1 Limited Precision

1.1 What is limited precision?

Not all numbers can be exactly represented. We begin by examining 0.1, which can also be written as $\frac{1}{10}$. This is a simple number, and exactly represented, at least in base 10. Computers, however, use base 2, and 0.1 is not exactly represented in base 2. No base will be able to represent all numbers exactly with a finite number of digits.

$\frac{1}{3}$ is a good example of this for base 10. Trying to write it in decimal form requires us to write "0.33333" with infinitely many 3s, or to introduce some notation: "$0.\bar{3}$. To represent it with just finite digits, we have to truncate after some number of digits. If we truncate at three digits, for example, we can get the following:

$$\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 0.333 + 0.333 + 0.333 = 0.999 \neq 1.$$

Any finite number of digits will be insufficient to represent the number exactly, but computers cannot store infinite digits, so we are stuck with limited precision.

1.2 How to store a number for a base 2 computer

A computer does not store numbers as such; it just stores bits that represent a number. There are conventions for converting a set of bits into a floating point number. The current prevalent system, which hardware is optimized for, is **double precision:** each float gets 8 bytes, so 64 bits.

To store a number, we need a sign bit, which is either positive or negative. This leaves us with 63 remaining bits to store the magnitude of the value. We use scientific notation for this: we have some **exponent** which determines the order of magnitude of the numbers we can span. The digits of precision of our value are given by the **mantissa:** the coefficient of our scientific notation. The current standard is IEEE754, which allocates 11 bits to the exponent and the remaining 52 to the mantissa.

Computers have built-in floating point unit that does float math for you; it has been optimized extensively to be very fast at specifically double precision float math: we refer to this as **machine precision**. You could do things at higher precision, but it is more difficult, slower, and more computationally expensive, so we want to work at machine precision.

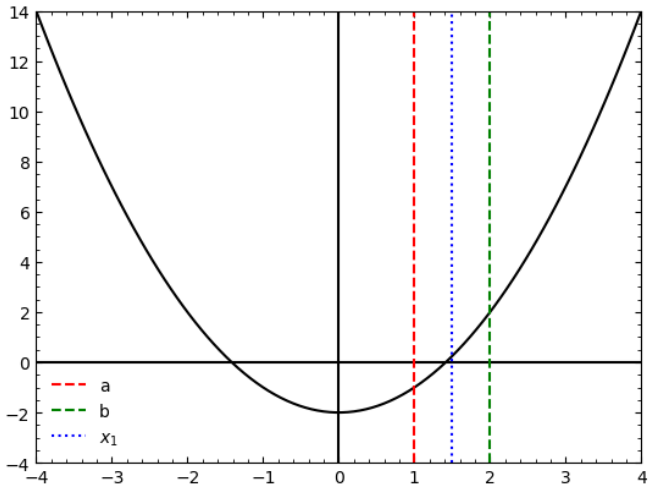
2 Root Finding

2.1 Bisection

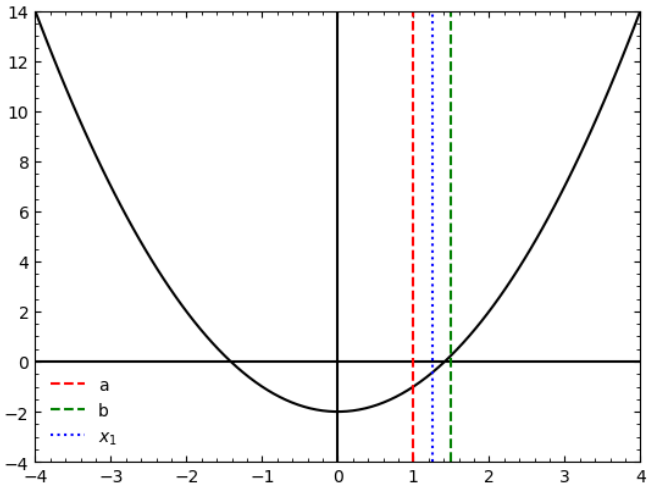
By root finding, we mean finding x where $f(x) = 0$: looking for extrema. This is a problem that comes up frequently, and we can turn many naturally occuring problems into root-finding-shaped problems. Say we want to calculate $\sqrt{2}$. We can set $x = \sqrt{2} \quad \rightarrow \quad x^2 - 2 = 0$ and then use root finding to compute $\sqrt{2}$.

To begin root finding, we want to have a range of values we expect the root to be in: this is called **bracketing**. For our above example, we know that $1 < 2 < 4$, so then $\sqrt{1} < \sqrt{2} < \sqrt{4}$, or $1 < \sqrt{2} < 2$. We expect our value for $\sqrt{2}$ to be between 1 and 2.

We move brackets closer to the root via **bisection**. Given $x \in (a, b)$, we make an initial guess: $x_1 = \frac{a+b}{2}$. If $f(a) < 0$ and $f(x_1) < 0$, we know that all values between a and x_1 are less than 0, and thus none of them are the root. The same goes if $f(b) > 0$ and $f(x_1) > 0$.



Here we see that the latter is true, so we can ignore everything between x_1 and b. We can set $b = x_1$, and then define $x_2 = \frac{a+b}{2}$ with our new definition of b.

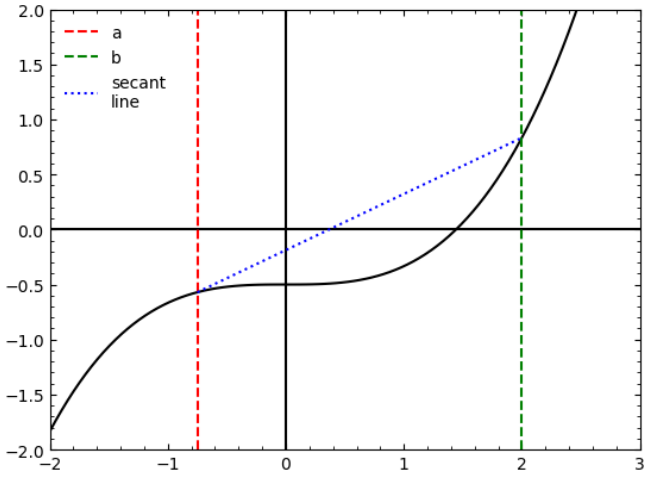


We continue this process, binary searching until we are close to the correct value (for some definition of close); limited precision prevents perfect accuracy.

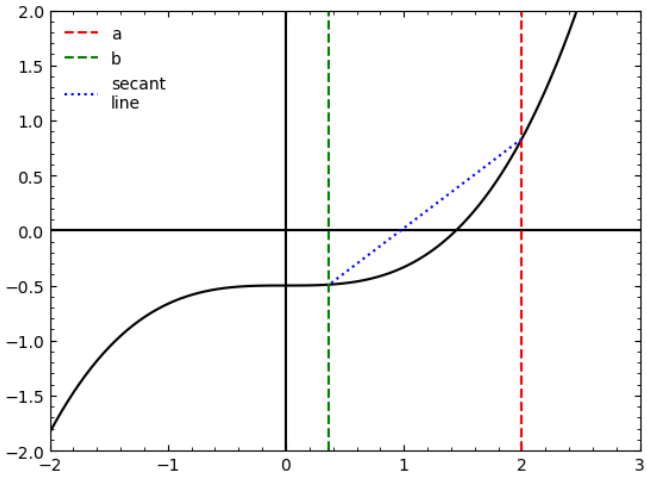
Bisection always finds a root, but works slowly, because it only ever cuts the region in half. We can speed the process up by using more information, but that will require more computations, which will sometimes make things slower again. There is no general best algorithm for root finding; it varies from case to vase.

2.2 Secant method

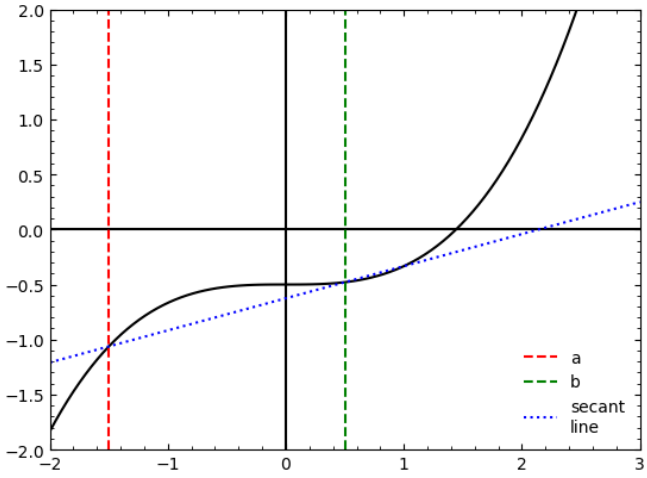
The secant method is built on the mean value theorem (for a continuous line, the secant line between two points will be parallel to the tangent line of at least one point on the arc). For this method, we do not need to bracket the root, though we can. Thus we can use information about how this curve is changing to improve our estimate, or get to a root more quickly. We again begin by picking two points, a and b, and draw the secant line between them.



(We see that this secant line is parallel to the tangent line at some point along the curve between a and b.) This secant line passes through 0 at some point, and that point becomes our new guess for the root. We relabel b as a, and set a to be x_0, the y-intercept of the secant line.



We repeat this process again until we are sufficiently close to the root of the function. Bracketing in not necessary here because if both lines are to one side of the root, the secant line will intersect on the other side, so it self-corrects:



So the procedure for the secant line is as follows:

- Begin with two guesses, a and b. They do not need to bracket the root.
- Construct the secant line between those points.
- Solve for x_0, where the secant line intersects the x-axis: $x_0 = b - (b-a)\frac{f(b)}{f(b)-f(a)}$
- Move $a \to b$ and $b \to x_0$; repeat.

More generally, given x_{n-1} and x_n, the algorithm guesses

$$x_{n+1} = x_n - (x_n - x_{n-1})\frac{f(x_n)}{f(x_n) - f(x_{n-1})}.$$

This is good because it uses information about the shape of the function, which can help us find a root more quickly. If we Taylor expand the function around the root, the first order term will almost certainly be linear, and the secant line is linear, so the Taylor expansion also justifies this linear approximation.

However, not enforcing bracketing means it can get really far away from the root. Analytically, this system will always converge (for non-degenerate cases), so this potential failure is a numerical issue.

2.3 Newton-Rhapson method (or just Newton's method)

We again want to use more information to speed up root finding, so we now explore using derivative information of the curve. (This, of course, requires that we know the derivative.) This algorithm relies on the Taylor expansion. We have some function $f(x)$ which we want to expand around a point, allowing some deviation from the point. $f(x+h)$ expanded around $h=0$, where h is a step size, is given by

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2!}f''(x)h^2 + \dots$$

If we assume h is small, we can truncate at order h^2 without losing too much information. Doing so leaves us with the linear approximation

$$f(x+h) = f(x) + f'(x)h.$$

If we begin at some x such that $f(x) \neq 0$ (in other words, not a root), our goal is to move to $x+h$ such that $f(x+h) = 0$. To do this, we require that $f(x+h) = f(x) + f'(x)h = 0$, which we can also write as

$$h = \frac{-f(x)}{f'(x)}.$$

This lets us find some value of h such that $x+h$ is a better guess for the root. It won't be exactly the root because we heavily truncated the Taylor expansion, so we iterate to progressively approach the root.

The algorithm is as follows:

- Start with a single guess for the root: x_0.
- Iterate: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. (This converges quadratically.)

This, too, can fail spectacularly—we are dividing by something, which should raise an alarm. What if we're dividing by 0? Numerically, we need to be concerned not only about dividing by 0, but also about dividing by a very small number, since that will cause the quotient to blow up. This method works best when you start close to the root, so a rule of thumb is to use another method to get close to the root, and the Newton-Rhapson method to refine the estimate.

2.4 Summary of root finding

Basic lessons learned in root finding are as follows:

- Always bracket the root—know a range of values where a root exists.
- There is no best algorithm.

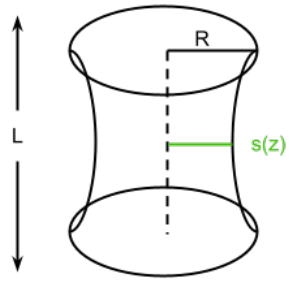
– "■Safe and slow"■ algorithms (like bisection) are useful to get close to the root.

Once you are close, polish the root with a "fast and unsafe" algorithm like the Newton-Rhapson method.

Brent's method, implemented in `scipy.optimize.brentq`, is a good method in that it mixes bisection, the secant method, and inverse quadratic interpolation.

2.5 Example: the shape of a soap film

We consider a film of soap between two rings:



The radius of the rings is R, but the radius of the film at any height z is given by $s(z)$. Our goal is to find the shape of the soap film, which in this case is determined by what would minimize surface tension, so we want to minimize surface area. Surface area is given by

$$A = \int 2\pi s(z) dl, \qquad \text{where } dl^2 = dz^2 + ds^2 \;\rightarrow\; dl = \sqrt{1 + s'^2}\;\; dz.$$

We can rewrite our expression for surface area as

$$A = 2\pi \int_0^L s\sqrt{1 + s'^2}\;\; dz.$$

We use the Euler-Lagrange equation to minimize surface area:

$$\frac{\partial f}{\partial s} - \frac{d}{dz}\left(\frac{\partial f}{\partial s'}\right) = 0.$$

We multiply this through by s':

$$\boxed{s'\frac{\partial f}{\partial s}} - s'\frac{d}{dz}\left(\frac{\partial f}{\partial s'}\right) = 0,$$

and notice that

$$\frac{df}{dz} = \left(\frac{\partial f}{\partial s}\right)\left(\frac{ds}{dz}\right) + \left(\frac{\partial f}{\partial s'}\right)\left(\frac{ds'}{dz}\right) = \boxed{s'\left(\frac{df}{ds}\right)} + \left(\frac{\partial f}{\partial s'}\right)\left(\frac{ds'}{dz}\right).$$

Then we can write

$$\frac{df}{dz} - \left(\frac{\partial f}{\partial s'}\right)\left(\frac{ds'}{dz}\right) - s'\left(\frac{ds}{dz}\right)\left(\frac{\partial f}{\partial s'}\right) = 0,$$

which reduces to

$$\frac{d}{dz}\left(f - s'\left(\frac{\partial f}{\partial s'}\right)\right) = 0.$$

So

$$f - s'\left(\frac{\partial f}{\partial s'}\right) = \alpha,$$

where α is some constant. Then

$$s\sqrt{1 + s'^2} \;\; - s'\left(\frac{\partial}{\partial s'}\right)\left(s\sqrt{1 + s'^2}\right) = \alpha = s\sqrt{1 + s'^2} \;\; - \frac{ss'^2}{\sqrt{1 + s'^2}}.$$

We can combine these expressions to find that

$$\frac{s + ss'^2 - ss'^2}{\sqrt{1 + s'^2}} = \alpha = \frac{s}{\sqrt{1 + s'^2}}.$$

This is a first order differential equation. We can rewrite it in a nicer form:

$$s^2 = \alpha^2 + \alpha^2 s'^2 \quad \rightarrow \quad \alpha \, \frac{ds}{dz} = \sqrt{s^2 - \alpha^2}.$$

Integrating this, we find

$$\alpha \int \frac{1}{\sqrt{s^2 - \alpha^2}} \, ds = \int dz = z.$$

We substitute in $s = \alpha \cosh(u)$ and find

$$\alpha \cosh^{-1}\left(\frac{s}{\alpha}\right) + \beta = z,$$

which we can use to find that

$$s(z) = \alpha \cosh\left(\frac{z - \beta}{\alpha}\right).$$

That was a lot of math! The point is we can do a lot of work analytically before we throw it at a computer. We ended up with α and β in our answer, which will depend on the system we're talking about. We have boundary conditions (both rings have the same radius R, and the distance between them is L, so we can use these to find the constants. The soap film starts and ends with radius R:

$$s(0) = R = \alpha \cosh\left(\frac{-\beta}{\alpha}\right) \quad \rightarrow \quad \beta = \alpha \cosh^{-1}\left(\frac{R}{\alpha}\right).$$

We can find α from the other endpoint condition:

$$s(L) = R = \alpha \cosh\left(\frac{L - \beta}{\alpha}\right) \quad \rightarrow \quad R = \alpha \cosh\left(\frac{L}{\alpha} - \cosh^{-1}\left(\frac{R}{\alpha}\right)\right).$$

We can't algebraically solve that for α; it's a transcendental equation that has nom-algebraic solutions. Thus, we solve it numerically—this is where we would use root finding.

