{\Delta A}{A}}{\frac{\Delta x}{x}}$$

For small Δx, we use the calculus approximation $\Delta A \approx A'(x)\Delta x$. This leads to the differential, one variable condition number

$$\kappa = \left| \frac{xA'}{A} \right| . \tag{8}$$

eq:kap

The condition number, κ, is dimensionless.

The condition number κ is *dimensionless*. For non-mathematicians, this means that whatever units the quantities x and A may have cancel in the ratio (8). For example, if A is the speed (in meters/second) of a rock with density x (in grams per cubic centimeter) then all these units (meters, grams, seconds) cancel. For mathematicians, "dimensionless" means *scale invariant*. You can rescale the answer by a parameter λ and the parameter by μ to get a re-scaled problem $B(x) = \lambda A(\mu x)$. The condition number of B is the same as the condition number of A.

Being dimensionless matters because only dimensionless numbers can be large or small in any real sense. It makes sense to say 600 is big, but the number 600 in "600 seconds" would be changed to 10, if we said the same time as "10 minutes". But even dimensionless numbers might need to be compared to standards. It is helpful to judge condition numbers using accuracy of floating point arithmetic or using uncertainties in input parameters.

Real problems typically have more than one input parameter and more than one output value. Conditioning and condition number is important for such problems, but it is trickier to define. Some aspects of conditioning are quite different in multi-variate settings. The present "scalar" condition number is invariant under rescalings, but re-scalings of multi-variate problems can improve the condition number, provided you scale different variables and/or outputs differently.

There is a real sense in which a problem that is too ill-conditioned cannot be solved by any algorithm. For example, suppose $\kappa = 10^{18}$ (this happens often). On the computer, the first thing you have to do is round x to double precision floating point (see Section 5). That is, the mathematical x is replaced by its floating point approximation $\mathrm{fl}(x)$. The relative difference $\Delta x/x = (\mathrm{fl}(x) - x)/x$ is on the order of the *machine precision* ϵ_{mach}, which is on the order of 10^{-17} in double precision. This means that just inputting the problem into the computer changes the answer by a factor of $\kappa\epsilon_{\mathrm{mach}} \approx 10$. If x is a measured quantity, its accuracy will be much less and the condition number constraint more severe.

A computational algorithm is called *stable* if its accuracy is roughly the best possible given the conditioning of the problem. If an algorithm is not stable

in this sense, then it is called *unstable*. To get an accurate answer, you need a stable algorithm for a well conditioned problem.

If an algorithm is unstable, the reason often is that it involves a sub-problem that is ill conditioned. Here is a famous example that the course will come back to later. It involves a system of linear differential equations

$$\dot{x} = Lx \; .$$

with x having n components and L being an $n \times n$ matrix. Some solution algorithms involve finding the eigenvalues and eigenvectors of L. There are common practical examples where the eigenvalue/eigenvector problem for L is extremely ill-conditioned even though the differential equation system itself is well conditioned. In such cases, an accurate solution strategy cannot use eigenvectors.

5 Floating point arithmetic

sec:fp

Scientific computing consists mostly of computer arithmetic. You should how this arithmetic works to compute well. The command `z=x+y` does not necessarily give the mathematical sum of x and y. What it does depends on the kind of computer number used for x and y.

There are two basic forms of computer number and computer arithmetic, *integer* and *floating point*. In Python, the format for holding a number is called the *data type* (`dtype` in numpy). There are several integer and floating point data types, depending on parameters such as the size (number of bits) and range (signed or unsigned integer).

The *bit* is the basic unit of information stored in a digital computer. A bit has two possible values, called 0 and 1. A *byte* is a sequence of 8 bits. The number of distinct bytes is $2^8 = 256$. This is enough to assign a distinct byte to every commonly used symbol in the Latin alphabet, which is why character strings may have one byte per character. A *word* is a sequence of bytes, often 4 bytes (32 bits) or 8 bytes (64 bits). There are $2^{32} \approx 4 \cdot 10^9 = 4$ billion distinct 32 bit words. A GHz (giga Hertz) is a billion operations per second, so a 2 GHz computer could list all 32 bit words in two seconds. A 64 bit word has about 4 billion more possibilities ($2^{64} = 2^{32} \cdot 2^{32}$), so a practical laptop will not exhaust all of them in a practical amount of time. *Exascale* computing means $10^{18} \approx 2^{64}/16$ operations per second. Such a machine can go through all 64 bit words in a practical amount of time. Billion dollar exascale computers are starting to be constructed.

5.1 integers

An integer data type is a representation of a mathematical integer in base 2 using a sequence of bits.

