Numerical & Computational Methods

6CCM359A, Fall 2020[1]

Salman Ahmad Faris

King's College London

E-mail: salman.faris@kcl.ac.uk

[1] *based on lectures by Benjamin Doyon. These notes, however, have been altered strongly after the lectures. In particular, some proofs were filled in by me; and some of the contents were adjusted so that organization is prioritized and things are coherent. All errors are surely mine, feel free to email me if you spot any. I would also like to thank Anas for our discussion on the Monte-Carlo method which was very useful for this notes.*

Contents

Math convention. In this course, we will always assume that any function f is $C^\omega(\Omega)$ where Ω is an open subset of $\mathbf{R}$ unless otherwise stated. For this reason, we are free to Taylor expand functions. Furthermore, we denote $\mathcal{R}[a,b]$ to be the space of Riemann-integrable functions over the compact interval $[a,b]$. We will thus always assume that $f \in \mathcal{R}[a,b]$ whenever we talk about integration.

Prerequisites. We assume a good command of real analysis (including some basic probability theory) and linear algebra. Knowledge of complex analysis and topology is a bonus but not necessary. However, we may use elements of topology in proofs because we believe that it make things more elegant and simple. We will also make use of the Stirling's approximation.

Theorem (*Stirling's approximation*)**.** Let $n \in \mathbf{N}$. Then

$$\ln n! = n \ln n - n + \mathcal{O}(\ln n).$$

By identifying the constant in the $\mathcal{O}(\ln n)$ term, we further have, for sufficiently large n the approximation

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.$$

Code convention. We will use Python 3. We will assume that all code has the following header code:

```
# System imports
from sys import exit
from typing import Callable, List, Tuple, TypeVar

# Third-party imports
import numpy as np
import matplotlib.pyplot as plt

# Custom type hints
Function = Callable
Real = TypeVar('Real', int, float)
Indeterminate = float
Matrix = np.array
Vector = np.array
mat_index = int
```

1 Let's go root hunting

The problem is simple: given a function $f(x)$, what are the values x_0 such that $f(x_0) = 0$? We will limit our discussion to real-valued functions with real variables. If the function is linear, it is boring. So, we are most interested in non-linear functions.

1.1 Newton method

The idea of the method is **linearizing $f(x)$ around a certain point x_0** of our choice and then solving the resulting linear equation. Say we get x_1 as a solution, then we repeat the linearizing step using x_1; so on and so forth. The caveat of this method is that we must know the derivative of f at the point we are linearizing.

Linearizing around a point x_n just means Taylor expanding the function f around x_n and neglecting $\mathcal{O}(h^2)$ terms. The latter condition is OK if we assume that the next point x_{n+1} we are searching for is near enough to x_n, so we will assume just that. Taking $n = 0$, linearizing around x_0 looks formally like

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \mathcal{O}(h^2) \approx f(x_0) + f'(x_0)(x - x_0).$$

The (approximate) solution to this linear equation is thus

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

We then repeat the process by linearizing around x_1 i.e. we have

$$f(x) \approx f(x_1) + f'(x_1)(x - x_1) \implies x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \text{ is a solution.}$$

It is then easy to see that iteratively, we would have the following general equation

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

which is known as the Newton's formula.

Theorem (*Newton method*)**.** Let x^* be a solution to $f(x) = 0$. If x_n is an approximation of x^* and $f'(x_n) \neq 0$, the next approximation is given by

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

with initial condition, a *good point* x_0.

Let us elaborate on what does it mean for the starting point x_0 to be a *good point*. Just looking at the Newton's formula, we know we want to avoid extremums when we pick x_0 as this may result in $f'(x_n) = 0$ for some n as we go through the iteration. Furthermore, we would want to choose x_0 to be in a neighbourhood (topologically and literally) of our hoped solution x^*. This is because... we want the sequence to not diverge. This is what we meant by x_0 being a good point. Loosely speaking, it is a point for which the method does not break.

Here's some observation regarding the sequence returned by the Newton method. Notice that if for some $n \in \mathbf{N}$, we have exactly $f(x_n) = 0$, then $x_{n+1} = x_n$ and so $x_i = x_n$

for all $i \geqslant n$. Such a point x_n is called a **stable point** of the recursion because it *stabilizes* the recursive process. Being a stable point is a good thing, but we also need it to be an **attractive point**. That is, as the recursion commences, the sequence tends to the point. Unfortunately, this **cannot be guaranteed**! Thankfully, in many cases, the Newton method works really well. That is, stable points are usually attractive points. Here is a recursive implementation of the Newton method.

Algorithm (*Newton method recursive*)**.** Recursive Python implementation:

```
def recursive_newton(f: Function, df: Function, x0: Real, n: int = 10):
    return x0 if n <=0 else recursive_newton(
                                f, df, x0 - f(x0)/df(x0), n-1)
```

We also have the following iterative implementation of the Newton method.

Algorithm (*Newton method iterative*)**.** Iterative Python implementation:

```
def iterative_newton(f: Function, df: Function, x0: Real, n: int = 10):
    xs = [x0]  # Sequence of xn.

    # Get latest x value in sequence and
    # apply the Newton recurrence formula.
    for _ in range(n):
        last = xs[-1]
        res = last - f(last)/df(last)
        xs.append(res)

    return xs[-1]
```

We can further abstract out the code in the `for` loop into a function `newton_formula()` that does only what the Newton's formula does. We will not do it here so that we can keep everything in a single function. Note that both algorithms have $\mathcal{O}(N)$ space complexity where N is the number of iterations or depth of the recursion. The time complexity for the iterative implementation is also $\mathcal{O}(N)$. However, the time complexity of the recursive implementation is a bit tricky and so we leave it as a fun exercise to the reader.

1.2 Secant method

The Newton method was very efficient but we needed to know the derivative of the function. Sometimes the derivative is simply not available. For example, when the function is too complicated.

The idea of the secant method is similar to the Newton method, which is also to make a linear approximation of $f(x)$. But this time, instead of using the tangent line, we we will be using the secant line. So, we start by choosing 2 points, say, x_0 and x_1 and approximate $f(x)$ as the secant line that crosses these two points. The line $L(x)$ that approximates $f(x)$ is thus

$$L(x) = \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1).$$

The solution to $L(x) = 0$ is then

$$x_2 = \frac{x_1 f(x_0) - x_0 f(x_1)}{f(x_0) - f(x_1)}.$$

We then repeat the process by choosing x_1 and x_2 and approximate f using the secant line through these points, so on and so forth. This gives the recurrence relation

$$\begin{aligned} x_n = \frac{x_{n-2} f(x_{n-1}) - x_{n-1} f(x_{n-2})}{f(x_{n-1}) - f(x_{n-2})} &= x_{n-1} - f(x_{n-1}) \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} \\ &= x_{n-1} - \frac{f(x_n)}{\Delta f(x_{n-1})/\Delta x_{n-1}} \end{aligned}$$

where $\Delta x_{n-1} = x_{n-1} - x_{n-2}$ and $\Delta f(x_{n-1}) = f(x_{n-1}) - f(x_{n-2})$.

Theorem (*Secant method*)**.** Let x^* be a solution to $f(x) = 0$. If x_n is an approximation of x^*, the next approximation is given by

$$x_{n+1} = x_n - \frac{f(x_n)}{\Delta f(x_n)/\Delta x_n},$$

with initial conditions x_0, x_1 and $\Delta x_n = x_n - x_{n-1}$ and $\Delta f(x_n) = f(x_n) - f(x_{n-1})$.

Now compare the secant recurrence formula with the Newton's formula. In the limit, the quotient $\Delta f(x_n)/\Delta x_n$ is precisely the derivative $f'(x_n)$. This explains why we went through the hassle to put the formula into that form.