3 Interpolation

3.1 What is interpolation?

Interpolation is used to approximate a function. Functions typically are of real values and return real values, but a numerical function has limited precision and finite memory, so you can't feed in any number and get out a precise value like you can for an analytic function. Discretized numerical functions have a finite domain and range; they are more like a table of inputs and outputs, but we get a visually smooth curve if the spacing between values is small enough. **Interpolation** is filling in the gaps between the discrete values in the table.

How we interpolate depends on our goals (e.g. wanting a function that's easy to evaluate, one that's integrable, differentiable, etc.).

Interpolating functions are approximations of functions based on tabulated values $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$. An interpolating function has a property that is "exact" at the tabulated points:

$$y_i = f(x_i).$$

In other words, our interpolating function exactly goes through these points. There are many ways to approximate functions:

- A polynomial that passes through all N points; this leads to a smooth interpolating function but is bad for large N
- A piecewise polynomial consisting of lines between every pair of points ("connecting the dots"); this isn't smooth
- Trigonometric and rational functions
- A **spline:** a smooth version of a piecewise polynomial

For polynomial interpolation, think of writing some polynomial $f(x) = a_0 + a_1 x + a_2 x^2 + \ldots$. We construct this using a Lagrange interpolating polynomial, which we will learn about in more detail later; this allows us to "just write down the answer."

For piecewise polynomial interpolation, instead of trying to put a full polynomial between all the points, we can connect two points linearly, three points quadratically, etc. We split the points into smaller chunks and make piecewise lower order polynomials. This is not differentiable where the pieces meet, but we can add more points without requiring higher order polynomials.

3.2 Polynomial Interpolation

Let $p_0(x),\ p_1(x),\ p_{n-1}(x)$ be a polynomial. The obvious choice here is to use $p_i = x^i$, but we could also use Legendre polynomials or some other variety. Our function is then

$$f(x) \approx \sum_{j=0}^{n-1} a_j p_j(x).$$

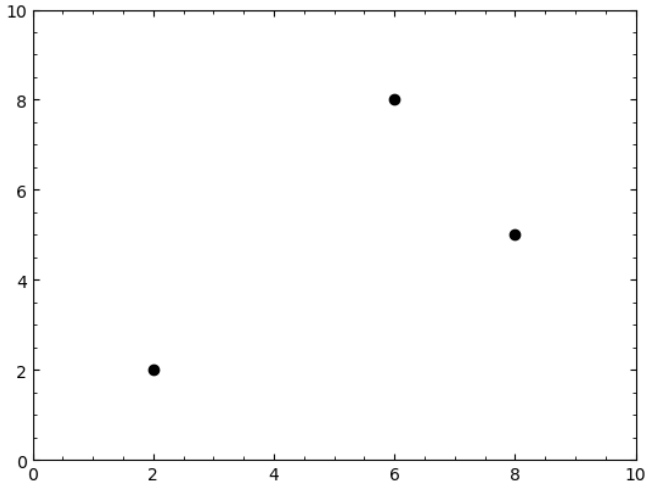
An interpolating polynomial goes through all our points (x_i, y_i),

$$y_i = f(x_i) \approx \sum_{j=0}^{n-1} a_j p_j(x_i),$$

such that an interpolating polynomial is a system of linear equations where $p_j(x)$ are given, but the coefficients a_j are unknown. This is a system of N equations for n unknowns, so it has a unique solution.

This method fails for large N: it exhibits **Runge's Phenomenon** of wild oscillation. Also, for large N, the high amount of addition and subtraction going on causes a loss of precision, as well as slow evaluation.

As an example, we consider quadratic interpolation (approximating some function as quadratic). For some random set of 3 points, we want to find a function $f(x) = ax^2 + bx + c$.



We can imagine the shape of the polynomial that passes through these points, but we need to numerically find a, b, and c:

$$f(x_0) = y_0 = ax_0^2 + bx_0 + c$$

$$y_1 = ax_1^2 + bx_1 + c$$

$$y_2 = ax_2^2 + bx_2 + c.$$

We see that this is a system of three equations and three unknowns, but we don't feel like solving it by hand, and have not yet learned to do it numerically, so we set the example aside.

In its stead, we turn to the **Lagrange interpolating polynomial**. We first look at a linear case, where we

have (x_0, y_0) and (x_1, y_1) and want to construct a polynomial passing through both. Our polynomial is given by

$$f(x) = \boxed{\left(\frac{x - x_1}{x_0 - x_1}\right)} y_0 + \boxed{\left(\frac{x - x_0}{x_1 - x_0}\right)} y_1.$$

We introduce some notation to make this easier to write:

$$L_{1,j}(x_i) = \delta_{i,j} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

where $\delta_{i,j}$ is the Kronecker delta. The 1 in the subscript of L indicates a first order Lagrange interpolating polynomial. Using this notation, our example becomes

$$f(x) = \boxed{L_{1,0}(x)} y_0 + \boxed{L_{1,1}(x)} y_1.$$

Generalizing this: to construct an nth order interpolating polynomial that passes through n+1 points, we construct

$$L_{n,j}(x_i) = \delta_{i,j} \quad \rightarrow \quad f(x) = \sum_{j=0}^{n} y_j L_{n,j}(x).$$

We can expand this out:

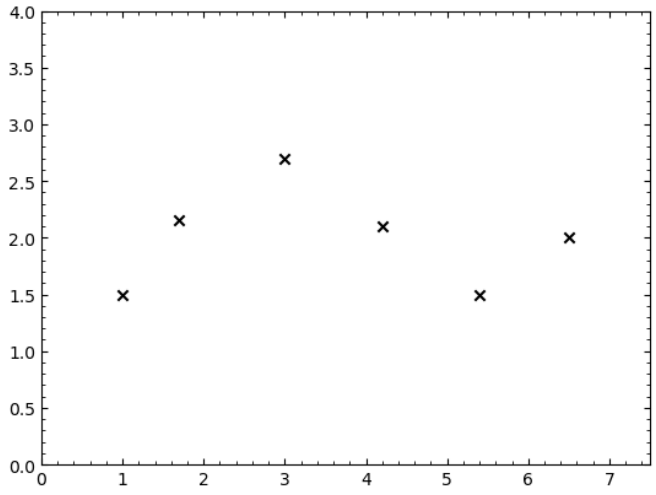
$$L_{n,j} = \frac{(x - x_0)(x - x_1)\dots(x - x_{j-1})(x - x_{j+1})\dots(x - x_n)}{(x_j - x_0)(x_j - x_1)\dots(x_j - x_{j-1})(x_j - x_{j+1})\dots(x_j - x_{n+1})}$$

where we skip subtracting x_j so as to not divide by 0. We can write this more neatly as

$$\prod_{\substack{i=0 \\ i \neq j}}^{n} \left(\frac{x - x_i}{x_j - x_i}\right).$$

This is nice most of the time, but fails for lots of points. That's sad, because having lots of points is supposed to make interpolation more accurate. Piecewise polynomials fix this: they use fewer points and more than one polynomial. Cubic polynomials especially are good building blocks—you can use higher order polynomials, but don't want to go too high. However, a regular piecewise polynomial is not smooth, and thus not differentiable at all points. A spline is the standard response to this.

A **spline** is a smooth, piecewise polynomial; in this case, a piecewise cubic polynomial with some boundary condition that ensures smoothness and makes sure the function is uniquely defined. (Note that we essentially always mean a cubic spline spline if we have not otherwise specified order.)



For each pair of points, we construct a cubic passing through those two points. To construct a unique cubic polynomial through two points instead of four, we need more constraints, which in this case are provided by boundary conditions to ensure smoothness. A spline stitches together polynomials to get a smooth, continuous, differentiable result. Boundary conditions also handle the endpoints of the function.

A note on spline vocabulary: the points used to construct the spline can also be called knots or nodes.

3.3 Splines

Given a function $f(x)$ defined on some interval $[a, b]$ at some set of ordered knots,

$$a \equiv x_0 < x_1 < x_2 < \cdots < x_{n-1} < x_n \equiv b,$$

let S be a cubic spline interpolant (interpolating function) defined by:

1. $\boldsymbol{S_j(x)}$**: a cubic polynomial defined on** $\boldsymbol{[x_j, x_{j+1}]}$. We construct lots of these to make the spline: one for each j, or one between each pair of knots. For this set of points, there are n such polynomials, one for each $j = 0, \ldots, n-1$. Since we are only using two points to define our cubic function, we need two more conditions to uniquely define our polynomial.
2. **Requiring** $\boldsymbol{S(x_j) = f(x_j)}$. This is the definition of an interpolating function: it goes through the given points. Note that we said S, not S_j, so we are talking about the entire spline. However, a spline is defined so that $S_j(x_j) = f(x_j)$ as well as $S_j(x_{j+1}) = f(x_{j+1})$. This provides two conditions on S_j that need to be satisfied for our spline to be an interpolating function.
3. **Requiring** $\boldsymbol{S_j(x_{j+1}) = S_{j+1}(x_{j+1})}$: the function must be continuous.
4. **Requiring** $\boldsymbol{S'_j(x_{j+1}) = S'_{j+1}(x_{j+1})}$.
5. **Requiring** $\boldsymbol{S''_j(x_{j+1}) = S''_{j+1}(x_{j+1})}$. This and condition 4 ensure smoothness at x_j, as well as add two more conditions for a total of 4, which we need to uniquely define a cubic. A non-cubic spline will have however many smoothness conditions are needed to uniquely define polynomials of its order.

These conditions almost uniquely define our spline, but we still need to set the behavior at the endpoints, x_0 and x_n. Some common boundary conditions are:

- Natural or free boundary conditions: $S''(x_0) = S''(x_n) = 0$
- Clamped boundary conditions: $S'(x_0) = f'(x_0)$ and $S'(x_n) = f'(x_n)$. This requires us to know the derivative of our function at the endpoints.
- "Not-a-knot" boundary conditions: that the first and last segments have the same third derivative (or the

highest nonzero derivative for our polynomial). This is generally considered a good default.

The implementation of a spline we will use (in `scipy.interpolate.make_interp_spline()` is a B-spline, which is a basic building block and one of the standard ways of constructing a spline.

There may be a temptation to use splines to extrapolate, but we should consider this very dangerous, and avoid doing it as much as possible. The difference between the two is that extrapolation tells you (or attempts to tell you) what is happening on either end of your data points, and interpolation tells you what is happening between your data points. Extrapolation is dangerous because we know nothing about the behavior of our function outside of the data points we have.

4 Numerical Derivatives

4.1 Methods to numerically differentiate

Numerical derivatives, while rarely necessary to calculate for their own sake (since we can almost always analytically differentiate) are a basis for many other functions like differential equations and numerical integration, so they're still important.

We begin with a Taylor expansion (our favorite thing from calculus!) with step size h, to expand a function $f(x+h)$ around $h=0$:

$$f(x+h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{3!}f'''(x)h^3 + \dots$$

If $h << 1$, we can truncate at $\mathcal{O}(h)$:

$$f(x+h) = f(x) + f'(x)h + \mathcal{O}(h^2).$$

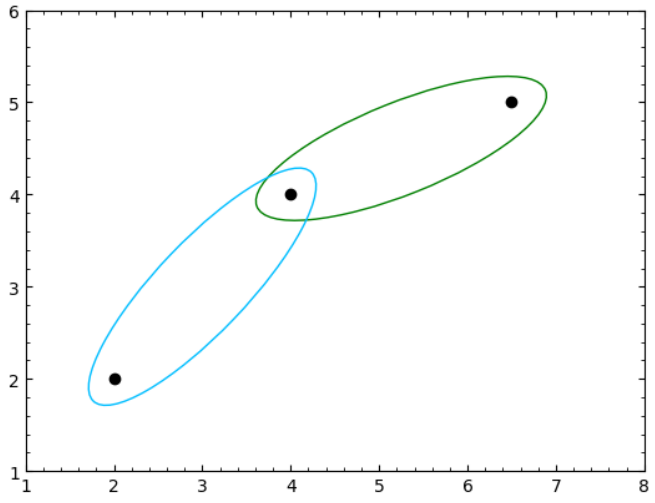
We can solve this for the derivative to find the **forward differencing** formula:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h),$$

where $\mathcal{O}(h)$ is the error term. We now expand around $f(x-h)$, truncate, and solve for $f'(x)$ to find the **backwards differencing** formula:

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \mathcal{O}(h).$$

We can graphically see the difference between these methods:



If we want the derivative at the center point, forward differencing uses the points circled in green, and backward

differencing uses the points circled in blue.

The error for this scales with h, so that we need a significantly smaller step size for some improvement in accuracy. To do better, we can consider our second expansion subtracted from the first: $f(x+h) - f(x-h)$. If we do that subtraction, we find that many terms cancel to leave us with

$$f(x+h) - f(x-h) = 2f'(x)h + \mathcal{O}(h^3).$$

Solving this for $f'(x)$ gives us the **center differencing** formula, where we do not use the value at the x at all, only its neighbors:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2).$$

With center differencing, reducing h by a factor of 2 means reducing error by a factor of 4.

Now we add our two expansions:

$$f(x+h) + f(x-h) = 2f(x)h + f''(x)h^2 + \mathcal{O}(h^4).$$

This lets us estimate the second derivative:

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + \mathcal{O}(h^2).$$

All of these methods are called stencils—a **stencil** is a method we use to calculate the derivative.

4.2 Richardson Extrapolation

Richardson extrapolation is a general technique for improving the accuracy of any algorithm involving a step size where the error scales with that step size. We take center differencing as an example:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2).$$

Recall that $\mathcal{O}(h^2)$ is just the leading order error, and we also have $\mathcal{O}(h^4)$, $\mathcal{O}(h^6)$, etc. More generally, we can think of this as something where we pick h and it returns a value. Thus, we name our center differencing function $F_1(h)$. This is exactly the derivative as h goes to 0, but it can only approach exactness to avoid division by 0.

Richardson extrapolation says we extrapolate to 0. We know $F_1(h)$ has error scaling as h^2, so we can say

$$F_1(h) = a_1 + \alpha_1 h^2 \qquad (\text{known},\ \text{unknown})$$

where a_1 is the true value as h goes to 0, and α_1 is the leading order error.

This leaves us with one equation and 2 unknowns, so we also evaluate $F_1\left(\frac{h}{2}\right)$:

$$F_1\left(\frac{h}{2}\right) = a_1 + \alpha_1 \left(\frac{h}{2}\right)^2.$$

With two equations and two unknowns, we can solve for our true value, a_1:

$$a_1 = F_1(h) + \frac{F_1\left(\frac{h}{2}\right) - F_1(h)}{1 - \frac{1}{4}}.$$

However, this still has error $\mathcal{O}(h^4)$, so we can think of a_1 as an algorithm to calculate the derivative with an error

that scales as h^4. We then define

$$F_2(h) \equiv a_1 = F_1(h) + \frac{F_1\left(\frac{h}{2}\right) - F_1(h)}{1 - \frac{1}{4}} = a_2 + \alpha_2 h^4.$$

We again look at $\frac{h}{2}$:

$$F_2\left(\frac{h}{2}\right) = a_2 + \alpha_2 \left(\frac{h}{2}\right)^2,$$

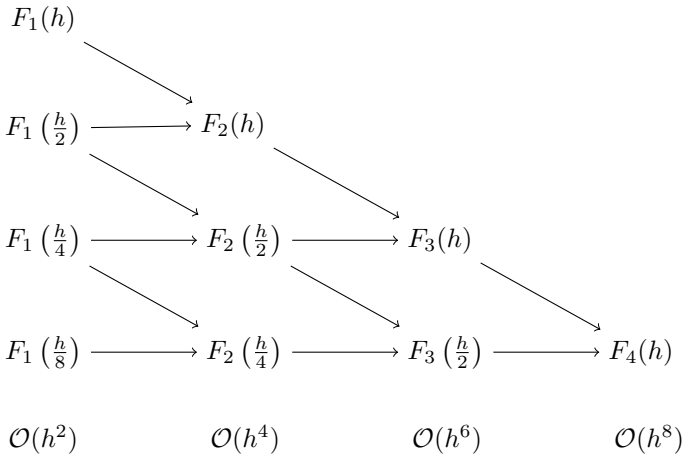
and can use this to find

$$a_2 = F_2(h) + \boxed{\frac{F_2\left(\frac{h}{2}\right) - F_2(h)}{1 - \frac{1}{16}}},$$

where the boxed portion is $F_3(h)$ and has error $\mathcal{O}(h^6)$. We can also write $F_2\left(\frac{h}{2}\right)$ as

$$F_2\left(\frac{h}{2}\right) = F_1\left(\frac{h}{2}\right) + \frac{F_1\left(\frac{h}{4}\right) - F_1\left(\frac{h}{2}\right)}{1 - \frac{1}{4}}, \tag{4.1}$$

which is just evaluations of the center differencing algorithm $F_1(h)$ for various values of h. We can see this procedure in a more organized, tabulated way:



Essentially, Richardson extrapolation lets us cancel errors with basic arithmetic and algebra to improve our estimate in any iterative algorithm involving a step size.