6 Software standards

sec:ss

As was said already, you have to be good at programming to be good at scientific computing. Studies and personal observation shows that good programmers all adhere, fairly strictly, to personal standards of good programming practice. Computational exercises for this course will be graded for code quality as well as for correctness. Points will be deducted for any of the following standards that are not followed. Some of them may seem to slow you down at first, but software is meant to be used ("production runs") and built on, both of which are harder if the original code is careless and sloppy.

Basic coding style

There should be lots of comments explaining the code. Variable names should be short enough to be easy to read and remember (e.g., `x` for a space variable, `t` for a time variable) and chosen to help the reader know what they're for (e.g. `dx` for Δx). Use white space (blanks and blank lines) to make items in code line up and to separate different parts of code. The top of each Python module should have comments (and, later, *docstrings*) saying what the code does, how it fits in with the overall project (if there's more than one module for the project) and who wrote it and when.

Code flexibility

Every parameter in the code should be a named variable with a comment saying what its role is. For example, rather than `for i in range(10):` (to do something ten times), define `its = 10 # number of iterations` (if it's iterations) followed by `for i in range(its):`. Putting the number 10 in directly is called a "hard wired constant". This makes it hard to modify the code, as you will learn if you break the rule.

Take into account computer arithmetic

Never test whether different floating point numbers are equal – they won't be even if they are equal "in exact arithmetic" (i.e., if a mathematical formula were evaluated exactly). Never make what can be an infinite loop, such as iterating until an error is equal to zero (won't happen) or even until an error is less than .000000001 (might not be met, depending on the sizes of the numbers involved). [By the way, write 1.e-8 than .00000001, because it's easier to read a number than to count zeros.] You don't know, in floating point, whether 10^{-8} is small, large, or impossibly small. Instead, count iterations and quit the loop after some number (e.g., 1000).

Do not fail silently

Be aware of things that can go wrong in a code and plan/code for them. Print an error message or raise an exception if an iteration fails, or if a parameter is

out of range or something like that. Otherwise, the computation can fail – give a completely wrong answer – without the user knowing.

Make easy to read output

Make printed messages line up to be easy to read. Make them clear on their own (e.g., not "error code number seven"). Use formatting for numerical output rather than the built in `str()`. Make clear tables of output.

7 Exercises

In the coding parts, please write code that conforms to all relevant coding standards of Section 6. Part of the homework grade will be based on code quality. Attach printouts of code modules and output to the written homework. Upload the module you used for Exercise 5.

Several exercises ask for "estimates" of quantities. This is something scientists and engineers typically do well and mathematicians do badly. To estimate, you simplify a problem to make calculations easier while, hopefully, preserving the nature of the problem. For example, in exercise 6, you might replace a random variable with a number representing about how big it is likely to be.

Estimates made in this way are not exact. But they allow you to understand practicalities. For example, the maximal N in exercise 5 might be 20 and you might estimate 40. Even though it's off by a factor of two, it explains that not vary large N (less than a hundred) can be impractically large for this. To do this kind of thing effectively, you have to simplify intelligently, understanding what is essential. Unfortunately, math students get informal intelligence drilled out of us.

1. Show that the problem of computing $\sin(x)$ is well conditioned for x near zero but poorly conditioned for x near π.

2. (from Jim Demmel's book *Practical Linear Algebra*) Show that evaluating the polynomial $f(x) = (x-1)^n$ in floating point is more accurate using the direct formula than the binomial expansion when $|x-1| < \frac{1}{2}$ and $n \geq 10$ (arbitrary values meaning x is close to 1, but maybe not very close, and n is large but maybe not very large)

$$(x-1)^n = x^n - nx^{n-1} + \frac{n(n-1)}{2}x^2 - \cdots \pm 1 \ .$$

   Make some specific quantitative statements about the relative error of the answer computer each way. *Hint.* Cancellation.

3. This exercise explores the tradeoff between roundoff and truncation error in estimating the derivative of a function using a finite difference. The

two point forward difference approximation (more on this next week) to the derivative is

$$f'(x) \approx D_+(f, \Delta x) = \frac{f(x + \Delta x) - f(x)}{\Delta x} \; .$$

Do a few experiments that involve making tables of $D_+(f, \Delta x)$ with x and f fixed and a range of Δx values. Each row of the table should include, besides Δx, f' and D_+, the relative and absolute errors. Each row should have results for single and double precision. Once you have the basic code, experiment with the range of Δx to see the truncation error/roundoff error tradeoff (decreasing Δx at first improves accuracy, then reduces accuracy). Try at least the examples $f(x) = e^x$ with $x = 1$, and $f(x) = \sin(x)$ with $x = 0$. The week 2 material will explain why results are better for the second example. The posted Python code `PrecisionDemo.py` for how to control precision.