Algorithm (*Secant method*)**.** Python implementation:

```
def secant(f: Function, x0: Real, x1: Real):
    xs = [x0, x1]. # Sequence of xn.
    tol = 10**(-7)
    while True:
        res = secant_recurrence_formula(f, xs[-1], xs[-2])
        err = abs(res-xs[-1])
        if err < tol:
            xs.append(res)
            break
        xs.append(res)
    return xs[-1]

def secant_recurrence_formula(f: Function, x0: Real, x1: Real):
    """The secant recurrence formula."""
    dx = float(x1 - x0)
    df = float(f(x1) - f(x0))
    return x1 - f(x1) * dx/df
```

1.3 Bisection method

There are cases where both the Newton and secant method don't work. For example when the function is not smooth: think piecewise functions consisting of linear pieces. In this case, the choice of the initial point would play a huge role in either finding a solution or

never finding one and getting stuck[1] in a loop. Fortunately, there is the bisection method which is guaranteed to converge to a solution whenever a certain condition is met. This method relies on the intermediate value theorem.

[1] *if we choose to recurse until the error is small*

Theorem (*IVT*)**.** Suppose $f : [a, b] \to \mathbf{R}$ be continuous. Then for all d strictly between $f(a)$ and $f(b)$, there exists $c \in (a, b)$ such that $f(c) = d$.

Proof. Let $\mathbf{R}$ be given the usual topology. Then any connected topological subspace of $\mathbf{R}$ is an interval. Let $I = [a, b]$. Since $f : I \to \mathbf{R}$ is continuous and I is connected, then $f(I)$ is connected and hence, an interval. Since d is strictly between $f(a)$ and $f(b)$, then $d \in f(I)$ by definition of an interval. So there exists $c \in I$ such that $f(c) = d$. But $d \neq f(a)$ and $d \neq f(b)$, so we must have $c \in (a, b)$. ∎

In particular, the bisection method relies on the following corollary.

Corollary (*Bisection guarantor*)**.** Suppose $f : [a, b] \to \mathbf{R}$ is continuous such that $f(a)f(b) < 0$. Then $f(x)$ changes[2] sign on $[a, b]$, and there exists $c \in (a, b)$ such that $f(c) = 0$.

[2] *at least once*

Proof. Since $f(a)f(b) < 0$, we have that 0 lies strictly between $f(a)$ and $f(b)$. By the IVT, there exists $c \in (a, b)$ such that $f(c) = 0$; and $f(x)$ changes sign precisely at c. ∎

That is, we meant $f(c+\epsilon)f(c-\epsilon) < 0$ for some $\varepsilon > 0$ sufficiently small.

This is the condition that we mentioned earlier — the condition that must be met first before the magic of the method happens. The bisection guarantor guarantees a root on $[a, b]$ *only if* the function is continuous on $[a, b]$ and that $f(a)f(b) < 0$.

Slogan. $f(a)f(b) < 0$ must be true for bisection method to work.

However, do not confuse this with the guarantee of convergence. *If this condition is met, convergence is assured for free.* This implication is why we called the corollary the *bisection guarantor*. The idea of the bisection method is the same as in the proof of the Bolzano-Weierstrass theorem. It is basically a binary search style algorithm. Suppose we start with two points a, b. We now look for the root of $f(x) = 0$ in the interval $[a, b]$. Consider the midpoint $c = (a + b)/2$. We now have three cases.

1. If $f(c) = 0$, we are done.
2. If $f(a)f(c) < 0$, repeat the process on the interval $[a, c]$.
3. If $f(b)f(c) < 0$, repeat the process on the interval $[c, b]$.

Below is the Python implementation of the method.

Algorithm (*Bisection method*)**.** Python implementation:

```
def bisection(f: Function, a: Real, b: Real):
    tol = 1/10 ** 10
    while True:
        c = float(a+b) / 2  # Compute midpoint.
        if abs(f(c)) < tol:
            return c
```

```
        elif f(a)*f(c) < 0:
            b = c
        elif f(b)*f(c) < 0:
            a = c
        else:
            return  # Bad exit.
```

Notice that we put a bad exit in the code. This is to prevent infinite loops when someone tries to run the code on a function which does not have a real root to begin with. For example, if someone would to try to run this code on the function $x^2 + 1$. This is also to prevent the bisection method from being applied on intervals where the bisection guarantor condition is not met.

1.4 How fast can a root-hunting method converge?

Suppose we have a sequence (x_n) obtained from one of the root-hunting numerical method which converges to x^*, the supposed root of the function we are computing, say, $f(x)$. Now, let's ask a bit more:

How fast do we approach it?

The natural way to measure this is to compute the ratio of succesive differences between the true root x^* and the step value x_n,

$$\frac{|x_{n+1} - x^*|}{|x_n - x^*|}.$$

Of course if x_{n+1} is nearer to x^* compared to x_n, this ratio should be relatively small. In fact, for the Newton method this ratio tends to 0. But in order to have an indication of how **fast** x_n approaches x^*, what we can do is to manipulate this ratio cleverly so that it approaches a constant. This can be done by making the denominator smaller. Since for n large, $|x_n - x^*|$ is small, the power of this number gives an even smaller value. So fixing some power $q \geqslant 1$, the following expression should be able to indicate the rate of convergence:

$$\lim_{n\to\infty} \frac{|x_{n+1} - x^*|}{|x_n - x^*|^q}.$$

But here's the tricky part:

- If q is too large, then the ratio may get too big and hence the limit may not exist.
- If q is too small, then the ratio may get too small and hence the limit is 0.

These are cases we don't want. So we want to choose a q so that this limit is some nonzero finite $\mu \in \mathbf{R}$. If this is possible, then we say the sequence (x_n) is q-convergent.

Definition (*q-convergent*)**.** Let $(x_n) \subseteq \mathbf{R}$ be a sequence which converges to x^*. We say (x_n) is **q-convergent** with $q \geqslant 1$ if

$$\lim_{n\to\infty} \frac{|x_{n+1} - x^*|}{|x_n - x^*|^q} = \mu,$$

for some $\mu \neq 0$. In particular, when $q = 2$, we say (x_n) is **quadratically convergent**[3]. The number μ is then said to be the **rate of convergence**.

[3] *or converges quadratically*

The higher the q the better. To see this, consider the following: Let (ε_n) be a sequence defined by $x_n - x^*$ where (x_n) is a sequence converging to x^*. In other words, (ε_n) is the error of (x_n) as it converges. Say, after 23 iterations, we have that $\varepsilon_{23} \approx 10^{-7}$. Let's look at what will happen on the 24-th iteration.

If (x_n) is 1-convergent, then we have $\varepsilon_{24} \approx \mu_1^{-1} 10^{-7} \sim 10^{-7}$ where μ_1 is the rate of convergence. That is, we reduce the error only by a factor of μ_1^{-1} which is quite slow. So n iterations into the method, we have $\varepsilon_n \sim \mu_1^{-n}$.

However, if (x_n) is 2-convergent, then we have $\varepsilon_{24} \approx \mu_2^{-1}(10^{-7})^2 = \mu_2^{-1} 10^{-14} \sim 10^{-14}$. Wow, we just doubled the number of exact digits in just one iteration. If (x_n) is 3-convergent, then we would triple the number of exact digits in a single iteration. The pattern should then be obvious as q increases.

Slogan. q-convergent $\implies$ The number of exact digits gained q-ple on each iteration.

Let's look at some examples.

Example. Consider the simple sequence $x_n = 3^{-2^n}$. Clearly, $x_n \to 0$ as $n \to \infty$. Now observe that

$$\frac{|x_{n+1}|}{|x_n|^q} = \frac{3^{-2^{n+1}}}{3^{-q2^n}} = 3^{q2^n - 2^{n+1}} = 3^{2^n(q-2)}.$$

So, this ratio converges iff $q \leqslant 2$. So x_n is 2-convergent.

There's a more general construction.

Example. Let $a, b \in \mathbf{R}$ be nonzero, $c > 0$ and $\lambda > 1$ and consider the sequence

$$x_n = ab^{-c\lambda^n},$$

which tends to 0 as $n \to \infty$. We claim that x_n is λ-convergent.

Proof. Observe that in the ratio, the a immediately cancels out and we have

$$\frac{|x_{n+1}|}{|x_n|^q} = \frac{b^{-c\lambda^{n+1}}}{b^{-qc\lambda^n}} = b^{\lambda^n c(q-\lambda)}.$$

Since $\lambda > 1$ and $c > 0$, this ratio converges iff $q \leqslant \lambda$. This implies that x_n is λ-convergent. ■

1.4.1 q-convergence of methods

Definition. We say that an iterative method is q-convergent if the sequence (x_n) produced by the method is q-convergent.

Using this definition, we can now say the following.