4.3 Improving the accuracy of numerical differentiation

We saw that there are multiple ways to calculate a derivative given 3 points, $f(x)$, $f(x+h)$, $f(x-h)$: forward and backward differencing, both with error $\mathcal{O}(h)$, and center differencing, with error $\mathcal{O}(h^2)$. To do better than $\mathcal{O}(h^2)$ via just adding more function calls, we use a Lagrange interpolating polynomial. For n+1 points, we can approximate the function with an nth order Lagrange interpolating polynomial:

$$f(x) = \sum_{k=0}^{n} f(x_k) L_{n,k}(x) + \boxed{\frac{(x - x_0)\ldots(x - x_n)}{(n+1)!} f^{(n+1)}(\xi(x))},$$

where the boxed term is defined as the error term $E(x)$. It is the (n+1)th order term of a Taylor expansion for an nth order polynomial, and represents our truncation error—the error in our algorithm due to using a finite representation of the function. Because this is an interpolating polynomial, we know that $E(x) = 0$ when passing through a point.

We can easily differentiate $f(x)$ defined this way:

$$f'(x) = \sum_{k=0}^{n} f(x_k) L'_{n,k}(x) + E'(x).$$

We evaluate this at a given point:

$$f'(x_j) = \sum_{k=0}^{n} f(x_k) L'_{n,k}(x_j) + E'(x_j).$$

Note that $L'_{n,k}(x_j)$ is not strictly 0—it would be if we were evaluating the polynomial, but we are evaluating its derivative. We know how to differentiate this. We also know how to differentiate $E(x)$, at least to leading order. We can bound the error term by choosing a value for ξ where we know that $f^{(n+1)}(\xi(x))$ is always less than or equal to our bound, and then treat the term as a constant whose value is its upper bound:

$$E'(x_j) = \frac{f^{(n+1)}(\xi_j)}{(n+1)!} \prod_{\substack{i=0 \\ i \neq j}}^{n} (x_j - x_i).$$

This gives us a way to calculate the derivative at any point to whatever order we want just by constructing a Lagrange interpolating polynomial.

We take as an example n=2, which corresponds to 3 (evenly spaced) points, since we have been using n+1 throughout. Our points are as follows:

$$x_0$$

$$x_1 = x_0 + h$$

$$x_2 = x_1 + h = x_0 + 2h.$$

We now construct our Lagrange interpolating polynomial for this set of points:

$$L_{2,0} = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} \quad \rightarrow \quad L'_{2,0}(x_0) = \frac{x_0 - x_1 + x_2 - x_2}{(x_0 - x_1)(x_0 - x_2)} = -\frac{3}{2h}.$$

Similarly,

$$L'_{2,1}(x_0) = \frac{2}{h} \qquad \text{and} \qquad L'_{2,2} = -\frac{1}{2h}.$$

We can use this to find that

$$f'(x_0) = \frac{1}{h}\left(\frac{-3}{2} f(x_0) + 2f(x_1) - \frac{1}{2} f(x_2)\right) - \frac{h^2}{3} f'''(\xi_0).$$

We can do the same for the other points:

$$f'(x_0 + h) = \frac{1}{h}\left(\frac{-1}{2} f(x_0) + \frac{1}{2} f(x_0 + 2h)\right) - \frac{h^2}{6} f'''(\xi_1)$$

$$f'(x_0 + 2h) = \frac{1}{h}\left(\frac{1}{2} f(x_0) - 2f(x_0 + h) + \frac{3}{2} f(x_0 + 2h)\right) - \frac{h^2}{3} f'''(\xi_2).$$

We then shift everything such that we are calculating the derivative at the same point in all 3 expressions: this

leaves us with three expressions for $f'(x_0)$.

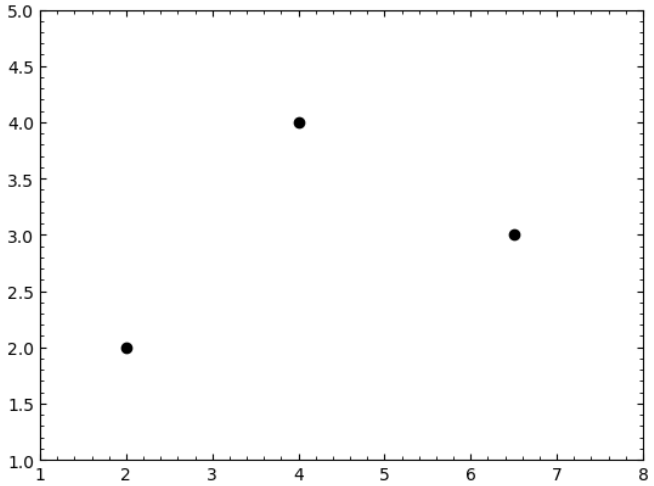
$$f'(x_0) = \frac{1}{2h}\left(-3f(x_0) + 4f(x_0+h) - f(x_0+2h)\right) - \frac{h^2}{3}f'''(\xi_0)$$

$$f'(x_0) = \frac{1}{2h}\left(f(x_0+h) - f(x_0-h)\right) - \frac{h^2}{6}f'''(\xi_1)$$

$$f'(x_0) = \frac{1}{2h}\left(f(x_0-2h) - 4f(x_0-h) + 3f(x_0)\right) - \frac{h^2}{3}f'''(\xi_2).$$

We recognize the second equation as center differencing, and see that the first and third are the same if we substitute in $h = -h$. All of these have errors that scale with h^2, but the second equation has half the contribution from error that the other two have.

To understand what these expressions are doing, we imagine having three points, and we want the derivative at each of those three:



We can use center differencing at the center point, since we know the value at both of the point's neighbors. We can't use center differencing for either the first or the third point, because only know two neighbors to one side, instead of one neighbors to both sides. Trying to differentiate at either of these points, we could fall back onto forward or backward differencing, except that their errors scale as h rather than h^2. The first and third equations derived above allow us to find the derivative at the first and third points seen here, respectively, while still having an order that scales as h^2. Thus, we have found three algorithms for equally spaced points that give us all possible derivatives for three points with errors that scale as h^2. We have seen that we can do something like this with Lagrange interpolating polynomials, but it was somewhat tedious. Richardson extrapolation is so useful because it gives us an organized way to get error to scale to a particular order for whatever algorithm we are using.

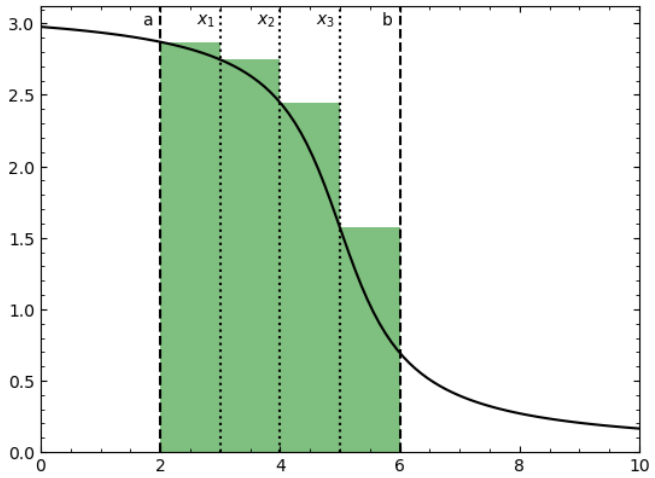
5 Numerical Integration

5.1 Techniques in numerical integration

Recall the Reimann sum from calculus:

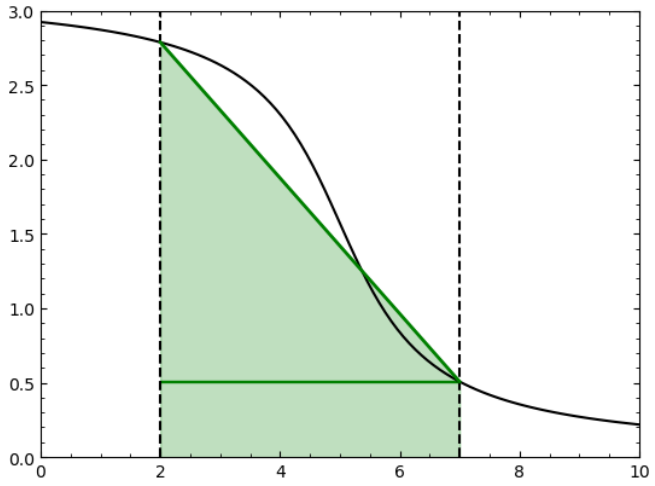
$$\int_a^b f(x)dx \approx \sum_{k=1}^{n} f(x_k)(x_k - x_{k-1}) \qquad \text{where } a \equiv x_0 < x_1 < x_2 < \cdots < x_n \equiv b$$

Geometrically, we can understand this as approximating an integral as the sum of areas of rectangles.



This is good way to get a rough idea of what we expect our integral to be, but is overall a poor algorithm for numerical situations where dx does not approach 0.

A better approach is to use the trapezoid rule, which linearly approximates the function between each point. We consider the simplest version, where we have some function and want to evaluate the integral from a to b. We know how to find the area of a trapezoid, and can find it even more easily if we split our resulting trapezoid up into a rectangle and a triangle.



We can then approximate our integral as

$$I = \int_a^b f(x)dx \approx A_{\text{rectamgle}} + A_{\text{triangle}},$$

where

$$A_{\text{rectamgle}} = (b-a)(f(b))$$

$$A_{\text{triangle}} = \frac{1}{2}(b-a)(f(a)-f(b))$$

so that our integral becomes

$$I = \int_a^b f(x)dx \approx (b-a)(f(b)) + \frac{1}{2}(b-a)(f(a)-f(b)) = \frac{1}{2}(b-a)(f(a)+f(b).$$

We can improve this estimate of the integral by approximating at a higher order than linear. Simpson's rule is for quadratic approximation, and is given by

$$\int_{x_0}^{x_2} f(x)dx = \frac{h}{3}(f(x_0)+4f(x_1)+f(x_2)) \qquad \text{where } x_1 = x_0+h, \quad x_2 = x_0+2h.$$

We can also approximate using a Lagrange interpolating polynomial—this is known as Newton-Côtes integration. Here, we integrate the Lagrange interpolating polynomial:

$$I = \int_a^b f(x)dx \qquad \text{let } f(x) = \sum_{k=0}^{n} f(x_k)L_{n,k}(x) + E_n(x)$$

then

$$I = \sum_{k=0}^{n} f(x_k) \boxed{\int_a^b L_{n,k}(x)dx} + \boxed{\int_a^b \prod_{j=0}^{n}(x-x_j)\left(\frac{f^{(n+1)}(\xi_n)}{(n+1)!}\right)}$$

where the term boxed in green consists entirely of known numbers (we can integrate that!) and the term boxed in blue is the error term. This simplifies to

$$I = \sum_{k=0}^{n} C_k f(x_k) + \text{error}, \tag{5.1}$$

where C_k is just some coefficient we know how to find from integrating the Lagrange interpolating polynomial.

We look at an example for which n=1 (so this must be the trapezoid rule). We let $x_0 = a$, $x_1 = b$, and $h = b - a = x_1 - x_0$. Recall that

$$L_{1,0} = \frac{x - x_1}{x_0 - x_1} \qquad \text{and} \qquad L_{1,0} = \frac{x - x_0}{x_1 - x_0}.$$

We use this to find our coefficients C_k:

$$C_k = \int_a^b L_{n,k}(x)dx$$

$$C_0 = \int_{x_0}^{x_1} L_{1,0}(x)dx = \int_{x_0}^{x_1} \frac{x - x_0}{x_0 - x_1} = -\frac{1}{h}\int_{x_0}^{x_1} x - x_1 \; dx$$

$$= -\frac{1}{h}\left[\frac{1}{2}x^2 - x_1 x\right]_{x_0}^{x_1} = \frac{1}{2h}(x_1 - x_0)^2 = \frac{1}{2}h$$

$$C_1 = \int_{x_0}^{x_1} l_{1,1}(x)dx = \frac{1}{2}h.$$

Plugging these coefficients into Eqn 5.1 yields

$$I = \frac{h}{2}(f(x) + f(1))$$

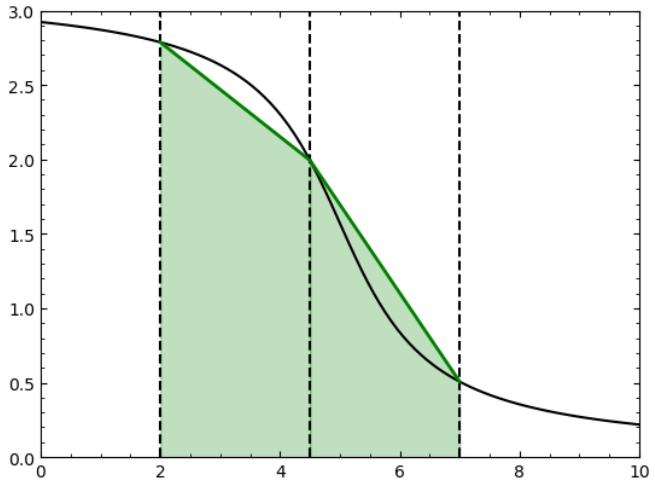
which, as expected, is simply the trapezoid rule. We also know the behavior of the error term for our Lagrange interpolating polynomial, so we can find the truncation error of the trapezoid rule:

$$\int_{x_0}^{x_1} (x - x_0)(x - x_1)\left(\frac{f''(\xi)}{2}\right) dx = -\frac{1}{12}h^3 f''(\xi).$$

We could continue doing this with higher order polynomials, but that becomes quite tedious, and the Lagrange interpolating polynomial breaks down at sufficiently high order. To get around this, we integrate piecewise, using a composite method. It's fine that this isn't smooth; integrals don't care about smoothness.

5.2 Composite methods

We can think about the composite trapezoid rule, for instance, as a Reimann sum but with trapezoids:



Here, $a = x_0$, $b = x_2$, and $x_1 = \frac{a+b}{2} = x_0 + h$, so that $h = \frac{b-a}{2}$. We can then write out the composite trapezoid rule:

$$I = I_1 + I_2 = \frac{h}{2}(f(x_0) + f(x_1)) + \frac{h}{2}(f(x_1) + f(x_2))$$

$$= \frac{h}{2}(f(x_0) + 2f(x_1) + f(x_2))$$

$$= h\left(\frac{1}{2}f(x_0) + f(x_1) + \frac{1}{2}x_2\right).$$

For the composite trapezoid rule, we see full contribution from all points in the middle, but only half contribution from the endpoints. We can generalize this for n regions (so n+1 points):

$$I = h\left(\frac{1}{2}f(x_0) + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{1}{2}f(x_n)\right) + \mathcal{O}(h^2)$$

where $h = \frac{b-a}{n}$, because the total area is divided into n regions. The error scales as h^2, so you can improve the error by making h smaller, which you can do by increasing the number of regions in the total area (and thereby decreasing the size of each individual region). We can also apply this concept to higher order interpolation, such as Simpson's rule.