ex:Fth

4. (*This exercise explores the mathematics of recurrence relations, in the special case of the Fibonacci numbers. This analysis of recurrence relations is used in many places in scientific computing.*) The *Fibonacci numbers* are a sequence[1] $f_0 = 1$, $f_1 = 1$, $f_2 = f_0 + f_1 = 2$, $f_3 = f_2 + f_1 = 3$, etc. The general rule for Fibonacci numbers is

$$f_{n+1} = f_n + f_{n-1} \; . \tag{9}$$

eq:Fr

This is an example of a *recurrence relation*. The sequence is completely determined by its *initial conditions* ($f_0 = 1$ and $f_1 = 1$), and the recurrence relation (9).

   (a) Show that there are two numbers r so that the sequence $f_n = r^n$ satisfies the recurrence relation (9). Note, these are not the Fibonacci numbers because they do not satisfy the initial conditions. Show that they satisfy $0 < r_1 < 1 < r_2$. The number r_2 is called the *golden mean* or the golden *ratio*. Figure 1 explains what this means.

[1] This sequence appears in a book *Liber Abaci* ("liber" means "book" (Latin) and "abaci" means "calculations" (arithmetic)) that was published in 1202 by the Italian trader now called "Fibonacci". The book explained that the *Arabic* number system, which he had just learned during a business trip to Beirut and we still use, makes arithmetic easier than the system generally used in Italy in 1200 – Roman numerals. For example, XIV + XCVI = CX Roman numeral equivalent of $14 + 96 = 110$. The Fibonacci numbers in Roman numerals are I, I, II, III, V, VIII, XIII, XXI, XXXIV, LV, LXXXIX, $\cdots$. Imagine an Italian trader (like Fibonacci) keeping financial records using Roman numerals. In the book, the "Fibonacci sequence" is used to show how easy the Arabic number system is.

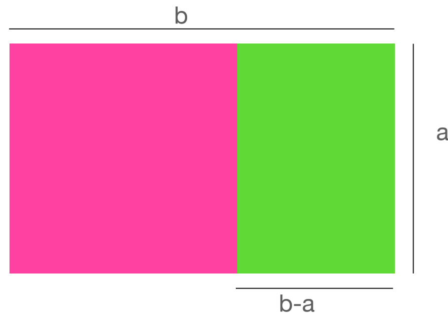

Figure 1: The perfect aspect ratio, according to ancient Greek temple builders, had long side b and short side a so that when you take out a square (magenta on the left), the remaining rectangle (green on the right) has the same aspect ratio. That means that $\frac{b}{a} = \frac{a}{b-a}$. The left side is the aspect ratio of the big rectangle. The right side is the aspect ratio of the green rectangle on the right. The ratio $\frac{b}{a} = r_2$ (in notation here) is used for the Parthenon. fig:GoldenMean

(b) Show that if f_0 and f_1 are any two numbers, then there are numbers x_1 and x_2 so that $f_0 = x_1 + x_2$ and $f_1 = x_1 r_1 + x_2 r_2$. Do this by finding a 2×2 matrix A so that these equations are given in matrix form as $Ax = f$, with

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \; , \quad f = \begin{pmatrix} f_0 \\ f_1 \end{pmatrix} \; .$$

Show that A is non-singular if and only if $r_1 \neq r_2$.

(c) Show that if f_n is any sequence that satisfies the recurrence relation (9) eq:Fr, regardless of initial conditions, then the sequence is given by

$$f_n = x_1 r_1^n + x_2 r_2^n \; .$$

The coefficients x_1 and x_2 are found as in part (b).

(d) Show that if $x_2 \neq 0$, then

$$\frac{f_{n+1}}{f_n} \;\to\; r_2 \; , \quad \text{as } n \to \infty \; .$$

Show that this ratio limit formula holds for the fibonacci numbers ($f_0 = f_1 = 1$). The ratio of Fibonacci numbers converges to the golden mean as $n \to \infty$.