Theorem. The Newton method either converges quadratically or linearly.

The idea of the proof is simple: compare two consecutive steps of the Newton method for a step n sufficiently large.

Proof. Suppose x^* is a root of $f(x)$. Then Taylor expanding f around x^* we have

$$f(x) = 0 + f'(x^*)(x - x^*) + \frac{1}{2}f''(x^*)(x - x^*)^2 + \mathcal{O}(x^3).$$

Assume we choose a good starting point and suppose we are at the n-th step of the Newton method for a sufficiently large n. Since we assumed that n is large enough, x_{n-1} is near to x^*, and so we can Taylor expand $f(x_{n-1})$ and its derivative and disregard $\mathcal{O}(h^3)$ terms. We have

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} = x_{n-1} - \frac{f'(x^*)\varepsilon_{n-1} + \frac{1}{2}f''(x^*)\varepsilon_{n-1}^2}{f'(x^*) + f''(x^*)\varepsilon_{n-1}},$$

where $\varepsilon_{n-1} = x_{n-1} - x^*$. We now have two cases.

Case 1: $f'(x^*) = 0$. The equation reduces to

$$x_n = x_{n-1} - \frac{\frac{1}{2}f''(x^*)\varepsilon_{n-1}^2}{f''(x^*)\varepsilon_{n-1}} = x_{n-1} - \frac{\varepsilon_{n-1}}{2}.$$

Subtracting by x^* on both sides, we then have

$$\varepsilon_n = \frac{\varepsilon_{n-1}}{2} \implies \lim_{n\to\infty} \frac{|\varepsilon_n|}{|\varepsilon_{n-1}|} = \frac{1}{2}.$$

Thus, in this case the method is **linearly convergent** with a rate of $1/2$.

This implies that $\varepsilon_n \sim 2^{-n}$ i.e. the error decrease by a factor of 2^{-1} on each iteration which is very slow. From a more practical point of view, we require (at least) 4 iterations before getting another exact digit of the estimation.

Case 2: $f'(x^*) \neq 0$. Subtracting by x^* on both sides and doing some algebra we have that

$$\varepsilon_n = \varepsilon_{n-1} - \varepsilon_{n-1}\left(1 - \frac{\varepsilon_{n-1}}{2}\frac{f''(x^*)}{f'(x^*)}\right) = \varepsilon_{n-1}^2 \frac{f''(x^*)}{2f'(x^*)}.$$

This implies that

$$\lim_{n\to\infty} \frac{|\varepsilon_n|}{|\varepsilon_{n-1}|^2} = \frac{1}{2}\left|\frac{f''(x^*)}{f'(x^*)}\right| \in \mathbf{R}.$$

In other words, the method is **quadratically convergent**. ■

Next, we will see that the secant method converges somewhere between linear and quadratic. Some people call this *superlinear*. So the method is slower than the Newton method but not too slow.

Theorem. The secant method is φ-convergent, where $\varphi = (1 + \sqrt{5})/2$ which is the golden ratio.

Proof. Let x^* be a root of $f(x)$ and write $\varepsilon_n = x_n - x^*$. Now observe that

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

But $x_n - x_{n-1} = \varepsilon_n - \varepsilon_{n-1}$ simply by adding $0 = x^* - x^*$. Subtracting by x^* on both sides and doing some algebra, we then have

$$\varepsilon_{n+1} = \varepsilon_n - \frac{f(x_n)(\varepsilon_n - \varepsilon_{n-1})}{f(x_n) - f(x_{n-1})} = \frac{\varepsilon_{n-1}f(x_n) - \varepsilon_n f(x_{n-1})}{f(x_n) - f(x_{n-1})}, \qquad (\dagger).$$

The Taylor expansion of $f(x_n)$ around x^* neglecting $\mathcal{O}(x_n^3)$ terms is given by

$$f(x_n) \approx \varepsilon_n f'(x^*) + \frac{\varepsilon_n^2}{2} f''(x^*).$$

we have used $f(x^*) = 0$ here.

We now look at the numerator and denominator of (†) separately. After Taylor expansion, we find that the numerator is

$$\frac{\varepsilon_{n-1}\varepsilon_n^2 f''(x^*)}{2} - \varepsilon_n \varepsilon_{n-1}^2 f''(x^*) = \frac{\varepsilon_n \varepsilon_{n-1} f''(x^*)}{2}(\varepsilon_n - \varepsilon_{n-1}).$$

After Taylor expansion, we find that the denominator is

$$f'(x^*)(\varepsilon_n - \varepsilon_{n-1}) + \frac{f''(x^*)}{2}(\varepsilon_n^2 - \varepsilon_{n-1}^2).$$

So the quotient (†) simplifies to

$$\varepsilon_{n+1} = \varepsilon_n \varepsilon_{n-1} \frac{\frac{f''(x^*)}{2}}{f'(x^*) + \frac{f''(x^*)}{2}(\varepsilon_n - \varepsilon_{n-1})} = \varepsilon_n \varepsilon_{n-1} \frac{L}{1 + L(\varepsilon_n + \varepsilon_{n-1})},$$

where

$$L = \frac{f''(x^*)}{2f'(x^*)}.$$

This implies that

$$\varepsilon_{n+1} \approx \varepsilon_n \varepsilon_{n-1} L.$$

Now suppose x_n is q-convergent. We claim that $q = (1+\sqrt{5})/2$. To see this, we go to definitions. By definition, we have

$$\frac{|\varepsilon_{n+1}|}{|\varepsilon_n|^q} \to \mu, \quad \text{and} \quad \frac{|\varepsilon_n|}{|\varepsilon_{n-1}|^q} \to \mu,$$

for some $\mu > 0$. So we have

$$\mu |\varepsilon_n|^q \approx |\varepsilon_{n+1}| \approx |\varepsilon_n||\varepsilon_{n-1}||L|,$$

which implies that

$$|\varepsilon_n|^{q-1} \approx \frac{|L|}{\mu}|\varepsilon_{n-1}| \implies |\varepsilon_n| \approx \left(\frac{|L|}{\mu}\right)^{1/(q-1)} |\varepsilon_{n-1}|^{1/(q-1)}.$$

By hypothesis, we thus have two equalities

$$q = \frac{1}{q-1}, \quad \text{and} \quad \mu = \left(\frac{|L|}{\mu}\right)^{1/(q-1)}.$$

The first equality gives that

$$q^2 - q - 1 = 0 \implies q = \frac{1 \pm \sqrt{5}}{2}.$$

But $q \geqslant 1$, so indeed $q = (1+\sqrt{5})/2 = \varphi$. Moreover, the second equality gives that

$$\mu^q = |L| \implies \mu = |L|^{1/q},$$

and so the rate of convergence of the method is

$$\mu = \left| \frac{f''(x^*)}{2f'(x^*)} \right|^{1/\varphi},$$

and we are done. ■

The slow speed is worth it given that we don't have to know the derivative of the function.

Theorem. The bisection method converges linearly with a rate of 1/2.

Proof. This is obvious from construction. ■

2 Approximating functions

To make things concrete fast, let's start with a problem statement. Given the function $f(x) = \sin(x)$, how would we evaluate this function numerically? You can't just write `f = lambda x: sin(x)` because then what does `sin` even means? This is the goal of this section: to allow writing functions such as $\sin(x)$ in code and be able to evaluate it numerically[4].

[4] *hence, approximately.*

2.1 Lagrange interpolation

One way to approximate a function $f(x)$ is to interpolate a finite sequence of known points $\{p_i\}$ where $p_i = (x_i, f(x_i))$. These may be points known, say, analytically, experimentally or numerically.

analytically: we know $\sin(\pi) = 0$

experimentally: we obtained data from experimental evidence

It is worth mentioning at this early stage that interpolating is **not** the same as forcing/fitting a curve onto the points — something that we will look into later. The difference is that the interpolated data goes through *all* the points of the original data. On the other hand, fitting a curve uses the idea of least error and tries to best estimate the points. It need not necessarily go through *any* of the points.

The most obvious way to interpolate data is using a polynomial. Given 2 data points, we can do an interpolation with a degree 1 polynomial. With 3 data points, we have one more degree of freedom and can do an interpolation with a degree 2 polynomial, so on and so forth. In fact such a polynomial is unique (as we will see below) and they are given by the so-called *Lagrange polynomials*.