Note that composite methods are an algorithm for calculating a quantity as a function of step size—we can use Richardson extrapolation to improve the accuracy of our estimates. Combining a composite method with Richardson extrapolation is called **Romberg integration**. We can apply Romberg integration using the trapezoid rule. We first choose

$$h \equiv \frac{b-a}{2^{k+1}}.$$

Notice that for a single step, $k = 1$, our step size is $h_1 = b - a$, which is the step size for the standard trapezoid rule. Then our first step of Romberg integration is given by

$$R_{k,1} \equiv \frac{h_k}{2}\left(f(a) + f(b) + 2\sum_{j=1}^{2^{k-1}-1} f(a + jh_k)\right).$$

This is just the composite trapezoid rule written differently than above, and $R_{k,1}$ is equivalent to what we would have called $F_1\left(\frac{h}{2^{k-1}}\right)$ when we discussed Richardson extrapolation. We know that the error in the composite

trapezoid rule scales as h^2, so we can write

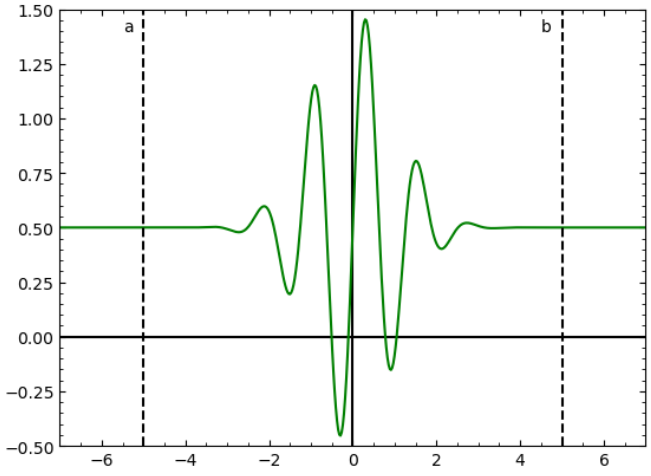
$$R_{k,1} = a_1 + \alpha_1 h_k^2$$

and apply Richardson extrapolation from here (doing so is left as an exercise to the reader).

However, we do not often end up using this, because we typically want to integrate actual functions we have defined, rather than pre-tabulated functions. The primary method we use is `scipy.integrate.quad()`.

5.3 How does scipy's quad method work?

`scipy.integrate.quad()` relies on **adaptive quadrature** (we can think of quadrature as integration). The idea here is that we do not always want to use the same step size. Consider the following function:



This is easy to integrate in the flat outer portions, but much harder to integrate in the central, oscillatory part. Picking a fixed step size that adequately samples the inner region would result in having far more data than necessary (and thus doing far more computations than necessary) for the outer region, but sampling loosely for the outer region means having insufficient information for the inner region. Adaptive quadrature adjusts the step size to the region. We can model this algorithm with the trapezoid method: bisect your region until you've found a step size such that the sum of the trapezoids' areas from the next bisection is roughly equal to the area of the current trapezoid. When you have the right step size for one region, you move onto the next. This allows you to only subdivide the portions of the function that need that level of detail.

5.4 Multidimensional Integrals

There are routines built into `scipy` to do this, which we would have to examine in more detail to use in practice, but we just want to briefly understand conceptually how they work. We examine some double integral where the outer integral is in x and the inner is in y:

$$I = \int_a^b dx \boxed{\int_{c(x)}^{d(x)} f(x,y)dy}.$$

We notice that after doing the boxed integral, it is just some function $g(x)$, so we can rewrite our outer integral as

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

Yes, $g(x)$ also happens to have an integral, but if we're using `quad()`, for example, it doesn't care; it will just call $g(x)$ and be on its way. We can think of this as just chaining together multiple one-dimensional integrals in a vaguely recursive way.

6 Differential Equations

6.1 ODE Initial Value Problems

We begin with a first order linear differential equation that has some parameter $\alpha > 0$. This can be solved by just integrating it, and doing so for a few varieties of first order linear diffeqs we see there are three types of solutions:

$$\frac{dy}{dt} + \alpha y = 0 \quad \rightarrow \quad y(t) = y_0 e^{-\alpha t} \qquad \text{(decyaing solution)}$$

$$\frac{dy}{dt} - \alpha y = 0 \quad \rightarrow \quad y(t) = y_0 e^{\alpha t} \qquad \text{(growing solution)}$$

$$\frac{dy}{dt} \pm i\omega y = 0 \quad \rightarrow \quad y(t) = y_0 e^{\mp \omega t} \qquad \text{(oscillating solution)}$$

Every technique we've talked about for differentiation and integration can be used to solve differential equations.

A simple technique widely used as a building block for solving differential equations is Euler's method. Consider

$$\frac{dy}{dt} = f(y(t), t).$$

We substitute in our forward differencing expression for $\frac{dy}{dt}$:

$$\frac{y(t + \Delta t) - y(t)}{\Delta t} = f(y(t), t).$$

We can solve this for y one time step further along—this is referred to as an **Euler step**:

$$y(t + \Delta t) = y(t) + \Delta t \; f(y(t), t)$$

This lets us evolve the system forward in time based just on information from the current time. To do this, we need some initial condition $y(t_0)$. We introduce some notation to move forward more easily:

$$t_n \equiv t_0 + n\Delta t \quad \rightarrow \quad t_n + 1 = t_n + \Delta t,$$

so that if we begin at t_0, then $t_1 = t_0 + \Delta t$. We can then rewrite our Euler step as

$$y(t_1) = y(t_0) + \Delta t \; f(y(t_0), t_0)$$

or, in shorthand,

$$y_1 = y_0 + \Delta t \; f_0$$

which generalizes to

$$y_{n+1} = y_n + \Delta t \; f_n.$$

We know that forward differencing has an error that scales as Δt, but we have multiplied it by Δt here, so that Euler's method has error $\mathcal{O}(\Delta t^2)$.

Is Euler's method stable? Evolving several time steps forward means building approximations upon approximations, so does error grow out of control or does it converge? The errors in our evolution propagate as

$$y_n = y_{n,\text{true}} + \delta y_n.$$

Including error, we can rewrite Euler's method as

$$y_{n+1,\text{true}} + \delta y_{n+1} = y_{n,\text{true}} + \delta y_n + \Delta t\ f(y_{n,\text{true}} + \delta y_n, t_n). \tag{6.1}$$

For small δy_n, we can Taylor expand the expression around $y_n = y_{n,\text{true}}$ or $\delta y_n = 0$:

$$f(y_{n,\text{true}} + \delta y_n, t_n) = f(y_{n,\text{true}}, t_n) + \left.\frac{\partial f}{\partial y}\right|_{(y_{n,\text{true}}, t_n)} \delta y_n + \dots$$

where we truncate all higher order terms. We can plug this into Eqn 6.1 to find

$$\boxed{y_{n+1,\text{true}}} + \boxed{\delta y_{n+1}} = \boxed{y_{n,\text{true}}} + \boxed{\delta y_n} + \Delta t \left(\boxed{f(y_{n,\text{true}}, t_n)} + \boxed{\left.\frac{\partial f}{\partial y}\right|_{(y_{n,\text{true}}, t_n)} \delta y_n} \right)$$

where the green terms all involve the true value, and the blue terms are all small. What we have done here is a **perturbation expansion:** considering a small perturbation around some value, where the unperturbed values satisfy our expression. Since the unperturbed values have no error,

$$y_{n+1,\text{true}} = y_{n,\text{true}} + \Delta t\ f(y_{n,\text{true}}, t_n)$$

cancels, and leaves us with only the small quantities. This is always true for a perturbation expansion. Thus, we end up with

$$\delta y_{n+1} = \left(1 + \Delta t\ \left.\frac{\partial f}{\partial y}\right|_n\right) \delta y_n$$

which we can also write as

$$\delta y_{n+1} = \beta \delta y_n \qquad \text{where } \beta = 1 + \Delta t\ \left.\frac{\partial f}{\partial y}\right|_n$$

So error scales to future steps according to β. **Our algorithm is stable if the error doesn't grow: $|\beta| \leq 1$.** The error bound for Euler's method to be stable is then

$$-1 \leq 1 + \Delta t\ \left.\frac{\partial f}{\partial y}\right|_n \leq 1$$

which we can split into two cases:

1. $\Delta t\ \left.\frac{\partial f}{\partial y}\right|_n \leq 0$ The derivative must be negative, so we must have a decaying solution.
2. $-2 \leq \Delta t\ \left.\frac{\partial f}{\partial y}\right|_n$ which tells us that $\Delta t \leq \frac{2}{\left|\left.\frac{\partial f}{\partial y}\right|_n\right|}$

In other words, Euler's method is stable for a decaying solution with sufficiently small Δt. For a first order linear ODE, a function with a decaying solution is just $f = \alpha y$, so that $\frac{df}{dy} = -\alpha$. This is true at any n. Thus, we need $\boxed{\Delta t \leq \frac{2}{\alpha}}$ for a stable algorithm.

The easiest way to think about a growing solution is as if we are going back in time, and then treating it as a

decaying solution.

A function with an oscillating solution is $f = \pm i\omega y$, and $\frac{df}{dy} = \pm i\omega$. To acocunt for imaginary solutions, we now say that stability requires

$$|\beta|^2 \leq 1.$$

For an oscillatory solution,

$$\beta = 1 + \left.\frac{\partial f}{\partial y}\right|_n \Delta t = 1 \pm i\omega t$$

and then

$$|\beta|^2 = 1 + \omega^2 \Delta t^2 \leq 1,$$

which is never true, so Euler's method is unstable for oscillatory solutions. Euler's method is **vectorized:** it can solve a system of first order differential equations.

We now consider higher order differential equations. Newton's second law is an example of a higher order diffeq:

$$\vec{F}(\vec{r}, \vec{v}, t) = m\vec{a} = m\frac{d^2\vec{r}}{dt^2}$$

which we can rewrite as

$$\frac{d^2\vec{r}}{dt^2} = \frac{\vec{F}(\vec{r}, \vec{v}, t)}{m}.$$

We want to turn this into a system of first order equations; all the algorithms we will discuss work on first order differential equations. The natural thing to do here is to say:

$$\frac{d^2\vec{r}}{dt^2} = \frac{d\vec{v}}{dt} = \frac{F}{m}.$$

This is a first order differential equation. This is the system we will use moving forward: we get to a first order equation by defining the first order to be something and then taking its derivative (we do this as many times as necessary to reduce a higher order differential equation to a system of first order diffeqs). For our above example, our system of equations is:

$$\frac{dx}{dt} = v_x \qquad \frac{dy}{dt} = v_y \qquad \frac{dz}{dt} = v_z$$

$$\frac{dv_x}{dt} = \frac{F_x}{m} \qquad \frac{dv_y}{dt} = \frac{F_y}{m} \qquad \frac{dv_z}{dt} = \frac{F_z}{m}$$

To uniquely specify this, we need initial conditions $\vec{r}(0)$ and $\vec{v}(0)$; these are typically given.

6.2 Runge-Kutta Methods

We can't use Euler's method for oscillatory solutions, so fourth-order Runge-Kutta methods (a general class of integrators) are the standard ODE solver. We begin by examining a second order ODE. To improve the accuracy of Euler's method, we want to use more information. We do this by, instead of taking a full Euler step blindly, taking a trial step and using it to improve our guess. We let

$$k_1 = \Delta t\ f(y_n, t_n) \qquad \leftarrow \qquad \text{full Euler step}$$

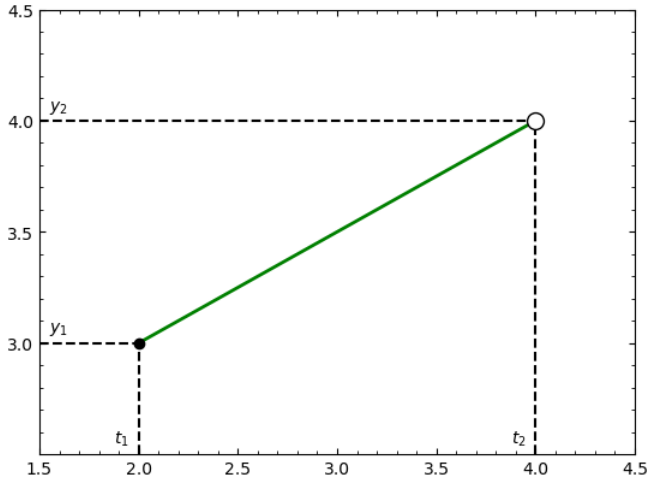
and

$$k_2 = \Delta t\ f\left(y_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2}\right) \qquad \leftarrow \qquad \text{half Euler step}$$

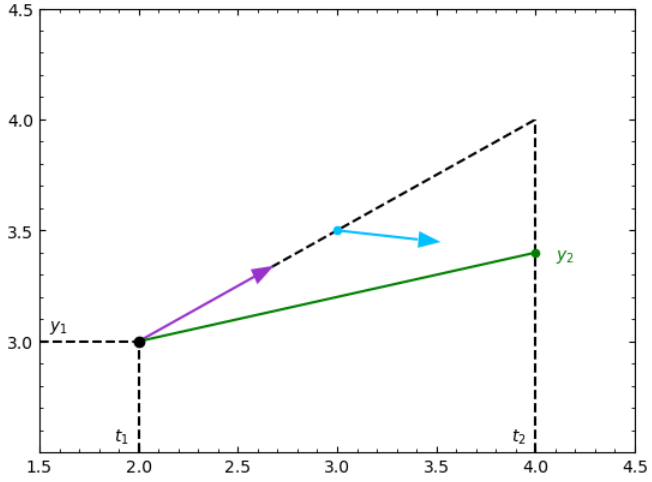
and use this to have

$$y_{n+1} = y_n + k_2 + \mathcal{O}(\Delta t^3),$$

which we call second order since we evaluate $\frac{dy}{dt}$ to the second order. Consider beginning from one point and wanting to find the y value a step further. In Euler's method, we estimate the derivative at t_1 and then linearly approximate—an Euler is a linear approximation of the derivative.



With a Runge-Kutta method, the first step is the same: we approximate the derivative at t_1, but then take a half step, find the derivative there, and use that to do a second linear interpolation that gives a better estimate of y_2. So we begin with an Euler step and evaluate the goodness of that estimate based on the half-Euler step, which implies a new estimate of the derivative.



The fourth order Runge-Kutta method is a default algorithm for differential equations, and is implemented using the same idea as the second order method we examined above—we use multiple Euler steps to refine our

estimate. Here, we use four:

$$k_1 = \Delta t\ f(y_n, t_n) \qquad \leftarrow \qquad \text{full Euler step}$$

$$k_2 = \Delta t\ f\left(y_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2}\right) \qquad \leftarrow \qquad \text{half step}$$

$$k_3 = \Delta t\ f\left(y_n + \frac{k_2}{2}, t_n + \frac{\Delta t}{2}\right) \qquad \leftarrow \qquad \text{half step after } k_2$$

$$k_4 = \Delta t\ f\left(y_n + k_3, t_n + \Delta t\right) \qquad \leftarrow \qquad \text{full step}$$

which we can combine to find that

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + \mathcal{O}(\Delta t^5).$$

It is rarely justified to work at a higher order than fourth order.

Improvements that can be made to this algorithm include adaptive step sizes (implementing adaptive quadrature; `scipy.integrate.solve_ivp()` does this) and using Richardson extrapolation to decrease the error in our estimates (this is done in Bulirsch-Stoer methods).

We also note than any numerical problem is different from the continuous version of that problem. Conserved quantities are very important, but we can lose information when discretizing for numerical solutions—numerican algorithms do not tend to conserve energy, momentum, angular momentum, or other quantities we care about conserving in physics contexts. For a given problem, we may need algorithms that explicity conserve the most important quantity in that problem. The staggered leap frog method conserves energy in a discretized form. Symplectic methods (based on Hamiltonian mechanics) conserve phase space (this can roughly be thought of as momentum/position space, and is relevant for questions like "is the solar system stable").

7 Interlude—Midterm Review

7.1 Topics covered thus far

- Limited precision
- Root finding: bisection, secant method, Newton-Rhapson method; the importance of bracketing
- Polynomial interpolation: Lagrange interpolating polynomial, piecewise polynomial interpolation, splines
- Differentiation: forward, backward, and center differencing; Richardson extrapolation
- Integration: Newton-Côtes integration, trapezoid and cumulative trapezoid methods, adaptive quadrature
- Differential equations: Euler's method and the Euler step, stability, Runge-Kutta methods

7.2 Selected questions and answers

7.2.1 Integer vs floating point operations

```
x = 4 # integer
x = 4. # float (explicitly specified to the computer by the decimal)
```

In both cases there are limits on sizes and ranges the values can have, and there can sometimes be a computational difference between integers and floating point numbers. For our purposes, they should be interchangeable, but it is always advisable to be careful. `numpy` arrays only store one data type (as opposed to built-in Python lists, which can store a variety), so specifying a single value as being a float will force the whole array to be floats:

```
np.ndarray([0,1,2]) # integers
np.ndarray([0.,1,2]) # floats
```

7.2.2 Lagrange interpolating polynomial vs splines

Let's say we want to interpolate between 3 points. A Lagrange interpolating polynomial will construct a single quadratic function that passes through all three points, while a spline will construct a piecewise cubic through each set of two points, and then match derivatives to ensure smoothness and make sure the cubics are uniquely defined.