(e) Suppose that the recurrence relation is implemented in floating point, as in `F[n+1] = F[n] + F[n-1]`. This would produce floating point numbers with relative errors ϵ_n, as expressed by $F_n = f_n(1 + \epsilon_n)$. The floating point addition also would have an error, as in $F_{n+1} = (F_n + F_{n-1})(1 + \delta_n)$. Derive the error propagation equation

$$\epsilon_{n+1} = \delta_n + \epsilon_n \frac{f_n}{f_{n+1}} + \epsilon_{n-1} \frac{f_{n-1}}{f_{n+1}} \; . \tag{10}$$ eq:epe

(f) Give an informal but quantitative argument that (eq:epe 10) implies that the relative error in computing the f_n in floating pointing point arithmetic is on the order of $n\epsilon$mach, as long as f_n is within the range of floating point numbers. *Hint.* The ratios on the right are roughly $\frac{1}{r_2}$ and $\frac{1}{r_2^2}$ respectively, unless n is small. Show that $\frac{1}{r} + \frac{1}{r^2} = 1$ for either root. You can do this directly from $r_2 = \frac{1}{2}(1+\sqrt{5})$, which makes it seem special for the Fibonacci recurrence relation (eq:Fr 9), or you can derive it from the polynomial equation r satisfies.

(g) (*The point of this sequence of steps*) Suppose we have computed up to some N and have in the computer $g_N = (1+\epsilon_N)f_N$ and $g_{N+1} = (1+\epsilon_{N+1})f_{N+1}$. Suppose we compute g_0 using the Fibonacci relation backwards: $g_{n-1} = g_{n+1} - g_n$. Suppose that the g_n calculations are done exactly. If $\epsilon_N = \epsilon_{N+1} = 0$ then $g_n = f_n$. Assume that $|\epsilon_N| \approx= \epsilon$ and $|\epsilon_N| \approx= \epsilon$. Find how large N can be so that $|f_0 - g_0| \leq f_0 = 1$. What are the numbers for $\epsilon \approx 10^{-7}$ (single precision) and $\epsilon \approx 10^{-16}$ (double precision)? How much difference does it make (how much does N change) if you replace the assumption by the more realistic (according to part (f)) bound $\epsilon_N \sim N\epsilon_{\text{mach}}$?

ex:Fib

5. Write a Python code to verify/explore the results of Exercise ex:Fth 4. Your code should start with f_0 and f_1 and use the Fibonacci recurrence to compute f_n for n up to some $N+1$, then turn around and re-compute f_{n-1} from f_n and f_{n+1} to get back to f_0. Compare the re-computed f_0 to the original one. Make a table (well formatted) of the relative and absolute error as a function of N for a range of N where the result is completely wrong. Repeat the experiment for the related recurrence $f_{n+1} = af_n + f_{n-1}$ with $a = 1 + .1 \cdot \sqrt{2}$. The precise formula for a is not important, only that it is close to 1 so the recurrences behave similarly and that it is not a simple number base 2. Describe the similarities and differences in the results. Use what you know about computer arithmetic to explain the difference.

ex:ml

6. (*This exercise illustrates that the range of values in floating point can be an issue, even when the range goes up to* 10^{300}. *It is "real" in the sense that more than one person working with me has had to modify a code to avoid this issue.*) Suppose you have data values X_i observed at times t_i with unknown precision σ (same for each observation). The model is

$$X_i = A\sin(\omega t_i + \theta) + r_i \; .$$

The observation errors are the "residuals" r_i, which we assume are Gaussian with mean zero and common variance σ^2. We wish to fit the model parameters A (amplitude), ω (frequency), and θ (offset). We also will fit the "nuisance parameter" (more properly, *latent* parameter) σ. With observation error, the probability density to observe x at time t is

$$f(x|t, A, \omega, \theta, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x - A\sin(\omega t + \theta))^2}{2\sigma^2}} \ .$$

The corresponding *likelihood* function is

$$L(A, \omega, \theta, \sigma) = \prod_{i=1}^{n} f(X_i|t_i, A, \omega, \theta, \sigma) \ .$$

Maximum likelihood estimation means finding parameters $A, \omega, \theta, \sigma$ to minimize L. To do this, a code must evaluate L with parameter values possibly far from optimal. Specifically, suppose there are $n = 100$ data points and you want to evaluate L with $\sigma = 1$ when the true value that made the data had $\sigma_* = 3$. Show that L is probably outside the range of double precision floating point. For this, assume that (a Gaussian random variable it typically about the size of its standard deviation)

$$\frac{(X_i - A\sin(\omega t_i + \theta))^2}{2\sigma_*^2} = \left(\frac{r_i}{\sigma_*}\right)^2 \sim 1 \ .$$

The *log likelihood* function is $l(\cdots) = \log(L(\cdots)$. The log likelihood is a sum over data, rather than a product. Show that the log likelihood is likely to be within the range of floating point for reasonable values of n and σ. [Of course, minimizing l also minimizes L. This exercise illustrates one of the reasons practical minimization codes typically use l instead of L.]