Definition (*Lagrange polynomial*)**.** The **Lagrange (interpolating) polynomial** is a polynomial $P_n(x)$ with degree n that passes through $n+1$ points $(x_0, y_0), \ldots, (x_n, y_n) \in \mathbf{R}^2$ where $x_i \neq x_j$ whenever $i \neq j$ defined by

$$P_n(x) = \sum_{i=0}^{n} y_i \Lambda_i(x),$$

where $\Lambda_i(x)$ is the **Lagrange basis polynomials** given by

$$\Lambda_i(x) = \prod_{j \neq i}^{n} \frac{x - x_j}{x_i - x_j}.$$

It is is easy to see the following fact straight from definitions:

$$\Lambda_i(x_k) = \begin{cases} 1, & \text{if } i = k, \\ 0, & \text{if } i \neq k. \end{cases} = \delta_{ij}.$$

This is an important fact to remember when writing the formula in code. The best part comes now. It turns out that we have the following truth.

Theorem. There exists a unique polynomial of degree n that passes through $n+1$ points in the plane $\mathbf{R}^2$. Such a polynomial is in fact the Lagrange interpolating polynomial.

In other words, given $n+1$ points, it is always possible to interpolate these points by a polynomial uniquely. This is very assuring from a numerical point of view. Like any (claimed) truth, this theorem requires a proof. But the proof of existence is basically the intelligent construction of the Lagrange polynomials, so we are done (thanks[5] to Warring, Euler and Lagrange). All that is left is to prove uniqueness.

[5] *Warring published it first in 1779. Euler rediscovered it in 1783. Lagrange published it again in 1795. Lagrange got the name! Source: WolframMathWorld.*

Proof. *Uniqueness.* Suppose P, Q are both such polynomials of degree n. Then we have $P(x_i) = Q(x_i) = y_i$ for all $0 \leqslant i \leqslant n$. In other words, the polynomial $R(x) = P(x) - Q(x)$ has $n+1$ roots. If $R(x) \neq 0$, then $R(x)$ has at most n roots by the Fundamental Theorem of Algebra since $\deg(R) \leqslant n$. 何! This contradicts our assumption that $R(x)$ has $n+1$ roots and so $R(x) = 0$ for all x i.e. $P(x) = Q(x)$. ■

Let's look at a concrete example.

Example. Suppose we have $n+1 = 3$ data points $(x_i, y_i) = (0,0), (1,2), (2,-1)$. The interpolation through these data points is given by the polynomial

$$P_2(x) = \sum_{i=0}^{2} y_i \Lambda_i(x) = 2\Lambda_1(x) - \Lambda_2(x).$$

Next, we compute the Lagrange basis polynomials. We don't have to compute the $i = 0$ basis polynomial since it vanishes when we calculate $P_2(x)$. The $i = 1$ basis polynomial is given by:

$$\Lambda_1(x) = \prod_{j \neq 1}^{2} \frac{x - x_j}{1 - x_j} = \left(\frac{x-0}{1-0}\right)\left(\frac{x-2}{1-2}\right) = x(2-x) = 2x - x^2.$$

The $i = 2$ basis polynomial is given by:

$$\Lambda_2(x) = \prod_{j \neq 2}^{2} \frac{x - x_j}{2 - x_j} = \left(\frac{x-0}{2-0}\right)\left(\frac{x-1}{2-1}\right) = \frac{1}{2}x(x-1) = \frac{x^2}{2} - \frac{x}{2}.$$

So the unique polynomial which interpolate the 3 data points is

$$P_2(x) = \frac{1}{2}(9x - 5x^2).$$

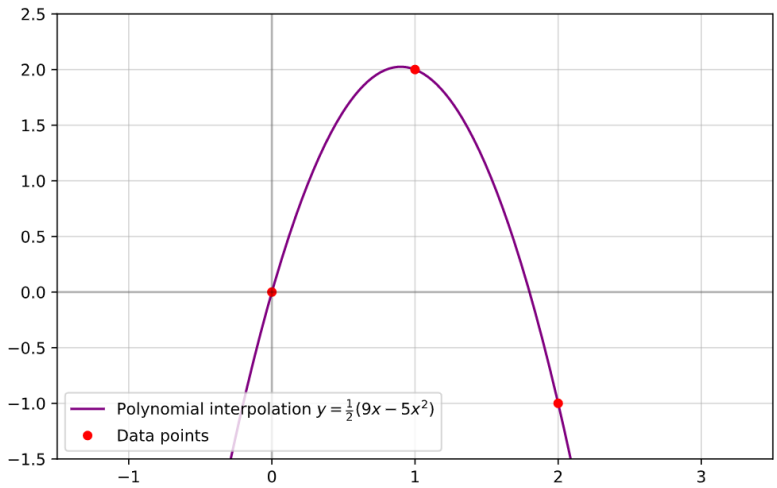

Figure 1. Polynomial interpolation of 3 data points

Finally, we look at the code. We need to first define the basis polynomial and then the Lagrange interpolation polynomial.

Algorithm (*Lagrange interpolation*)**.** Python implementation.

```
def basis_polynomial(x: Indeterminate, i: int, xs: List[Real]):
    """The Lagrange basis polynomial.

    Args:
        x: Indeterminate.
        i: Index of the basis.
        xs: x-coordinate of known data points to be interpolated.
    """
    xi = xs[i]
    prod = 1
    for xj in xs:
        if xj != xi:
            prod *= float(x-xj) / (xi-xj)
    return prod

def lagrange(x: Indeterminate, xs: List[Real], ys: List[Real]):
    """The Lagrange interpolation polynomial.

    Args:
        x: Indeterminate.
        xs: The xi values of the known points to be interpolated.
        ys: The yi values of the known points to be interpolated.

    Returns:
        float: A single interpolated point.
    """
    psum = 0
    for i, (_, yi) in enumerate(zip(xs, ys)):
        psum += yi * basis_polynomial(x, i, xs)
    return psum
```

Notice that the function returns only a single interpolated point. Utilizing `numpy` broadcasting, we can get the interpolation for a whole function given some finite amount of data points.

2.2 Interpolation errors

Awesome, we were able to approximate the function via interpolation. The problem now is that the interpolation approximates the function poorly at the boundary of the interval[6] we are approximating on. To understand why this is the case and how can we solve this, we have to discuss what error did we make when interpolating and how to minimize it.

[6] *defined by the first and last points of our finite data set*

Consider a function $f(x)$ with an interpolation polynomial $P_n(x)$ which interpolates the points (x_i, y_i) where $y_i = f(x_i)$ for $i = 0, 1, \ldots, n$. If we compute the difference between

the function and its interpolation (over these points), we should get something of the form

$$f(x) - P_n(x) = r(x)\mathcal{N}_{n+1}(x),$$

for some error function $r(x)$ and $\mathcal{N}_{n+1}(x) = \prod_{i=0}^{n}(x - x_i)$. This choice of $\mathcal{N}_{n+1}(x)$ is the "obvious" choice for the factor in the difference as it satisfies (on the rhs) the property that $f(x_i) - P_n(x_i) = 0$ for all i. In fact, we actually have a precise definition for the $r(x)$.

Theorem (*Exact difference theorem*)**.** Let $f \in C^{n+1}[a, b]$ and $(x_0, y_0), \ldots, (x_n, y_n)$ be $n + 1$ distinct points where $x_i \in [a, b]$ and $y_i = f(x_i)$. Let P_n be the degree n interpolation polynomial which interpolates these $n+1$ points. Then for any $x \in [a, b]$, there exists $\zeta_x \in [a, b]$ such that

$$f(x) - P_n(x) = \frac{f^{(n+1)}(\zeta_x)}{(n+1)!}\mathcal{N}_{n+1}(x).$$

Note. ζ_x is dependent on x. Hopefully this is obvious from the way the assumption is written.

Remark. In the lectures, Ben actually take $a = x_0$ and $b = x_n$. This is not necessary but our proof below will use this approach.

Note that the error function now is defined exactly:

$$r(x) = \frac{f^{(n+1)}(\zeta_x)}{(n+1)!},$$

which is a function of x because ζ_x was dependent on x.