The two methods are very different from one another, so it comes down to which method is better for any particular situation. Lagrange interpolating polynomials can be useful for low order functions (since it is just a Taylor expansion). They are also useful if you just want a polynomial through your points. However, usually a spline is the better option, since is more accurate for more data points, where a Lagrange interpolating polynomial will oscillate wildly. So we generally prefer splines, but should not regect Lagrange interpolating polynomials outright.

7.2.3 General algorithm for Richardson extrapolation

We want some $Q = f(h)$ where h is a nonzero step size. Then

$$Q = f(h) + \boxed{\mathcal{O}(h^r)}.$$

The goal of Richardson extrapolation is to minimze the boxed error term. To do this, we evaluate Q at h and $\frac{h}{2}$ and see hoe the error scales; we then find some intelligent way to combine these so that the error term cancels. Bam! We've gotten rid of the error. We're then left with $\mathcal{O}(h^s)$ where $s > r$, and we repeat the procedure to cancel out this error and end up with an error that scales with some value $> s$. This is roughly the same idea as combing forward and backward differencing to get center differencing, which has better error scaling.

8 Linear Algebra

8.1 Basics of linear algebra

Think about a single linear equation—solving this is just basic algebra.

$$7x = 21 \quad \rightarrow \quad x = 3.$$

Alternatively, we could think of this as

$$x = (7^{-1})\ 21 \qquad \text{where } (7^{-1}) \times 7 = 7 \times (7^{-1}) = 1.$$

In general, the solution to $ax = b$ is given by $x = a^{-1}b$. This is a unique solution, except for when $a = 0$. In such cases, we say that a is **singular**.

Linear algebra means extending this idea to a system of equations:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$$

where we assume a and b are known. Writing all of this out is a pain, so we introduce shorthand: $\mathbf{A}\vec{x} = \vec{b}$, where

$$\mathsf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{23} \end{bmatrix} \qquad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

When you multiply a matrix by a vector, the output is a vector. This process is a linear transformation—taking linear combinations of x to create a new object ($\vec{b}$) that's a vector.

$$b_i = (\mathsf{A}\vec{x})_i = \sum_j \mathsf{A}_{ij}x_j.$$

We define a vector as something that transforms like a vector (note: I have no idea what this means. Ask Copi or a math major). In Newtonian mechanics, a transformation of interest is rotation. Consider $\vec{F} = m\vec{a}$: we can orient our free body diagram coordinate system as we please. We can independently choose coordinate systems for multiple free body diagrams and apply Newton's second law to each of them independently because of how vectors transform under rotation.

Vectors, for computers, are not inherently the same as arrays, but we can use arrays to represent them.

The dot product,

$$\vec{a} \cdot \vec{b} = \sum_i a_i b_i,$$

produces a scalar, and is invariant under rotation. Matrix multiplication is just a dot product, so it, too, reduces the dimensionality of the output:

$$\mathsf{C} = \mathsf{AB} \quad \rightarrow \quad C_{i,j} = \sum_k A_{ik} B_{kj}.$$

Some other notable matrix properties are:

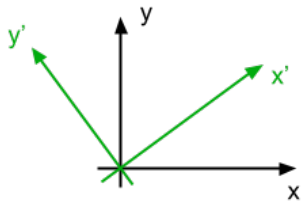
- **Determinant:** the determinant of a matrix turns the matrix into a scalar. This is easy for a 2x2 matrix:

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}.$$

  A **singular matrix** has a determinant equal to 0.
- **Inverse:** the inverse of a matrix exists for square, non-singular matrices, and satisfies $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{1}$, where 1 is the identity matrix.
- **Trace:** the trace of a matrix is the sum of the diagonal, and is another scalar we can define for our matrix.
- **Transpose:** the transpose of a matrix reflects the original matrix across the diagonal (i.e., swaps rows and columns): $\mathbf{A}^{\mathsf{T}}{}_{ij} = a_{ji} = \mathbf{A}_{ji}$.

There are several important types of matrices:

- **Symmetric:** $\mathsf{A}^{\mathsf{T}} = \mathsf{A}$
- **Orthagonal:** $\mathsf{A}^{-1} = \mathsf{A}^{\mathsf{T}}$
  - An example of an orthagonal matrix is a rotation matrix. Note: we want to avoid calculating inverses numerically.
- **Rotation matrix:** this preserves the orthagonality and size/length of anything it is multiplied by; orthagonal rotation changes vectors' direction but not magnitude.



8.2 Newton-Rhapson Method

We are revisiting this method because it uses linear algebra. We saw it in the context of one-dimensional root finding, but concluded that algorithms that enforce bracketing are better. We now use it to look at a system of

nonlinear equations. Consider a system of equations

$$f_1(\vec{y}) = 0,\ f_2(\vec{y}) = 0, \ldots,\ f_n(\vec{y}) = 0 \qquad \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

We Taylor expand $f_1(\vec{y}\Delta\vec{y})$ around $\Delta\vec{y} = 0$:

$$f_1(\vec{y} + \Delta\vec{y} = f_1(\vec{y}) + \sum_j \left.\frac{\partial f}{\partial y_j}\right|_{\vec{y}} + \frac{1}{2}\sum_{j,k} \left.\frac{\partial^2 f}{\partial y_j \partial y_k}\right|_{\vec{y}} \Delta y_j \Delta y_k.$$

We truncate this at the second order, thereby linearizing it, and then use a vector-like notation to express it: we have n functions, so consider $\vec{f}(\vec{y} + \Delta\vec{y})$ a vector of functions. If we expand each of these up to linear order, we find

$$\vec{f}(\vec{y} + \Delta\vec{y}) = \vec{f}(\vec{y}) + \boxed{\mathsf{J}_{\vec{f}}(\vec{y})}\Delta\vec{y}$$

where the boxed term is the **Jacobian matrix** for $\vec{f}$ evaluated at $\vec{y}$, and defined as

$$J_{i,j} = \frac{\partial f_i(\vec{y})}{\partial y_j}.$$

We can use this for root finding: finding where $\vec{f}(\vec{y}) = 0$. We begin with some initial guess $\vec{y}_0$, where $\vec{f}(\vec{y}_0) \neq 0$. We want to improve this guess my moving by some Δy such that $f(\vec{y}_0 + \Delta\vec{y}) = 0$:

$$\vec{f}(\vec{y}_0 + \Delta\vec{y}) \approx \vec{f}(\vec{y}_0) + \mathsf{J}_{\vec{f}}(\vec{y}_0)\Delta\vec{y} = 0.$$

The only unknown in that expression is $\Delta\vec{y}$, so we solve for it:

$$\mathsf{J}_{\vec{f}}(\vec{y}_0)\Delta\vec{y} = -\vec{f}(\vec{y}_0).$$

That gives us a system of linear equations for $\Delta\vec{y}$, since even though $\vec{f}$ can be nonlinear, we have approximated it as linear. We have not yet covered how to numerically solve a linear system of equations (we'll get to that later) but we state the formal solution anyway:

$$\Delta\vec{y} = -\mathsf{J}_{\vec{f}}(\vec{y}_0)^{-1}\vec{f}(\vec{y}_0).$$

This tells us that our next guess will be given by

$$\vec{y}_1 = \vec{y}_0 + \Delta\vec{y} = \vec{y}_0 - \mathsf{J}_{\vec{f}}^{-1}\vec{f}(\vec{y}_0).$$

Unless our system truly is linear, this will still be an imperfect solution, so we iterate from there:

$$\vec{y}_2 = \vec{y}_1 - \mathsf{J}_{\vec{f}}^{-1}\vec{f}(\vec{y}_1),$$

or, more generally,

$$\vec{y}_{n+1} = \vec{y}_n - \left(\mathsf{J}_{\vec{f}}\left(\vec{y}_n\right)\right)^{-1}\vec{f}(\vec{y}_n).$$

Here we will explicitly calculate J^{-1}. This is bad! In general, we want to avoid explicitly calculating an inverse, and we will be able to do so in the future, when we have learned how to numerically solve linear systems of equations.

A note on iterative algorithms more generally: convergence is determined by new steps being sufficiently small (the tolerance parameters in many iterative algorithms we have used thus far define "sufficiently") but it is also good to have some fail-safe or maximum number of iterations, in case our convergence criterion is too stringent, or the method simply does not converge.

8.3 Linear Systems of Equations

We just covered a method of solving a linear system of equations that requires explicitly calculating the inverse of a matrix. We don't want to do this! Now we will learn how not to.

Consider the system

$$10x_1 - 7x_2 - x_3 = 8 \tag{8.1}$$

$$-3x_1 + 2x_2 + 6x_3 = 4 \tag{8.2}$$

$$5x_1 - x_2 + 5x_3 = 6 \tag{8.3}$$

To solve this system, we automate substitution.

1. Combine 8.1 and 8.2 to get rid of x_1.
2. Combine 8.1 and 8.3 to get rid of x_1. This leaves us with 2 equations and 2 unknowns (x_2 and x_3).

$$\begin{aligned} 8.2 - \left(\frac{-3}{10}\right)(8.1) \quad &\rightarrow \quad \left(2 - \frac{21}{10}\right)x_2 + \left(6 + \frac{3}{10}\right)x_3 = 4 + \frac{12}{5} \\ &= \; -\frac{1}{10}x_2 + \frac{63}{10}x_3 = \frac{32}{5} \end{aligned} \tag{8.2'}$$

$$\begin{aligned} 8.3 - \left(\frac{1}{2}\right)8.1 \quad &\rightarrow \quad \left(-1 + \frac{7}{2}\right)x_2 + \left(5 + \frac{1}{2}\right)x_3 = 6 - 4 \\ &= \; -\frac{5}{2}x_2 + \frac{9}{2}x_3 = 2 \end{aligned} \tag{8.3'}$$

3. Then combine 8.2' and 8.3' to get rid of x_2:

$$8.3' - (-25)(8.2') \quad \rightarrow \quad 162x_3 = 162 \tag{8.3"}$$

We have reduced our original system to a simpler one consisting of 8.1, 8.2', and 8.3", which has all the information we need to solve the system:

$$10x_1 - 7x_2 + x_3 = 8 \tag{8.1}$$

$$-\frac{1}{10}x_2 + \frac{63}{10}x_3 = \frac{32}{5} \tag{8.2'}$$

$$162x_3 = 162 \tag{8.3"}$$

We can solve this simpler system using backwards substitution: begin with 8.3" and then walk backwards:

$$\begin{aligned} 8.3'' \quad &\rightarrow \quad x_3 = 1 \\ 8.2' \quad &\rightarrow \quad x_2 = -10\left(-\frac{63}{10}(1) + \frac{32}{5}\right) = -1 \\ 8.1 \quad &\rightarrow \quad x_1 = \frac{1}{10}(7(1) - (1) + 8) = 0 \end{aligned}$$

which leaves us with

$$x_1 = 0 \qquad x_2 = -1 \qquad x_3 = 1.$$

This would have been a little simpler to solve if we were smarter about the substitutions we chose, but this is a simple, algorithmic way to solve a linear system of equations, and we can do it numerically.

We rewrite the system in matrix form:

$$\mathsf{A}\vec{x} = \vec{b} \qquad \text{where} \quad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \qquad \vec{b} = \begin{pmatrix} 8 \\ 4 \\ 6 \end{pmatrix}, \quad \text{and} \quad \mathsf{A} = \begin{pmatrix} 10 & -7 & 1 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{pmatrix}.$$

We can then say

$$(\mathsf{A}\vec{x})_i = \sum_{j=1}^{3} A_{ij}x_j = b_i.$$

To express this more neatly, we turn to an **augmented matrix**. Earlier, to solve the system, we had been doing operations on the coefficients of equations, which we can think of as doing operations on A. We have to do the same operations on the right hand side as the left, and an augmented matrix is useful notation for this. We'll represent these in shorthand as

$$\left[\mathsf{A}, \vec{b}\,\right] \equiv \left(\mathsf{A} \mid \vec{b}\,\right)$$

and written out as

$$\left(\begin{array}{ccc|c} 10 & -7 & 1 & 8 \\ -3 & 2 & 6 & 4 \\ 5 & -1 & 5 & 6 \end{array}\right).$$

The algorithm we worked through above is **Gaussian elimination:** applying some algorithm to reduce the aug-

mented matrix to row-echelon form, where all entries below the diagonal are 0.

$$\left[\mathsf{A}, \vec{b}\,\right] \xrightarrow[8.3-\frac{1}{2}8.1]{8.2-\frac{3}{10}8.1} \left(\begin{array}{ccc|c} 10 & -7 & 1 & 8 \\ 0 & -\frac{1}{10} & \frac{63}{10} & \frac{32}{5} \\ 0 & \frac{5}{2} & \frac{9}{2} & 2 \end{array}\right)$$

$$\xrightarrow{8.3-\frac{50}{2}8.2} \boxed{\left(\begin{array}{ccc|c} 10 & -7 & 1 & 8 \\ 0 & -\frac{1}{10} & \frac{63}{10} & \frac{32}{5} \\ 0 & 0 & 162 & 162 \end{array}\right)} \quad \leftarrow \text{row echelon form}$$

This algorithm has replaced (transformed) A to U, where U is **upper triangular:** $U_{ij} = 0 \;\; \forall \;\; i > j$. We now have a new system, $\mathsf{U} = \vec{x} = \vec{\beta}$ which can be solved with backwards substitution:

$$x_3 = \frac{\beta_3}{U_{33}}$$

$$x_2 = \frac{\beta_2 - U_{23}x_3}{U_{22}}$$

$$x_1 = \frac{\beta_1 - U_{12}x_2 - U_{13}x_3}{U_{11}}.$$

In general, for any system of this form,

$$\frac{\beta_i - \sum_{j=i+1}^{n} U_{ij}x_j}{U_{ij}}.$$

We introduce a bit more notation: the augmented matrix can be represented by $\tilde{\mathsf{A}}$. With that, we can summarize the general algorithm we encountered here as:

1. Define $\tilde{\mathsf{A}} \equiv \left[\mathsf{A}, \vec{b}\,\right]$.
2. Reduce it to row echelon form: replace the rows of $\tilde{\mathsf{A}}$:

$$\tilde{A}_j \leftarrow \tilde{A}_j - \left(\frac{\tilde{A}_{ji}}{\tilde{A}_{ii}}\right)\tilde{\mathsf{A}}_j \qquad \text{for all } j > i.$$

$\tilde{A}_{ii}$, the diagonal element, is called the **pivot**.

This algorithm can fail mathematically/analytically if the pivot is 0, or numerically if the pivot is very small.

8.4 Linear systems: pivoting

Dividing by the pivot is a notable issue with Gaussian elimination. One solution to this is **pivoting**. Recall our system:

$$10x_1 - 7x_2 - x_3 = 8 \tag{8.1}$$

$$-3x_1 + 2x_2 + 6x_3 = 4 \tag{8.2}$$

$$5x_1 - x_2 + 5x_3 = 6 \tag{8.3}$$

which we can express as $\tilde{\mathsf{A}} = \left[\mathsf{A}, \vec{b}\,\right]$. The Gaussian elimination formula to reduce it to row-echelon form is

$$\tilde{A}_j \leftarrow \tilde{A}_j - \left(\frac{\tilde{A}_{ji}}{\boxed{\tilde{A}_{ii}}}\right)\tilde{\mathsf{A}}_j \qquad \text{for all } j > i$$

where the boxed term is the pivot. We have problems for any pivot at or near 0. If we were doing substitution by hand this would be great; it would simplify the algebra. But computers aren't that smart, so to handle this, we switch the order of the equations. This is **pivoting:** swapping the rows in $\tilde{\mathsf{A}}$, so that our true algorithm is Gaussian elimination with pivoting. Pivoting generally can be used to enhance the numerical stability of the algorithm. There are many ways to choose the next pivot. A common way is to choose the **maximum pivot:** for a given i find the max value of $\left|\tilde{A_{ji}}\right|$ for $i > j$, and then swap the ith and jth rows.

Then our actual algorithm is using Gaussian elimination with some form of pivoting to reduce $\tilde{\mathsf{A}}$ to row-echelon form, and then solve the simplified system with backwards substitution. If we want to solve the same left hand side for a new right hand side, we do the exact same thing every time. This can be computationally expensive, so we turn to **decomposition**.

8.4.1 LU decomposition

The goal of any type of decomposition is to decompose our matrix into some other form. Here, we want to decompose $\mathsf{A} = \mathsf{LU}$, where L is a lower triangular matrix and U is an upper triangular matrix. We can always do this decomposition for any square matrix. This yields the matrices

$$\mathsf{U} = \begin{pmatrix} U_{00} & U_{01} & U_{02} & \dots & U_{0n} \\ 0 & U_{11} & U_{12} & \dots & U_{1n} \\ 0 & 0 & U_{22} & \dots & U_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \end{pmatrix}$$

and a lower triangular matrix in a very specific form—it has ones along the diagonal:

$$\mathsf{L} = \begin{pmatrix} 1 & 0 & 0 & \dots \\ L_{10} & 1 & 0 & \dots \\ L_{20} & L_{21} & 1 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}.$$

Suppose we've done this decomposition and now want to solve $\mathsf{A}\vec{x} = \vec{b}$. Now we have

$$\mathsf{L}\boxed{\mathsf{U}\vec{x}} = \vec{b}$$

where the boxed term is just a vector which we rename $\vec{z}$, so that our system of linear equations is given by

$$\mathsf{L}\vec{z} = \vec{b}.$$

If we multiply this out, we get

$$z_0 = b_0 \qquad \leftarrow \text{first row}$$

$$L_{10}z_0 + z_1 = b_1 \qquad \leftarrow \text{second row}$$

$$\text{etc.}$$

which we can generalize to

$$z_i = b_i - \sum_{j-0}^{i-1} L_{ij}z_j$$

where we know z_j from previous steps. Thus, after LU decomposition, we just have a system of linear equations with a lower triangular matrix of coefficients, so we can solve for $\vec{z}$ with forward substitutions. However, we wanted to solve for $\vec{x}$, not $\vec{z}$, so we use $\mathsf{U}\vec{x} = \vec{z}$, where U and $\vec{z}$ are known, which we can solve for $\vec{x}$ with backwards substitution.

This method relies on having an LU decomposition and not running into numerical issues dividing by the pivot, so the most general algorithm is **LU decomposition with pivoting**:

$$\mathsf{A} = \mathsf{PLU}$$

where P is the **permutation matrix**, which permutes the rows to handle pivoting. Specifically, a permutation matrix swaps the last two rows. For a 3x3 matrix,

$$\mathsf{P} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

and

$$\mathsf{P}\vec{v} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} v_0 \\ v_2 \\ v_1 \end{pmatrix}.$$

This is useful for multiple right hand sides because we don't have to redo the decomposition, just the basic algebra.

8.5 Eigenvalue problems

So far, we have been looking at linear transformations like $\mathsf{A}\vec{x} = \vec{b}$. Now we consider instead something like $\vec{L} = \mathsf{I}\vec{\omega}$, where I is the moment of inertia tensor. When I is just a number in introductory physics, that's because it's the moment about a single axis. If we want a moving axis of rotation, we need to know I around every axis; I tells us

this. This concept is something we will come back to a lot to understand eigenvalue problems.

Consider

$$\mathsf{A}\vec{v} = \lambda\vec{v} \tag{8.4}$$

, where λ is an eigenvalue and $\vec{v}$ is an eigenvector. Multiplying A by $\vec{v}$ just returns $\vec{v}$ scaled by some number λ. Given a matrix A, this will only happen for some choices of λ and $\vec{v}$. Thinking about this geometrically, the **eigenvectors** are the directions associated with A, which will only be scaled by the linear transformation of multiplying $\mathsf{A}\vec{v}$. We can rearrange 8.4 to

$$(\mathsf{A} - \lambda 1)\vec{v} = 0 \tag{8.5}$$

where 1 is the identity matrix. $(\mathsf{A} - \lambda 1)$ is just a matrix, so this is a system of equations. Typically there is a unique solution—the trivial one, $\vec{v} = 0$. This is the unique solution for an invertible matrix, but we want nontrivial solutions. There will not be a unique, nontrivial solution if $\mathsf{A} - \lambda 1$ is not invertible (singular). A matrix is singular when $\det(\mathsf{A} - \lambda 1) = 0$, so we're interested in these solutions. The determinant of $\mathsf{A} - \lambda 1$ will be an nth order polynomial in λ: the **characteristic polynomial**, whose roots are the eigenvalues.

8.5.1 Example

Consider

$$\mathsf{A} = \begin{pmatrix} 5 & 0 & \sqrt{3} \\ 0 & 3 & 0 \\ \sqrt{3} & 0 & 3 \end{pmatrix}.$$

We want to find the eigenvalues, so we set $\det(\mathsf{A} - \lambda 1) = 0$:

$$\begin{vmatrix} 5-\lambda & 0 & \sqrt{3} \\ 0 & 3-\lambda & 0 \\ \sqrt{3} & 0 & 3-\lambda \end{vmatrix} = 0.$$

We evaluate a 3x3 determinant "in minors" by turning it into a sum of 2x2 determinants:

$$0 = (5-\lambda)\begin{vmatrix} 3-\lambda & 0 \\ 0 & 3-\lambda \end{vmatrix} - (0)\begin{vmatrix} 0 & 0 \\ \sqrt{3} & 3-\lambda \end{vmatrix} + (\sqrt{3})\begin{vmatrix} 0 & 3-\lambda \\ \sqrt{3} & 0 \end{vmatrix}$$

$$0 = (5-\lambda)(3-\lambda)^2 - 0 + (\sqrt{3})(-\sqrt{3})(3-\lambda)$$

$$0 = (5-\lambda)(3-\lambda)^2 - 3(3-\lambda)$$

$$0 = (3-\lambda)((5-\lambda)(3-\lambda) - 3)$$

$$0 = (3-\lambda)(15 - 8\lambda + \lambda^2 - 3)$$

$$0 = (\lambda - 3)(\lambda - 2)(\lambda - 6)$$

$$\boxed{\lambda = 2, 3, 6} \quad \leftarrow \quad \text{eigenvalues of the matrix } \mathsf{A}$$

Now that we have the eigenvalues, we can find the eigenvectors. There is one eigenvector for each eigenvalue. Let $\vec{v}_j$ be an eigenvector; then,

$$\vec{v}_j = \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}.$$

This is a three component eigenvector because A is a 3x3 matrix. Eigenvectors will satisfy

$$\mathsf{A}\vec{v}_j = \lambda_j \vec{v}_j,$$

which we can use to solve for them. Writing it out in full, we get

$$\begin{pmatrix} 5 & 0 & \sqrt{3} \\ 0 & 3 & 0 \\ \sqrt{3} & 0 & 3 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \lambda_j \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}.$$

Multiplying this out gives us

$$5\alpha + \sqrt{3}\,\gamma = \lambda_j \alpha$$

$$3\beta = \lambda_j \beta$$

$$\sqrt{3}\,\alpha + 3\gamma = \lambda_j \gamma$$

which simplifies to

$$(\lambda_j - 5)\alpha = \sqrt{3}\,\gamma$$

$$\lambda_j = 3 \qquad \text{or} \qquad \beta = 0$$

$$\sqrt{3}\,\alpha = (\lambda_j - 3)\gamma$$

which is a system we can solve for each of our three eigenvalues.

We begin with the $\lambda = 2$ case, where our system is

$$-3\alpha = \sqrt{3}\,\gamma \qquad\qquad \beta = 0 \qquad\qquad \sqrt{3}\,\alpha = -\gamma.$$

Combining the first and third equations, we find

$$-3\alpha = -3\alpha$$

and realize that the first and third equations are rearranged versions of the same thing, which leaves us with two equations for three unknowns. We should have expected this, though; we set the determinant equal to 0 to find the eigenvalues, but doing that forced the equations not to be linearly independent. Note also that $\vec{v}$ is not a unique solution; any linear scaling of it ($2\vec{v}$, $6\vec{v}$, etc.). **Eigenvectors are unique directions, not unique vectors.** Since we can always rescale $\vec{v}$, we can choose a normalization to provide another constraint on our system. The most common choice is to make the eigenvectors unit vectors, so that

$$\alpha^2 + \beta^2 + \gamma^2 = 1.$$

Using this constraint along with $\gamma = -\sqrt{3}\ \alpha$ and $\beta = 0$ (which we know, since λ here is not 3), we find

$$\alpha^2 + \beta^2 + \gamma^2 = 1$$

$$\alpha^2 + 3\alpha^2 = 1$$

$$4\alpha^2 = 1$$

$$\alpha = \pm\frac{1}{2}.$$

We can just choose the positive case, since we are allowed to rescale as we please. Then $\gamma = -\frac{\sqrt{3}}{2}$, so that for $\lambda_1 = 2$,

$$\vec{v}_1 = \begin{pmatrix} \frac{1}{2} \\ 0 \\ -\frac{\sqrt{3}}{2}. \end{pmatrix}$$

For $\lambda_2 = 3$:

$$-2\alpha = \sqrt{3}\ \gamma$$

$$\beta \text{ can be anything, since } \lambda = 3$$

$$\sqrt{3}\ \alpha = 0$$

Substituting $\alpha = 0$ into the first equation, we find that $\gamma = 0$, and then turn to our normalization constraint to find β:

$$\alpha^2 + \beta^2 + \gamma^2 = 1$$

$$\beta^2 = 1$$

$$\beta = 1$$

so that for $\lambda_2 = 3$,

$$\vec{v}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}.$$

Finally, for $\lambda_3 = 6$, we can simplify our system to

$$\beta = 0 \qquad \alpha = \sqrt{3}\ \gamma \qquad \sqrt{3}\ \alpha = 3\gamma$$

so we can use the normalization constraint to find that

$$3\gamma^2 + \gamma^2 = 1$$

$$\gamma = \frac{1}{2}$$

$$\alpha = \frac{\sqrt{3}}{2}$$

so that for $\lambda_3 = 6$,

$$\vec{v}_2 = \begin{pmatrix} \frac{\sqrt{3}}{2} \\ 0 \\ \frac{1}{2} \end{pmatrix}.$$

We now construct a matrix B which has the eigenvectors as columns:

$$\mathsf{B} = \begin{pmatrix} \vec{v}_1 & \vec{v}_2 & \vec{v}_3 \\ \vdots & \vdots & \vdots \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \\ 0 & 1 & 0 \\ -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \end{pmatrix}.$$

This is useful to us because of the **similarity transform: $\mathsf{B}^{-1}\mathsf{AB}$**, which diagonalizes the matrix. However, it requires knowing the inverse, which we do not want to calculate. We can get around this by knowing that our matrix A was symmetric, and **for a symmetric matrix (where $\mathsf{A} = \mathsf{A}^T$, B is orthagonal**, where for an **orthagonal** matrix, $\mathsf{B}^{-1} = \mathsf{B}^T$. Thus, for our symmetric matrix, the similarity transform is given by $\mathsf{B}^T\mathsf{AB}$:

$$\begin{pmatrix} \frac{1}{2} & 0 & -\frac{\sqrt{3}}{2} \\ 0 & 1 & 0 \\ \frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 5 & 0 & \sqrt{3} \\ 0 & 3 & 0 \\ \sqrt{3} & 0 & 3 \end{pmatrix} \begin{pmatrix} \frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \\ 0 & 1 & 0 \\ -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{2} & 0 & -\frac{\sqrt{3}}{2} \\ 0 & 1 & 0 \\ \frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 1 & 0 & 3\sqrt{3} \\ 0 & 3 & 0 \\ -\sqrt{3} & 0 & 3 \end{pmatrix}$$

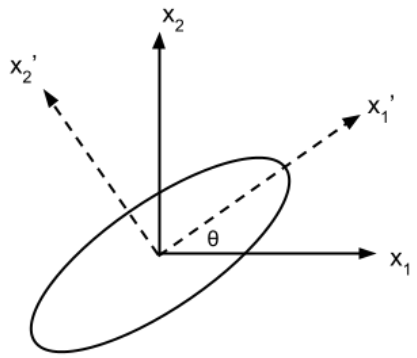
$$= \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 6 \end{pmatrix} \quad \leftarrow \text{matrix of eigenvalues}$$

In general, $\mathsf{B}^{-1}\mathsf{AB}$ is equal to a diagonal matrix of eigenvalues D:

$$\mathsf{D} = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_n \end{pmatrix}.$$

8.6 Geometric interpretation of eigenvalues

Consider an ellipse:



The general form of the equation for an ellipse is

$$\alpha_1 x_1^2 + 2\beta x_1 x_2 + \alpha_2 x_2^2 = \gamma$$

and the standard form is

$$\left(\frac{x_1'}{a}\right)^2 + \left(\frac{x_2'}{b}\right)^2 = 1.$$

This form looks more like the formula for a circle, so it is likely more familiar to us. However, it does not allow for an orientation. To convert between the prime and normal coordinate plane, we solve an eigenvalue problem (this is the same idea as finding the principal axes in a moment of inertia tensor problem).

The general form of an ellipse is a "quadratic" form because it is quadratic in our coordinates (x_1, x_2), either singly or paired. We can write this as a matrix: $\vec{x}^T \mathsf{A} \vec{\gamma}$:

$$\begin{pmatrix} x_1 & x_2 \end{pmatrix} \begin{pmatrix} \alpha_1 & \beta \\ \beta & \alpha_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}.$$

To find the principal axes, we solve an eigenvalue problem. We want to solve $\mathsf{A}\vec{v} = \lambda\vec{v}$ for λ and B. Our matrix is symmetric, so we know that B will be orthagonal ($\mathsf{B}^{-1} = \mathsf{B}^T$). From the similarity transform, we know that

$$\mathsf{B}^{-1}\mathsf{A}\mathsf{B} = \mathsf{D} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}.$$

We can then rewrite our expression via multiplying by the identity matrix and then regrouping:

$$\gamma = \vec{x}^T \mathsf{A} \vec{x} = (\vec{x}(\mathsf{B})(\mathsf{B}^T)\mathsf{A}(\mathsf{B})(\mathsf{B}^T)\vec{x})$$

$$= \boxed{\vec{x}^T \mathsf{B}} \; \boxed{(\mathsf{B}^T \mathsf{A} \mathsf{B})} \; \boxed{(\mathsf{B}^T \vec{x})}$$

where $\vec{x}^T\mathsf{B}$ can be rewritten as $(\mathsf{B}^T\vec{x})^T = \vec{x}'^{\,T}$ (the transpose of a product is equal to the product of the transposes

in reverse order), $(\mathsf{B}^T\mathsf{A}\mathsf{B})$ is just D, and $(\mathsf{B}^T\vec{x})$ is a linear transformation to some new vector $\vec{x}'$. Thus,

$$\gamma = \vec{x}'^{\,T}\mathsf{D}\vec{x}' = \begin{pmatrix} x_1' & x_2' \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \lambda_1 x_1'^2 + \lambda_2 x_2'^2,$$

which we can easily convert to standard form.

$\vec{x}' = \mathsf{B}^T\vec{x}$ tells us about the rotation between coordinate systems. Since B^T is orthagonal, we can use it as a rotation matrix (this also requires that its determinant is 1). We learn from this that the semi-major and semi-minor axes are related to the eigenvalues.

The general form given above more generally represents a comic section; it is only an ellipsoid if $\gamma > 0$.

More examples of eigenvalue problems:

- Quantum mechanics: the time independent Schrödinger equation is an eigenvalue problem. $\hat{H}\psi = E\psi$ has the same form as $\mathsf{A}\vec{v} = \lambda\vec{v}$, where ψ is the eigenvalues and E is the eigenvectors.
- Classical mechanics: finding normal modes for an oscillating system. When building a bridge, for example, we do not want its natural frequency to be something that naturally occurs, so we want to know its natural frequency to ensure this; natural frequencies are just eigenvalues.

In summary, **an eigenvalue problem diagonalizes a matrix**, and a diagonal matrix is easier to work with than a general one. We an often apply this just as we did when simplifying the ellipse.