The theorem essentially tells us that we can characterize the difference between $f(x)$ and $P_n(x)$ in a very precise way.

Proof. The trick is to consider the following function

$$\phi(y) = f(y) - P_n(y) - r(x)\mathcal{N}_{n+1}(y).$$

Now observe that $\phi \in C^{n+1}[a, b]$ by hypothesis; so we can differentiate and have

$$\phi^{n+1}(y) = f^{n+1}(y) - r(x)(n+1)! \qquad (*)$$

Also observe that ϕ vanishes at $n+2$ points: $x_1, \ldots, x_n$ and x. Choose any two distinct points, say, x_i, x_j where $i \neq j$. Then by Rolle's theorem, there exists $c \in [x_i, x_j]$ such that $\phi'(y) = c$. There are $n + 1$ combinations to make, so ϕ' vanishes at $n + 1$ points. Now apply Rolle's theorem to the n possible combinations from these $n + 1$ points. Then ϕ'' vanishes at n points. Continuing in this way, we have that ϕ^{n+1} vanishes at a single point $\zeta_x \in [a, b]$. Evaluating $(*)$ at $y = \zeta_x$ and putting $r(x)$ on one side, we are done. ■

The two distinct points can be (i). x_i, x_j such that $i \neq j$; or (ii). x_i, x for any i.

What's actually important for us is the following consequence.

Corollary (*Interpolation error*)**.** Assuming the assumptions in the exact difference the-

orem, we have the following bounds on the difference

$$\begin{aligned}|f(x)-P_n(x)| &\leqslant \max_{\lambda\in[a,b]}\frac{|f^{(n+1)}(\lambda)|}{(n+1)!}|\mathcal{N}_{n+1}(x)|\\ &\leqslant \max_{\lambda\in[a,b]}\frac{|f^{(n+1)}(\lambda)|}{(n+1)!}\max_{x\in[a,b]}|\mathcal{N}_{n+1}(x)|.\end{aligned}$$

Moreover, if the x-coordinate of the $n+1$ points are equally spaced, say $x_i = x_0 + ih$ for $i = 0, 1, \ldots, n$ and some $h > 0$, then for all $x \in [x_{k-1}, x_k]$ where $k = n-i$, we have

$$|f(x)-P_n(x)| \leqslant \binom{n+1}{k}^{-1}\max_{\lambda\in[a,b]}\left|f^{(n+1)}(\lambda)\right| h^{n+1}.$$

Proof. The first bound is obvious from the previous theorem. In the case that the x-coordinate of the $n+1$ points are equally spaced with $x_i = x_0 + ih$, then since $x \in [x_{k-1}, x_k]$, we have that

$$\begin{aligned}|\mathcal{N}_{n+1}(x)| = \prod_{i=0}^{n}|x_i - x| &= \prod_{i=0}^{k-1}|x_i - x|\prod_{i=k}^{n}|x_i - x|\\ &\leqslant \prod_{i=0}^{k-1}|h(k-i)|\prod_{i=k}^{n}|h(i-k+1)|\\ &= k!\,(n-k+1)!\,h^{n+1}.\end{aligned}$$

Now put the two bounds together. ∎

Remark. To see the inequality, we would advise you to try it one step at a time. The key point is that $x \in [x_{k-1}, x_k]$. Let's prove that $|x_0 - x| \leqslant kh$. This is not too difficult: we have

(1). $x \leqslant x_k = x_0 + kh$. This implies that $x - x_0 \leqslant kh$.

(2). $x \geqslant x_{k-1} = x_k - h = x_0 + kh - h$. This implies that $x - x_0 \geqslant kh - h \geqslant kh$.

Together, we get $|x - x_0| \leqslant kh$ as desired. Using the same approach, we get all the bounds $|x_i - x| \leqslant h(k-i)$ for all $i = 0, \ldots, k-1$. How about the second bound $|x_i - x| \leqslant h(i-k+1)$ for $i = k, \ldots, n$?

Again, do it one step at a time. We will prove that $|x - x_k| \leqslant h$. We have

$$x_k - h = x_{k-1} \leqslant x \leqslant x_k \leqslant x_k + h.$$

This implies that $-h \leqslant x - x_k \leqslant h$ or equivalently, $|x - x_k| \leqslant h$. From this, we can prove easily that $|x - x_{k+1}| \leqslant 2h$ via the triangle inequality:

$$|x - x_{k+1}| = |x - x_k - h| \leqslant |x - x_k| + h \leqslant 2h.$$

Likewise, we can prove the other bounds.

This corollary leads to the following information.

Error (*Interpolation error on equally spaced data points*)**.** Assuming the assumptions in the exact difference theorem, with the additional assumption that the $n+1$ data points are equally spaced with $x_i = x_0 + ih$ for $i = 0, 1, \dots, n$ and some $h > 0$, we have

$$|f(x) - P_n(x)| \sim h^{n+1}.$$

That is, $P_n(x)$ has error of order $\mathcal{O}(h^{n+1})$.

This order of error is in fact true for any choice of points x_i, and the h occurring above is then the order of the difference between two consecutive x_i. Note that on the equally spaced case, the difference was bounded by an explicit coefficient

$$\binom{n+1}{k}^{-1} h^{n+1} \max_{\lambda \in [a,b]} \left| f^{(n+1)}(\lambda) \right|.$$

It turns out that this coefficient can be further improved[7]. To do this, we will use the so-called *Chebyshev points*.

[7] *i.e. we can choose our points x_i more cleverly so that we have a coefficient that is smaller than what we currently have*

Definition (*Chebyshev points*)**.** The **Chebyshev points** over $[a, b]$ is the sequence (x_k) defined by the equation

$$x_k = \frac{a+b}{2} - \frac{a-b}{2} \cos\left(\pi \frac{k + 1/2}{n+1} \right),$$

for $k = 0, \dots, n-1$.

They are the zeros/nodes of the Chebyshev polynomials, $T_{n+1}(x)$. For simplicity, we will restrict our discussion to the case $a = -1$ and $b = 1$ as this is the convention in the definition of Chebyshev polynomials. But the question is why Chebyshev points?

Proposition (*Chebyshev is small*)**.** Let $T_{n+1}(x)$ be a Chebyshev polynomial (of the first kind) of degree $n+1$. Then

$$\max_{x \in [-1,1]} |T_{n+1}(x)|,$$

is minimal amongst all polynomials of degree $n+1$ with fixed leading coefficient.

Therefore, choosing $\mathcal{N}_{n+1}(x) = T_{n+1}(x)$ ensures that $\max_{x \in [-1,1]} |\mathcal{N}_{n+1}(x)|$ is minimal amongst all degree $n+1$ polynomials. But since $\mathcal{N}_{n+1}(x) = \prod_i (x - x_i)$, and x_i are its roots, so we must choose x_i to be the roots of the Chebyshev polynomial i.e. the Chebyshev points! Over $[-1, 1]$, these points are thus

$$x_k = \cos\left(\pi \frac{k + 1/2}{n+1} \right).$$

The only ingredient left is to know the bound on the Chebyshev polynomials. One can show that over $[-1, 1]$, the Chebyshev polynomial is bounded by

$$|\mathcal{N}_{n+1}(x)| = |T_{n+1}(x)| \leqslant 2^{-n}.$$

Thus, we have the following result.

Theorem (*Interpolation error on Chebyshev points*)**.** Assuming the assumptions in the exact difference theorem, with the additional assumption that we use Chebyshev points, we have

$$|f(x) - P_n(x)| \leqslant \frac{1}{2^n(n+1)!} \max_{\lambda \in [a,b]} \left| f^{(n+1)}(\lambda) \right|,$$

To verify that the rhs is a better bound i.e. it is smaller than what we had earlier, let's compare the coefficient

$$\frac{1}{2^n(n+1)!}$$

we get from using Chebyshev points to the coefficient we got from the equally spaced case:

$$\binom{n+1}{k}^{-1} h^{n+1}.$$

As h was the distance between two consecutive points (equally spaced) over the interval $[a, b]$, we have

$$h \sim \frac{b-a}{n}.$$

But now $a = -1$ and $b = 1$ and so $h \sim 2/n$. By using the Stirling approximation, one can then show that

$$\frac{1}{2^n(n+1)!} \leqslant \max_{k \in \{0,\ldots,n\}} \binom{n+1}{k}^{-1} h^{n+1},$$

and hence, the interpolation error in the Chebyshev points case is smaller than in the equally spaced points case. This explains why Chebyshev points is the better choice. Below is the algorithm for generating Chebyshev points.