8.6.1 Back to solving $\mathsf{A}\vec{x} = \vec{b}$

Returning to our old example of having $\mathsf{A}\vec{x} = \vec{b}$ and wanting to solve for $\vec{x}$. Given λ and B, we can rewrite this as

$$\mathsf{B}(\mathsf{B}^{-1}\mathsf{A}\mathsf{B})\mathsf{B}^{-1}\vec{x} = \vec{b}$$

where the parenthetical quantity is the similarity transform, and yields D. Multiplying both sides by B^{-1} gives us

$$(\mathsf{B}^{-1}\mathsf{B})\mathsf{D}\mathsf{B}^{-1}\vec{x} = \mathsf{B}^{-1}\vec{b}$$

where $\mathsf{B}^{-1}\mathsf{B}$ is just the identity matrix, so

$$\mathsf{D}\mathsf{B}^{-1}\vec{x} = \mathsf{B}^{-1}\vec{b}.$$

We define $\mathsf{B}^{-1}\vec{x} \equiv \vec{x}'$ and $\mathsf{B}^{-1}\vec{b} \equiv \vec{b}'$; these are just linear transformations of the original vectors. This leaves us with

$$\mathsf{D}\vec{x}' = \vec{b}' = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_n \end{pmatrix} \begin{pmatrix} x_1' \\ \vdots \\ x_n' \end{pmatrix} = \begin{pmatrix} b_1' \\ \vdots \\ b_n' \end{pmatrix}$$

which is equivalent to $\lambda_1 x_1' = b_1'$, etc. From this, we can find that $x_1 = \frac{b_1'}{\lambda_1}$, or $\vec{x} = \mathsf{D}^{-1}\vec{b}'$. We don't generally like inverses, but they're fine for diagonal matrices:

$$\mathsf{D} = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix} \quad \rightarrow \quad \mathsf{D}^{-1} = \begin{pmatrix} \frac{1}{\lambda_1} & 0 & 0 \\ 0 & \frac{1}{\lambda_2} & 0 \\ 0 & 0 & \frac{1}{\lambda_3} \end{pmatrix}.$$

To find the full solution, recall that $\vec{x}' = \mathsf{D}^{-1}\vec{b}' = \mathsf{B}^{-1}\vec{x}$, so

$$\vec{x} = \mathsf{B}\vec{x}'.$$

That said, this is a lot of calculations and inverses, and we don't generally want to do that, but it is worth seeing that this formally works.

9 Optimization and Curve Fitting

9.1 Parameter estimation

We again return to our system of linear equations: $\mathsf{A}\vec{x} = \vec{b}$. We have been assuming A is square, but what if it is an mxn matrix, with m equations for n unknowns? If m¿n, we have too many equations. We know there is no unique solution, but we want to find the best or closest solution—how do we quantify "best"? We use the Euclidian distance, or "sum of squares."

Suppose we have measurements $y(x_1)$, $y(x_2)$, ... with uncertainties σ_i. We also have some model $y(x, \vec{a})$ which we can evaluate at any x but depends on some parameters $\vec{a} = (a_1 \quad a_2 \quad \dots)$. Our goal will often be to determine the best fit parameters. We have some simplifying assumptions to make this easier, the most significant of which is assuming Gaussian uncertainties. For Gaussian uncertainties, we know the probability of making a measurement and getting y_i with uncertainty σ_i from our model $y(x, \vec{a})$:

$$P_i \propto \exp\left(\frac{-1}{2}\left(\frac{y_i - y(x_i, \vec{a})}{\sigma_i}\right)^2\right).$$

For many *independent* measurements, the total probability is given by

$$P_{\text{tot}} = \prod_{i=1}^{N} P_i \propto \prod_{i=1}^{N} \exp\left(\frac{-1}{2}\left(\frac{y_i - y(x_i, \vec{a})}{\sigma_i}\right)^2\right).$$

This is unnormalized, so we call it the **likelihood $\mathcal{L}$**:

$$\mathcal{L} = \mathcal{L}_0 \exp\left(\frac{-1}{2}\boxed{\sum_{i=1}^{N}\left(\frac{y_i - y(x_i, \vec{a})}{\sigma_i}\right)^2}\right)$$

where the boxed term is χ^2, or "chi-squared." We notice it looks kind of like a distance—especially ignoring the σ term, it is just the sum of the squares of the differences between to things, which is exactly how we measure the distance between two points. The only difference is that it is weighted: measurements with higher uncertainty are given less weight than those with low uncertainty.

9.2 Quantifying best/closest

When we talk about the best or closest fit, we have found this by maximizing likelihood—we want to get a **maximum likelihood estimator** for our parameters. Notice that

$$\ln(\mathcal{L}) = \sum_{i=1}^{N}\left(\frac{y_i - y(x_i, \vec{a})}{\sigma_i}\right)^2 + \ln(\mathcal{L}_0).$$

Thus, maximizing likelihood is the same as maximizing $\ln(\mathcal{L})$, since the natural log is a monotonically increasing function. $\ln(\mathcal{L}_0)$ is constant, so as the parameters change, only the χ^2 term of the likelihood changes. Also, because of the $-\frac{1}{2}$ term, maximizing $\ln(\mathcal{L})$ is the same as minimizing $-\frac{1}{2}\sum_{i=1}^{N}\chi^2$. This is often treated as a starting point to minimize χ^2 and find our best fit parameters. We know from calculus how to minimize a quantity:

$$\frac{\partial \chi^2}{\partial a_k} = 0$$

$$\sum_{i=1}^{N} 2\left(\frac{y_i - y(x_i,\vec{a})}{\sigma_i}\right)\left(\frac{-1}{\sigma_i}\cdot\frac{\partial y}{\partial a_k}\right) = 0$$

$$\sum_{i=1}^{N}\left(\frac{y_1 - y(x_i,\vec{a})}{\sigma_i^2}\right)\left(\frac{\partial y(x_i,\vec{a})}{\partial a_k}\right) = 0.$$

This tells us the relationship between the parameters $\vec{a}$. We have one such equation for each parameter we have $(k = 1,\ldots,n)$. This gives us n equations and n unknowns. It is not necessarily a linear system, but it is still solvable.

9.3 Example: fitting to a line; finding fit uncertainty and goodness of fit

The references to linearity here mean that the model's parameters are linear (they do not appear in exponents, trigonometric functions, etc.). Our model in this example is

$$y(x,\vec{a}) = a_1 + a_2 x$$

so we can find $\frac{\partial y}{\partial a_1} = 1$ and $\frac{\partial y}{\partial a_2} = x$, and write down our system from there:

$$\sum_{i=1}^{N}\frac{y_i - a_1 - a_2 x_i}{\sigma_i^2} = 0$$

$$\sum_{i=1}^{N}\left(\frac{y_i - a_1 - a_2 x_i}{\sigma_i^2}\right)x_i = 0.$$

This is a linear system for a_1 and a_2. We introduce some notation to make this easier to write:

$$S = \sum_{i=1}^{N}\frac{1}{\sigma_i^2} \qquad S_x = \sum_{i=1}^{N}\frac{x_i}{\sigma_i^2} \qquad S_{xx} = \sum_{i=1}^{N}\frac{x_i^2}{\sigma_i^2} \qquad S_y = \sum_{i=1}^{N}\frac{y_i}{\sigma_i^2} \qquad S_{xy} = \sum_{i=1}^{N}\frac{x_i y_i}{\sigma_i^2}$$

All of these are known quantities. We can use these to write

$$a_1 S + a_2 S_x = Sy \tag{9.1}$$

$$a_1 S_x + a_2 S_{xx} = S_{xy}. \tag{9.2}$$

We can solve this system:

$$S(9.1) - S_x(9.2) = \boxed{a(SS_{xx} - S_x^2)} = S_{xy} - S_x S_y$$

where we name the boxed term Δ, and note that it only depends on x:

$$\Delta \equiv a(SS_{xx} - S_x^2).$$

We can use this to solve for our best fit for a_2:

$$a_2 = \frac{SS_{xx} - S_x S_y}{\Delta}$$

and do the same for a_1:

$$S_{xx}(9.1) - S_x(9.2) \to a_1(SS_{xx} - S_x^2) = a_1\Delta = S_{xx}S_y - S_x S_{xy}$$

$$a_1 = \frac{S_{xx}S_y - S_x S_{xy}}{\Delta}.$$

We have now the best fit parameters for a line using χ^2. However, we're not done—we did not account for uncertainty.

9.3.1 Finding fit uncertainty

The uncertainty in these measurements means we have some "wiggle room" where models are still close to the data, so we want to know how well we have estimated our parameters; we want to find σ_{a_1} and σ_{a_2}. Because we have assumed Gaussian uncertainty, we know how to find these: we know how to propagate error for Gaussian uncertainties (cue the groans):

$$\sigma_{a_1}^2 = \sum_{i=1}^{N} \left(\frac{\partial a_1}{\partial y_i}\right)^2 \sigma_i^2.$$

We can calculate this analytically, since a_1 only depends on y in S_y and S_{xy}, and linearly there:

$$\frac{\partial a_1}{\partial y_i} = \frac{1}{\Delta}\left(S_{xx}\left(\frac{1}{\sigma_i^2}\right) - S_x\left(\frac{x_i}{\sigma_i^2}\right)\right)$$

$$\frac{\partial a_2}{\partial y_i} = \frac{1}{\Delta}\left(S\left(\frac{x_i}{\sigma_i^2}\right) - S_x\left(\frac{1}{\sigma_i^2}\right)\right).$$

Plugging these into the error propagation formula gives us

$$\sigma_{a_1}^2 = \sum_{i=1}^{N}\left(\frac{S_{xx}^2 - 2S_x S_{xx}x_i + S_x^2 x_i^2}{\Delta^2 \sigma_i^2}\right)$$

which we can simplify if we recall how we defined S, S_x, and S_{xx}:

$$\sigma_{a_1} = \frac{1}{\Delta}\left(S_{xx}^2 S - S_x^2 S_{xx}\right) = \frac{S_{xx}}{\Delta}$$

and

$$\sigma_{a_2}^2 = \frac{S}{\Delta}.$$

Now we can find the best fit parameters and their uncertainties for a line, but we're still not done. This is the best fit, but that doesn't necessarily make it a good fit, so we want to determine how good our fit is.

9.3.2 Goodness of fit

To find goodness of fit, we will be using frequentist statistics. We have probably seen the terms "1-σ," "2-σ," etc. used when discussing data or fits in the past. These refer to Gaussian uncertainties. If $P(x)dx$ is the probability of measuring $x \in [x, x+dx]$, then

$$P(x)dx = \frac{1}{\sqrt{2\pi}\ \sigma} \exp\left(\frac{-1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

where μ is the mean and σ is the variance. We can now ask what the probability is of $\mu - x_0 \leq x \leq \mu + x_0$:

$$p(x_0) = \int_{-x_0+\mu}^{x_0+\mu} P(x)dx = \frac{1}{\sqrt{2\pi}\ \sigma} \int_{-x_0+\mu}^{x_0+\mu} \exp\left(\frac{-1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) dx.$$

We shift this integral by setting $x \equiv x - \mu$ so that we integrate from $-x_0$ to x_0. If we use $y = \frac{x-x_0}{\sqrt{2}\ \sigma}$ and the fact that we're integrating over a symmetric range, we can say

$$p(x_0) = \frac{2}{\sqrt{\pi}} \int_0^{\frac{x}{\sqrt{2}\ \sigma}} e^{-y^2} dy.$$

This is not an integral we can do analytically, so we define it to be something instead. This is the **error function:**

$$\text{erf}(t) \equiv \frac{2}{\sqrt{\pi}} \int_0^t e^{-y^2} dy$$

so then

$$p(x_0) = \text{erf}\left(\frac{x}{\sqrt{2}\ \sigma}\right).$$

This gives us the probability of finding some x within x_0 of the mean for a Gaussian distribution with variance σ.

When we say "1-σ," "2-σ," etc., we mean "within n-σ of the mean." The probability of being within 1-σ of μ is $\text{erf}\left(\frac{1}{\sqrt{2}}\right) \approx .68 = 68\%$. The probability of being within 2-σ of the mean is $\text{erf}\left(\frac{2}{\sqrt{2}}\right) \approx .95 = 95\%$, and within 3-$\sigma$ is $\text{erf}\left(\frac{2}{\sqrt{3}}\right) \approx .997 = 99.7\%$. 3-$\sigma$ is typically considered good enough, but the modern standard in physics is 5-σ because not all errors are Gaussian, not everything is statistical, etc..

We can also consider the probability of not being within x_0 of the mean. We know the total probability must be 1, so we can define

$$q(x_0) = 1 - p(x_0) = 1 - \text{erf}\left(\frac{x_0}{\sqrt{2}\ \sigma}\right)$$

which shows up often enough that we call it **erfc**, where the c indicates "complement." Thus,

$$q(x_0) = \text{erfc}\left(\frac{x_0}{\sqrt{2}\ \sigma}\right)$$

Now we know how to use the probability distribution to find the probability of being a certain distance from the mean.

Note: whenever dealing with statistics, we must be very careful to know what exact question we are asking, and be very careful that the statistics we are doing answer that specific question.

We want to determine if our best fit is a good fit. We had assumed Gaussian uncertainty, but our values were squared in our χ^2 analysis, and squared Gaussians have a different probability distribution: the χ^2 distribution. For $\chi^2 \in [0, \infty)$ is given by

$$P(\nu, \chi^2)dx^2 = \frac{1}{2^{\frac{\nu}{2}}\Gamma(\frac{\nu}{2})}(\chi^2)^{\frac{\nu}{2}-1}e^{-\frac{\chi^2}{2}}dx^2$$

where Γ indicates the Gamma function, which is a generalization of the factorial, and ν is the number of **degrees**

of freedom: $\nu = N - n$. For N data points (the same N as in our sum) we fit n free parameters. We can think of the degrees of freedom as "the number of data points minus how many we used to fit our model." Think about fitting a line—with 2 points, there is 1 line through them so fitting is simple, and $\nu = 0$. The more points we add, the morw we can move the line around to be closer to, or further from, some of the data points, since we can't be close to all of them.

Determining goodness of fit follows the same ideas as before. We begin by asking what the probability is that $\chi^2 < \chi^2_0$:

$$p(\chi^2_0) = \int_0^{\chi^2_0} P(\nu, \chi^2) d\chi^2 \equiv \frac{\gamma(\frac{\nu}{2}, \frac{\chi^2_0}{2})}{\Gamma(\frac{\nu}{2})}$$

where γ is the incomplete Gamma function; if $\chi^2_0 \to \infty$, γ would be equal to Γ. We are more interested in the ratio here than in the numerator or denominator individually, so we say

$$p(\chi^2_0) \equiv P\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right)$$

where P is the normalized incomplete Gamma function, which goes to 1 as $\chi^2_0 \to \infty$. (This function exists in `scipy.special`.)

Goodness of fit is defined as the probability of getting a value for χ^2 greater than the value we found. Once we have our best fit parameters, we can calculate χ^2 and see how far we are from the model. To interpret this, we assign a probability to it based on how probable it is to get a χ^2 bigger than this if our model is true. Instead of integrating from 0 to χ^2, we go from χ^2 to ∞:

$$Q\left(\frac{\nu}{2}, \frac{x^2_0}{2}\right) = 1 - P\left(\frac{\nu}{2}, \frac{\chi^2_0}{2}\right),$$

where Q is a stanrd function defined for us as the normalized complement of the incomplete Gamma function (this, too, is in `scipy.special`).

As a rule of thumb, we use the reduced χ^2 (also called χ^2_ν):

$$\chi^2_\nu = \frac{\chi^2}{\nu}.$$

A good fit has $\chi^2_\nu \approx 1$.

9.4 Summary and

If we have independent Gaussian uncertainties, which we usually assume we do, we know that χ^2 is a **"maximum likelihood estimator"** of a model $y(x, \vec{a})$:

$$\chi^2 = \sum_{i=0}^{N} \boxed{\left(\frac{y_i - y(x_i, \vec{a})}{\sigma_i}\right)^2}. \tag{9.3}$$

We expect $\chi^2 \approx \nu$. For measurements within 1-σ, the boxed portion of Eq 9.3 will be roughly 1, so if we sum over it, we should get roughly N. However, N is the number of data points. not the degrees of freedom ($\nu = N - n$). With n free parameters, we can write a system of n equations for the boxed portion of Eq 9.3 and set it equal to 0, since we can use n data points to uniquely solve for the parameters. This won't be the best fit, but it tells us we can use n pieces of information, so that number is subtracted from total data points to find degrees of freedom. The more degrees of freedom we have, the closer χ^2 should be to ν, and the closer χ^2_ν should be to 1.