Algorithm. Python implementation.

```
def chebyshev(a: Real, b: Real, n: int) -> List[Real]:
    """Generates a sequence of `n` Chebyshev points
    over the interval [a, b].

    Args:
        a: The startpoint of [a, b].
        b: The endpoint of [a, b].
        n: The number of Chebyshev points to be generated.

    Returns:
        A sequence of Chebyshev points.
    """
    cheb_points = []
    for i in range(n):
        cos_term = np.cos(np.pi*(i+0.5)/(n+1))
        point = float(a+b)/2 - float(a-b)/2*cos_term
        cheb_points.append(point)
    return cheb_points
```

2.3 Polynomial fits

It can happen that we have a set of points (x_i, y_i) which when drawn onto a graph, looks like a function $f(x)$ but with some noise[8]. This usually happens in practice when gathering experimental data. For example, let's say we collected data regarding the trend between time and temperature as a coffee cools down. Such collected data would have all sort of errors (noise): thermometer error, human error, heat loss due to coffee container error, etc. Suppose the data collected are points that look like an exponential function $ae^{bx} + c$ with $a, b, c \in \mathbf{R}$ on the plane $\mathbf{R}^2$. But here's the problem, we don't know what exactly a and b are.

[8] *By noise, we mean that some points are maybe offsetted up, down, left or right from the expected function.*

To give a full characterization of the trend we are observing, we would really want to know what are these values. Well, we could have try Lagrange interpolation but we just mentioned that this set of points that we know contains all sort of error and noise. If we would to interpolate it, the function that we get might approximate the true function, but not very well[9]. For example, the trend of the points is actually linear, but the interpolation could give a chaotic oscillating function.

[9] *Remember that an interpolation will go through* ***all*** *the data points we used*

Enter curve fitting. We take a step back, and instead now aim to be as near to these values by using a smaller set of parameters $\{a_i\}$ than the number of points that we have. Through curve fitting, we could now at least get an approximate value $\mathfrak{a}, \mathfrak{b}$ and $\mathfrak{c}$ so that we have the approximation function $\mathfrak{a}e^{\mathfrak{b}x} + \mathfrak{c}$. And that is the main gist of this section: to approximate coefficients of a certain approximation function.

> **Slogan.** Approximate coefficients of approximation function.

So how do we go about doing this formally? Let $f(x)$ be the expected function that we want to approximate and suppose that we have $n + 1$ known values[10] (x_i, y_i) where $y_i = f(x_i)$. Now consider the function

$$\alpha(x) = \sum_{i=0}^{m} a_i \beta_i(x)$$

[10] *Note that these are known values of the function that we want to approximate and* ***not*** *data points that we are trying to approximate.*

where $m \leqslant n$, $a_i \in \mathbf{R}$ and $\beta_i(x)$ are basis functions.

Basis functions could be a set of monomials, polynomials, exponentials or many other functions.

> **Remark.** Note that $m \leqslant n$ is a choice. If $n = m$, and we choose the basis functions to be monomials, we recover the polynomial interpolation problem. If $n < m$, then does it really makes sense? This is like approximating $n = 2$ data points with a degree $m = 3$ polynomial i.e. a cubic, which is absurd. Our case in practice is the following: consider a data science problem where you have $n = 100$ points. To see a linear trend, it suffices to approximate these data points with a degree $m = 1$ polynomial. If this did not capture the trend, maybe try a degree $m = 2$ polynomial, so on and so forth. Do you see now why our choice $m \leqslant n$ just make sense for our problem?

The game now is to use $\alpha(x)$ to approximate $f(x)$. But what does this means? This is just saying that we want to find good $m+1$ parameters $a_0, \ldots, a_m$ so that $\alpha(x)$ in some sense is "close" to $f(x)$ given the $n+1$ points we know about $f(x)$. And how do we do this? Enter the idea of least square error. Consider the following sum of difference of squares:

We made a choice by considering least square error, we could have chosen any other metric.

$$S = \sum_{i=0}^{n} (y_i - \alpha(x_i))^2 = \sum_{i=0}^{n} \left(y_i - \sum_{j=0}^{m} a_j \beta_j(x_i) \right)^2 .$$

The idea is then to minimise S with respect to the variations of a_j. That is, we want

$$\frac{\partial S}{\partial \mathbf{x}} = \mathbf{0}$$

where $\mathbf{x} = (a_0, \dots, a_m)$. So we have to solve the equation $\partial S/\partial a_k = 0$ for all $k = 0, \dots, m$. By elementary calculus, we have

$$\frac{\partial S}{\partial a_k} = -2\sum_{i=0}^{n} \beta_k(x_i)\left(y_i - \sum_{j=0}^{m} a_j\beta_j(x_i)\right) = -2\sum_{i=0}^{n} \beta_k(x_i)(y_i - \alpha(x_i)),$$

and so solving the equation $\partial S/\partial a_k = 0$ is equivalent to solving the problem $A\mathbf{x} = \mathbf{b}$ where A is the symmetric $(m+1) \times (m+1)$ matrix

$$A_{jk} = \sum_{i=0}^{n} \beta_j(x_i)\beta_k(x_i),$$

and $\mathbf{b}$ is the $m+1$ vector

$$b_j = \sum_{i=0}^{n} \beta_j(x_i)y_i.$$

The coefficients a_i are then encoded as the entries of the $m+1$ vector $\mathbf{x}$ that we want to solve; and linear algebra tells us that as long as A is invertible, this is as easy as

$$\mathbf{x} = A^{-1}\mathbf{b}.$$

Slogan. Least square error is a linear algebra problem!

We are done in finding the coefficients, but one question remains and that is: does the solution $\mathbf{x} = (a_0, \dots, a_m)$ truly minimizes S. Again, linear algebra gives us the right tools. It suffices to check that A is a positive matrix and this is easy. Given any $m+1$ vector c, we just need to show $c^T Ac \geqslant 0$. But this is just the triple sum

$$\sum_{i=0}^{n}\sum_{j,k=0}^{m} c_j\beta_j(x_i)c_k\beta_k(x_i) = \sum_{i=0}^{n}\left(\sum_{j=0}^{m} c_j\beta_j(x_i)\right)^2 \geqslant 0,$$

as desired. Finally, let's talk a bit more about the choice of basis functions as they were arbitrary throughout the discussion. To solve our coffee problem earlier, we could choose the basis $\{1, e^x, e^{2x}, \dots, e^{nx}\}$ for some $n \in \mathbf{N}$. But since a lot of functions (that we care about) has a power series expansion, the most natural candidate for the basis functions would be the monomials $\{1, x, x^2, \dots, x^n\}$. That is, we choose $\beta_j(x) = x^j$. This gives

as linear combination of monomials looks like a power series

$$A_{jk} = \sum_{i=0}^{n} x_i^{j+k}, \quad b_j = \sum_{i=0}^{n} x_i^j y_i, \quad \alpha(x) = \sum_{i=0}^{m} a_i x^i. \tag{2.1}$$

With this choice of basis, we are essentially approximating f using polynomials; hence, the name **polynomial fit**. Let's look at an example.

Example. Suppose we have the same $n + 1 = 3$ data points $(x_i, y_i) = (0,0), (1,2), (2,-1)$ as in the example in the Lagrange interpolation. We will look at fitting a polynomial of degree $m = 1$ and degree $m = 2$; both of which are less than or equal to $n = 2$. Recall that in (2.1), we have $0 \leqslant j, k \leqslant m$. So with $m = 2$,

we have to do 9 computations for A_{jk} and 3 computations for b_j. But thankfully by symmetry of A, we can do 3 computations less.