If we have a nonlinear model, minimizing χ^2 can be hard; it requires an initial guess, unlike fitting to a line.

A general linear least squares fit requires a model of the form

$$y(x, \vec{a}) = a_1 g(x) + a_2 g(x) + \ldots$$

We do not care what the functions $g(x)$ are; we only need the model to be linear in the parameters $\vec{a}$.

9.4.1 Interpreting Reduced χ^2

χ^2_ν should be close to 1; a value near 0 indicates a bad fit. We do not expect data to match a model exactly, since there is uncertainty in our measurements. A bad fit typically means that $\chi^2 >> \nu$. A χ^2 value that is too small is also a bad fit, but much rarer—it typically only happens if σ is too big, since then each point's contribution is less than the ideal value of 1.

10 Fourier Transforms

10.1 Intro (using continuous Fourier transforms

Consider a continuous function $h(t)$. We define the Fourier transform theoretically (this is the definition used in quantum mechanics):

$$H(\omega) = \int_{-\infty}^{\infty} h(t) e^{-i\omega t} dt \qquad \text{where } \omega \text{ is angular frequency.}$$

We then define an inverse Fourier transform:

$$h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(\omega) e^{i\omega t} d\omega$$

where the $\frac{1}{2\pi}$ term is a normalization factor that just has to be somewhere—there are multiple conventions, and it is important to know which convention any given numerical implementation of the Fourier transform uses. Experimentally, we tend to prefer "regular" frequency f to angular frequency ω, so we can

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{-2\pi i f t} dt$$

and then the inverse Fourier transform is given by

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{2\pi i f t} df$$

where the 2π gets absorbed into the exponent in our conversion between frequencies. We introduce notation for $H(f)$ and $h(t)$ being related as Fourier pairs: $h(t) \Leftrightarrow H(f)$.

Examples of Fourier transforms include

- Convolution: we can smooth a function $h(t)$ using some other function $g(t)$ (such as a Gaussian). Think of this as sliding $g(t)$ along $h(t)$. We express this as

  $$h * g = \int_{-\infty}^{\infty} h(\tau) g(t - \tau) d\tau$$

  where we are sliding $g(t)$ along and counting up the contributions we get at each point from $h(t)$. We can show that $\boldsymbol{h * g \Leftrightarrow H(f)G(f)}$.
- We can also define the power in a signal. We call this a periodogram, power spectrum, power spectral density, etc.. These are similar combinations of things normalized in different ways and useful for different purposes. Typically we'll want a 1-sided PSD, which has only the positive frequencies. Here, PSD $\propto |H(f)|^2 + |H(-f)|^2$.

10.2 Discrete Fourier transform

We briefly cover some mathematical detail (which also appears in quantum mechanics). In a discrete Fourier transform, we are expanding some function in terms of another function or set of functions (here, $e^{2\pi ift}$, which acts as a basis). The Fourier transform gives us the coefficients/amplitudes of each of these modes (which are related to plane waves). This is useful because it is an orthonormal set of functions to expand other functions. In the continuous case they're subtly orthonormal:

$$\int_{-\infty}^{\infty} e^{-2\pi i(f-f')t} dt = \delta(f - f')$$

$$\int_{-\infty}^{\infty} e^{-2\pi i(t-t')f} df = \delta(t - t'),$$

where δ is the Dirac delta function. Discrete orthonormality, meanwhile, is clearer (or so I'm told):

$$\frac{1}{N}\sum_{n=0}^{N-1} e^{\frac{2\pi i(k-k')n}{N}} = \delta_{kk'} = \begin{cases} 0 & k \neq k' \\ 1 & k = k' \end{cases}$$

$$\frac{1}{N}\sum_{k=0}^{N-1} e^{\frac{2\pi i(n-n')k}{N}} = \delta_{nn'}$$

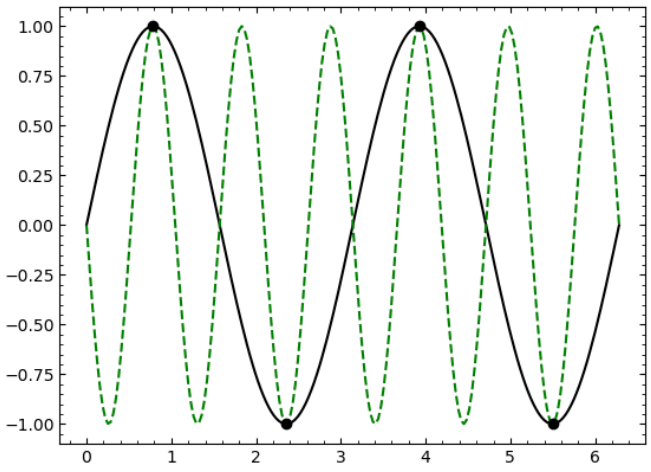
This orthonormality lets us represent a signal with finite number of samples. However, we are limited by the frequency range that our signal covers due to **aliasing**.

10.2.1 Aliasing and the Nyquist frequency

We discretely sample our signal out to some fixed interval Δ:

$$t_k = k\Delta \qquad \text{and} \qquad h(t_k) \equiv h_k = h(k\Delta)$$

Because this is a finite sample, we do not have all the information, so how do we choose Δ? With a large Δ, you can miss features in the signal, but with a small Δ, you end up with a ton of data points. Also, for any given Δ, multiple frequencies get mapped to a single frequency: **aliasing**.



Those two signals are equally well described by our data—how do we know which is right? Higher frequencies

get aliased (combined) with lower frequencies. We typically have "band-limited" samples, which we only care about within a finite frequency range (for example, there's no point in storing music at frequencies beyond those humans can hear). **We choose our sampling rate $\frac{1}{\Delta}$ to be at least twice the maximum frequency we are interested in**:

$$\Delta_{\min} = \frac{1}{2f_{\max}}.$$

Thus, there is a critical frequency to avoid aliasing, which we call the **Nyquist frequency**:

$$f_N = \frac{1}{2\Delta}.$$

So for our discrete Fourier transform, consider some signal sampled at a fixed interval Δ. We can represent this signal in the frequency domain at $f_n = \frac{n}{N\Delta}$ for $n = -\frac{N}{2}, \ldots, \frac{N}{2}$. Discretizing our Fourier transform integral gives

$$H_n \equiv H(f_n = \Delta \sum_{k=0}^{N-1} h_k e^{-2\pi i f_n t_k} = \Delta \sum_{k=0}^{N-1} h_k e^{\frac{-2\pi i n k}{N}}$$

and its inverse is

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{\frac{2\pi i k n}{N}}.$$

It may seem strange that these are sums from 0 to N-1, since we are supposed to be allowing negative frequencies, but it turns out that $-\frac{N}{2}$ to $\frac{N}{2}$ is the same as 0 to N-1. Also, evaluating at f_n for $|n| \leq \frac{N}{2}$ corresponds to N+1 samples, when we only have N. We had only said this because it is consistent with the Nyquist frequency, where $f_{\frac{N}{2}} = \frac{1}{2\Delta}$. To better understand the concept of negative frequencies and how they relate to positive ones, we examine both the $n = 1$ and $n = N - 1$ terms.

- $n = 1$: $H_1 \propto \exp(\frac{-2\pi i k}{N})$
- $n = N - 1$: $H_{N-1} \propto \exp(\frac{-2\pi i k (N-1)}{N}) = \exp(-2\pi i k) \exp(\frac{2\pi i k}{N}) = \exp(\frac{2\pi i k}{N})$

We see that H_1 has the same behavior as H_{N-1}. $\boldsymbol{f_{-1} = f_{N-1}}$, so it doesn't matter what we name them.

We need consider endpoints: we saw that H_{N-1} is related to H_1, not H_0. Why are N frequencies relevant, not N+1? Notice that $f_{\frac{N}{2}} = f_{-\frac{N}{2}}$ or $f_0 = f_n$. **If we included both of those, we'd be double counting, so we exclude one of the two.**

10.3 Fast Fourier transform

When we talk about a Fourier transform, we almost always mean the fast Fourier transform (FFT). The simple way to implement a discrete Fourier transform (with sums over things, etc.) typically takes time $\mathcal{O}(N^2)$. The modern version started in the 1960s by J.W. Cooley and John Tukey, who developed the FFT algorithm, has $\mathcal{O}(N \log_2(N))$, which increases roughly linearly with N.

This algorithm revolutionized the world by enabling wireless communication. The FFT lets us convert digitized signals into voice, music, video, etc. because handling noisy signals happens in the frequency domain. The FFT can turn 2 week computations into 30 second ones, so anything we can do even partially with a Fourier transform is done that way, because the FFT is so fast.

There are lots of implementations; for a while, a very strong (and open-source) algorithm was FFTW, which stands for Fastest Fourier Transform in the West). It can be used in Python, but, there's somewhat complicated pre-computation to do to take full advantage of its speed.

Note again that it is important to be aware of which conventions are used in any implementation of the FFT we use.

As a fun fact of sorts: arbitrary precision arithmetic can be done at least in part with the FFT.

11 Random Numbers

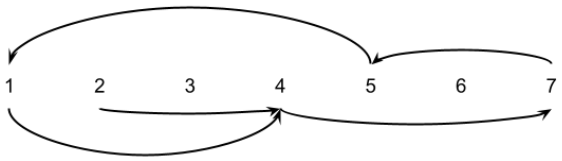
11.1 How do computers generate random numbers?

Computers are deterministic: given an algorithm and an initial state, they must prpduce the same output every time. How can we produce random numbers using a deterministic device? We more or less can't, so we settle for pseudorandom numbers. For scientific applications, this is actually preferable—we want reproducibility.

By a random sequence, we mean something that satisfies some statistical properties; roughly, that there are no repeated patterns.

Psuedorandom numbers are generated by an algorithm giving the computer a "long" sequence of numbers which it steps between "independently." We begin with a **seed** which tells us where to begin, and move along the sequence from there. Starting from the same seed means stepping along the same sequence.

The computer does not necessarily step between all the numbers in a sequence: take a short sequence of seven numbers. One possible path is



It is easy to make mistakes when coding a random number generator: some famous bad ones are `randU` and the one Notch wrote for Minecraft. A more general bad algorithm for psuedorandom number generation is a **multiplicative congruential algorithm**. The idea of this generator is to generate iteratively, where eacn new random number I_{j+1} is given by

$$I_{j+1} = aI_j \% m \qquad \text{where } \% \text{ is the modulo operator, and } a \neq 0.$$

The maximum length of such a sequence is given by m. A good minimal choice for this algorithm is $a = 7^6$ and $m = 2^{31} - 1 \approx 2 \times 10^9$. (Note that m is relatively small for the maximum length of our sequence.) We still need a seed, and want to automate choosing it. A common choice used to be the current time, which worked well until people started running things in parallel on computers with synchronized clocks. We want a truly random seed to generate sufficiently random psueudorandom numbers.

We get these random seeds from physics—decay, noise, etc. are all random processes. This was first done with the introduction of `\dev\random`, which was added to the Linux kernel in 1994 by Linus Torvalds, and is used everywhere by now. There are all sorts of random things computers can access (packet delivery online, user keystroke spacing, disk accessing, etc.). It can store information from these sources of entropy, and use them to create a pool of random numbers. Numpy and other modern random number generators automatically draw from this pool. This pool can get depleted quickly, which is why we typically use the pool to seed psuedorandom number

generators instead of drawing our set of random numbers directly from the pool.

11.2 Changing the distribution of random numbers

Random number generators typically generate random numbers uniformly between 0 and 1. If we want a non-uniform distribution, we must transform to it. For a uniform distribution between 0 and 1, the probability of getting a number in that range is

$$\int_0^1 p_u(u)du = 1.$$

We can generalize this to find that the probability of getting $u' \in [0, u]$ is

$$\int_0^u p_u(u')du' = u.$$

Suppose we want our random numbers to be distributed such that $p(y)dy \propto ydy$ for $y \in [2, 4)$. We first need to normalize:

$$\int_2^4 p(y)dy = 1$$

$$= \int_2^4 Nydy = \frac{N}{2}y^2\Big|_2^4$$

$$= \frac{N}{2}(16 - 4) = 6N \rightarrow p(y)dy = \frac{1}{6}ydy$$

So we use the uniform distribution and then map each random number to the equivalent one from another distribution. The probability of getting some value is equivalent to the area under the curve at that value, so we map the area to get u from $P(u)$ to an equal area for getting y from $p(y)$:

$$\int_0^u P_u(u')du' = u = \int_2^y p(y')dy' = \int_2^y \frac{1}{6}y'dy' = \frac{1}{2}y'^2\Big|_2^4 = u = \frac{1}{2}(y^2 - 4) \rightarrow \boxed{y = \sqrt{12u + 4}}.$$

Picking uniformly distributed random numbers and plugging them into the map will give us the distribution we want.

We do one more example of transformation: in this case, our desired distribution is an inverted parabola:

$$p(y)dy = N(1 - y^2)dy. \tag{11.1}$$

We begin by normalizing:

$$1 = \int_{-1}^1 p(y)dy = \int_{-1}^1 1 - y^2 dy$$

$$= N\left(2 - \frac{1}{3}(2)\right) = \frac{4}{3}N \rightarrow N = \frac{3}{4}$$

so that

$$p(y)dy = \frac{3}{4}(1 - y^2)dy.$$

The next step is mapping:

$$U = \int_{-1}^{y} p(y')dy' = \frac{3}{4}\int_{-1}^{y} 1 - y'^2 dy' = \frac{3}{4}\left(y' - \frac{1}{3}y'^3\right)\Big|_{-1}^{y}$$

$$= \frac{3}{4}\left(\left(y - \frac{1}{3}y^3\right) - \left(-1 + \frac{1}{3}\right)\right) = \frac{3}{4}\left(y - \frac{1}{3}y^3 + \frac{2}{3}\right)$$

$$\to \frac{1}{3}y^3 - y - \frac{2}{3} + \frac{4}{3}u = 0.$$

Finally, we invert this—we want y as a function of u. We can do this the hard way, or with a spline.

An alternative (and simpler) method for generating random numbers from some distribution is the rejection method, which says we know how to pick uniformly distributed numbers on the interval $[a, b]$. We consider this as a 2D problem: draw a box around our function, and uniformly sample inside that box. We know which values are under our curve, so we can just reject all samples not under the curve. A problem with this method is that we are throwing away some samples, which is a waste of effort, so we prefer the transformation method, when we can use it. That said, the rejection method is not bad for complicated regions, especially for higher dimensions. It also tells us something about the area under the curve—the area of the box is easy to find, and we know both the total number of points and how many fall under the curve. This is telling us something about integrals, which brings us to **Monte Carlo integration**.

11.3 Monte Carlo Integration

We have some integral

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

where we have picked random values for x that tell us where we want to sample our function. We can use this to estimate the integral. Doing this at fixed intervals is functionally a Reimann sum. We pick uniformly distributed numbers u, which we can map to $[a, b)$ with $x = (b - a)u + a$, and then we can approximate I with

$$I \approx \frac{b-a}{N}\sum_{j=1}^{N} f(x_j).$$

However, the error here scales as $\frac{1}{\sqrt{N}}$, so Monte Carlo integration is good for getting a rough estimate of the integral but not great for accuracy. However, it is useful for higher dimensions or complicated shapes—if we can specify a high-dimensional volume in some way, we can know if we're inside or outside of the shape.

A Markov chain is an algorithm with no memory—future states do not depend on past states, only the current one. A classic example is a random walk, where one decides which direction to walk based on, say, a coin flip. Each step is independent of the previous one.

We are interested in Markov Chain Monte Carlo (MCMC) algorithms. We just discussed Markov chains, and here Monte Carlo means there is randomness involved. MCMC algorithms are like a random walk but more specific: we want a probability distribution based on a relationship between all the parameters, we might want a likelihood function, etc. This can be hard to evaluate in high dimensions, e.g. 5. Imagine discretizing the space in 10 steps in either direction—this would be 10^5 calculations, 10 steps would be 10^{10} calculations, etc. MCMC algorithms are useful for situations like these. A good example of such an algorithm is the Metropolis-Hastings algorithm.