Case $m = 2$. After some painful computations, we end up with

$$A = \begin{pmatrix} 3 & 3 & 5 \\ 3 & 5 & 9 \\ 5 & 9 & 17 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix}.$$

The equation $A\boldsymbol{x} = b$ where $\boldsymbol{x} = (a_0, a_1, a_2)$ can be put into the augmented matrix form

$$\left[\begin{array}{ccc|c} 3 & 3 & 5 & 1 \\ 3 & 5 & 9 & 0 \\ 5 & 9 & 17 & -2 \end{array}\right]$$

This equation can be reduced easily using Gaussian elimination. We would end up with

$$\left[\begin{array}{ccc|c} 5 & 9 & 17 & -2 \\ 0 & 12 & 26 & -11 \\ 0 & 0 & -10 & 25 \end{array}\right].$$

So solving this, we get

$$\boldsymbol{x} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 9/2 \\ -5/2 \end{pmatrix}.$$

Therefore, the polynomial fit through these data points is given by

$$\alpha(x) = \frac{1}{2}(9x - 5x^2),$$

which coincides with the Lagrange interpolation polynomial, as expected.

Case $m = 1$. This case is more interesting. We don't have to do too much computations anymore as the matrix A and vector b here is just the "truncation" of the previous ones simply by removing the second row and column. We have

$$A = \begin{pmatrix} 3 & 3 \\ 3 & 5 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

The equation $A\boldsymbol{x} = b$ where $\boldsymbol{x} = (a_0, a_1)$ in augmented matrix form is

$$\left[\begin{array}{cc|c} 3 & 3 & 1 \\ 3 & 5 & 0 \end{array}\right] \longrightarrow \left[\begin{array}{cc|c} 3 & 3 & 1 \\ 0 & 2 & -1 \end{array}\right].$$

So solving this, we get

$$\boldsymbol{x} = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} 5/6 \\ -1/2 \end{pmatrix}.$$

Therefore, the polynomial fit through these data points is given by

$$\alpha(x) = \frac{5}{6} - \frac{1}{2}x = \frac{1}{2}\left(\frac{5}{3} - x\right).$$

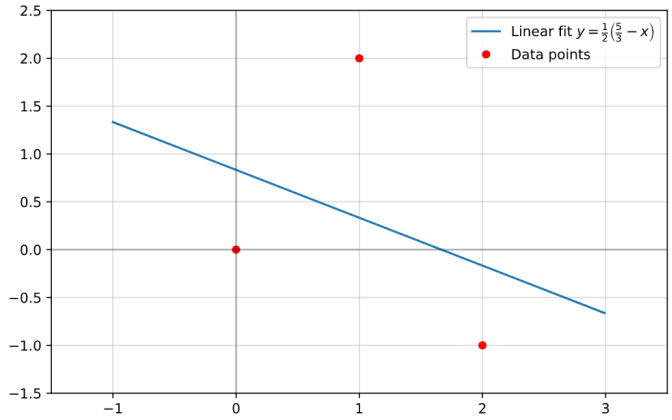

Figure 2. Linear fit of 3 data points

Let's look at the code implementation for polynomial fit.

Algorithm (Polynomial fit)**.** Python implementation.

```
def find_coeff_ai(m: int, xs: List[Real], ys: List[Real]) -> Function:
    """Find the coefficients, ai, for the polynomial fit.

    Args:
        m: Number of coefficients ai we want to find.
        xs: The xi values of the known points to be fitted.
        ys: The yi values of the known points to be fitted.

    Remark:
        Note that len(xs), len(ys) can be >= m as per the theory.

    Returns:
        A polynomial fit of the data points (xs, ys).
    """
    # Create a zero (m+1) * (m+1) matrix and a zero (m+1) vector.
    A = np.zeros((m+1, m+1))
    b = np.zeros(m+1)
    for k in range(m+1):
        b[k] = sum(xs**k * ys)
        for i in range(m+1):
            A[k, i] = sum(xs**(k+i))

    # Solve Ax = b for x.
    coefs: List[Real] = np.linalg.solve(A, b)

    def polyfit(x: Indeterminate) -> List[Real]:
        monomial_basis: List[Real] = x**np.array(range(m+1))
        return sum(coefs*monomial_basis)

    return polyfit
```

3 Linear systems

Let us now talk about solving linear systems of algebraic equations. This is one of the most important subject in numerical methods as it is used in many numerical algorithms. In fact, it relates to the question of inverting a matrix.

Let us formally state the problem. Suppose we are given a set of n linear equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_1 &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_2 &= b_2, \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n, \end{aligned}$$

where a_{ij}, b_i are given for all $i, j = 1, \ldots, n$. So we have $n^2 + n$ parameters and we are looking for n variables x_i. We can easily write this in a more linear algebra fashion by taking $A = (a_{ij})$, $\mathbf{b} = (b_1, \ldots, b_n)$ and the vector to be solved for $\mathbf{x} = (x_1, \ldots, x_n)$. So our system of equations is really just the matrix equation $A\mathbf{x} = \mathbf{b}$.

If $\det A = 0$, then this equation has infinitely many solutions or no solution at all. Numerically, this is an interesting case as one can imagine that if $\det A \approx 0$, then something weird might happen. However, we will only restrict our attention to when $\det A \neq 0$. In this case, this equation either has no solution or has a unique solution given by matrix inversion, $\mathbf{x} = A^{-1}\mathbf{b}$. So our problem is really a problem of inverting a matrix and we want to be able to do it computationally.

3.1 Gaussian elimination

One way to do it is to use Gaussian elimination. It turns out that computationally, it is more efficient and less expensive than the standard method which involves Laplace expansions. Now, the process/steps of Gaussian elimination is best viewed when the equation is written in the augmented matrix form

$$[A \mid \mathbf{b}] = \left[\begin{array}{ccc|c} A_{11} & \cdots & A_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ A_{n1} & \cdots & A_{nn} & b_n \end{array}\right].$$

This is because we will be going to do some certain modifications to these equations such that information is preserved. Let us remind ourselves what are these modifications: we can

1. Exchange two rows.
2. Multiply a row by a nonzero scalar.
3. Add two rows together.

When we were doing this analytically, we never thought about the overall steps, so to speak, in the Gaussian elimination process. It turns out that there are two fundamentals phases: elimination; and back-substitution.

3.1.1 Elimination phase

Essentially, the elimination phase is putting the augmented matrix into an echelon form. It was easy for a human to do it but how do we tell a computer how to do it? For this, we have to formalize this process a bit more. Let us try to put this simple system

$$\left[\begin{array}{ccc|c} a_1 & a_2 & a_3 & d_1 \\ b_1 & b_2 & b_3 & d_2 \\ c_1 & c_2 & c_3 & d_3 \end{array}\right],$$

into echelon form. If we wanted to make the second row, R_2 so that b_1 die, we can do the transformation

$$R_2 \to R_2 - R_1 \cdot \lambda_{R_1}(b_1),$$

where $\lambda_{R_1}(x) = b_1/(\text{leading entry of } R_1) = b_1/a_1$. Likewise, on the third row R_3 we do the transformation

In general, we have that $\lambda_R(x) := x/(\text{leading entry of row } R)$

$$R_3 \to R_3 - R_1 \cdot \lambda_{R_1}(c_1),$$

to kill the c_1 entry. At this point, we have the matrix

$$\left[\begin{array}{ccc|c} a_1 & a_2 & a_3 & d_1 \\ 0 & b_2' & b_3' & d_2' \\ 0 & c_2' & c_3' & d_3' \end{array}\right].$$

Observe that the first row is left unchanged. Such a row, R is called a **pivot** of the step and $\lambda_R(x)$ is its **elimination parameter** . It is crucial to note that in the definition of the elimination parameter, we have divided by the leading entry of the pivot and this could lead to division by zero!

> **Slogan.** Pivots must have a nonzero leading entry.

So the pivot we choose must have a nonzero leading entry. Let us assume for now that this is always the case for all pivots that we choose. We proceed to carry out this elimination phase on a subsystem such that the first column is zero. That is, we now look at this system

Intuitively, it is clear that if the row we want to choose as a pivot has a zero leading entry, then we just swap it with some other row. The problem, is we are doing this computationally. What does it mean to "swap with some other row" for a computer? This is an issue that we look into later.

$$\left[\begin{array}{cc|c} b_2' & b_3' & d_2' \\ c_2' & c_3' & d_3' \end{array}\right],$$

which is a subsystem of our original one. Let the first row be a pivot. Then the transformation

$$R_2 \to R_2 - R_1 \cdot \lambda_{R_1}(c_2'),$$

puts the matrix into

$$\left[\begin{array}{cc|c} b_2' & b_3' & d_2' \\ 0 & c_3'' & d_3'' \end{array}\right],$$

and so the original matrix is now

$$\left[\begin{array}{ccc|c} a_1 & a_2 & a_3 & d_1 \\ 0 & b_2' & b_3' & d_2' \\ 0 & 0 & c_3'' & d_3'' \end{array}\right].$$

Awesome, and we are now in echelon form. Note that it took us 2 elimination steps for this 3×3 matrix to reach echelon form. In general, it will take $n - 1$ elimination steps for an $n \times n$ matrix.

The computational implementation of the elimination phase is thus the following: Go

to the first row. Choose it as a pivot. Kill all the terms below its leading entry. Now go to the second row. Choose it as a pivot. Kill all the terms below its leading entry. So on and so forth.

Ultimately, note that there are no terms to be killed if you are at the last row. That is you need to go up to $n-1$ pivots only[11]. However, if we are able to choose a pivot, say, at row j, then we always have to kill $n-j$ terms i.e. we have to loop over $n-j$ rows for the killing step. In code, this means that we have to loop over `range(j+1, n)`. For a pivot R_i, the killing step is simply the elimination parameter, $\lambda_{R_i}(x)$ applied to the leading entry of the pivot and then we do the necessary subtraction for the row to be killed. The algorithm should now be obvious.

[11] *In code, this means that you have to run up to the $(n-2)$-th row as we started counting from 0. In Python, we implement this by looping over `range(0, n-1)` as with a step of 1, this will end at $n-2$.*

Algorithm. Python implementation.

```
def row_echelon(A: Matrix, b: Vector):
    """Put the augmented matrix [A|b] into row echelon form.

    Args:
        A: An n*n matrix
        b: An n*1 vector

    Returns:
        An upper triangular matrix A and an updated vector b
        such that [A|b] is in row echelon form.
    """
    n = len(b)
    A, b = A.copy(), b.copy()

    # Iterate through all pivots.
    for j in range(0, n-1):

        # Kill all the terms below the leading entry of pivot.
        for i in range(j+1, n):
            elim_param = A[i, j] / A[j, j]

            A[i] = A[i] - elim_param * A[j]
            b[i] = b[i] - elim_param * b[j]

    return A, b
```

3.1.2 Back-substitution phase

Now that we have a nice augmented matrix in echelon form, it is time to do a backward substitution. Intuitively, this is clear to do. However, remember that we are telling a computer to do it so we need to make the steps precise. Consider the following echelon-ed matrix:

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a_{22} & a_{23} & b_2 \\ 0 & 0 & a_{33} & b_3 \end{array}\right] \xrightarrow{\text{equivalent to}} \begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{22}x_2 + a_{23}x_3 &= b_2. \\ a_{33}x_3 &= b_3 \end{aligned}$$

It is obvious that we now have our first solution $x_3 = b_3/a_{33}$. The most human way to proceed now is just to back substitute one by one. But we will consider a different approach that has a nice computational fashion to it. The idea is to "diagonalize" the matrix by mimicking our elimination phase but now from the bottom up. So the goal is to kill the term a_{23} and a_{13}. We can do this using 2 transformations: the transformation

$$R_2 \to R_2 - R_3 \cdot \frac{a_{23}}{a_{33}}, \qquad b_2' = b_2 - b_3 \cdot \frac{a_{23}}{a_{33}} = b_2 - x_3 a_{23},$$

and the transformation

$$R_1 \to R_1 - R_3 \cdot \frac{a_{13}}{a_{33}}, \qquad b_1' = b_1 - b_3 \cdot \frac{a_{13}}{a_{33}} = b_1 - x_3 a_{13}.$$

We thus have the reduced matrix

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & 0 & b_1' \\ 0 & a_{22} & 0 & b_2' \\ 0 & 0 & a_{33} & b_3 \end{array}\right],$$

and so we get the solution $x_2 = b_2'/a_{22}$. We then proceed to kill a_{12}. To do this, use the transformation

$$R_1 \to R_1 - R_2 \cdot \frac{a_{12}}{a_{22}}, \qquad b_1'' = b_1' - b_2' \frac{a_{12}}{a_{22}} = b_1' - x_2 a_{12},$$

and so we have the more reduced matrix

$$\left[\begin{array}{ccc|c} a_{11} & 0 & 0 & b_1'' \\ 0 & a_{22} & 0 & b_2' \\ 0 & 0 & a_{33} & b_3 \end{array}\right].$$

So we get the final solution $x_1 = b_1''/a_{11}$. What's important here is that we have a general step to get each one solution. Let's take a closer look at the whole process again but now from a general point of view.

1. Firstly, note that the coefficients a_{ij} never change in this back substitution phase.

2. Secondly, observe that on the i-th step of the back substitution, we have to kill $n - i$ terms for an $n \times n$ matrix.

   just like in the elimination phase

3. Thirdly, we note that the solution is given by the general equation

$$x_{n-j} = \frac{b_{n-j}^{(j)}}{a_{n-j,n-j}},$$

   for $j = 0, 1, \ldots, n-1$ where the superscript index denotes the number of transformations that have occured.

   In the example earlier, we had $n = 3$. And our solutions were $x_3 = b_3/a_{33}$ with $j = 0$ and $x_2 = b_2'/a_{22}$ with $j = 1$.

4. Fourthly, the transformation on the b terms at each step j **fixed** can be simplified into the equation

$$\mathbf{b}' = \mathbf{b} - x_j \mathbf{A}_j$$

   where $\mathbf{A}_j = (a_{kj})$ for $k = 1, \ldots, n$.

   i.e. $\mathbf{A}_j$ is the j-th column of the matrix A. Indeed, this can be viewed as a difference of row/column vectors.

From our recollection, the algorithm for backward substitution should now be obvious. However, unlike putting a matrix into its row echelon form, it does not really make sense

to implement a lone backward substitution algorithm. So we combine these two phases into a single Gauss elimination algorithm.

Algorithm. Python implementation.

```
def gauss(A: Matrix, b: Vector):
    """Executes Gaussian elimination on the augmented matrix [A|b].

    Returns:
      A vector x such that Ax = b.
    """
    n = len(b)
    A, b = A.copy(), b.copy()
    A, b = row_echelon(A, b)

    # Backward substitution phase
    x = np.zeros(n)
    for j in range(n-1, -1, -1):
        x[j] = b[j] / A[j, j]
        b = b - x[j]*A[:, j]
    return x
```

3.2 The pivot problem

Recall that we have assumed that any choice of our pivot has a nonzero leading entry so that division by zero is averted. This was a strong assumption we make insofar that the theory and examples that we have been giving rely heavily on it. We will now discuss a technique that will not only just solve this problem, but also improve the original method we have.

As we mentioned earlier in a margin note, the solution is intuitively clear. If the row we want to choose as a pivot has leading entry zero, then we just swap it with some other row whose leading entry at that step is nonzero. But the question is how do you choose this other row? Well since we are only "afraid" of zeros, this problem is not very severe. One way to tackle it is to take the absolute value of each leading entries of pivot candidates and see which one is the biggest.

Let $[\,A \mid \mathbf{b}\,]$ be our augmented matrix where $A = (a_{ij})$ is an $n \times n$ matrix and $\mathbf{b} = (b_1, \ldots, b_n)$ is an $n \times 1$ column vector. Furthermore, let $\mathbf{A}_j = (a_{kj})$ for $k = 1, \ldots, n$ i.e. the j-th column of the matrix A, viewed as an $n \times 1$ column vector. Then fix $j \in \{1, \ldots, n\}$ and define the following set

$$\operatorname{abs} \mathbf{A}_j = \{\text{absolute values of entries of } \mathbf{A}_j\} = \{|a_{ij}| \mid i = 1, \ldots, n\}.$$

With this definition, the algorithm is the following: Suppose we are at the j-th elimination step where $j = 1, \ldots, n-1$. Then we have to do the following 4 steps.

1. Find the maximal (w.r.t absolute value) leading entry between the rows by computing $\max \operatorname{abs} \mathbf{A}_j$, say this value[12] is $a_{\lambda j}$.
2. Swap the j-th row with the λ-th row.

[12] *that is, the maximal value is at the λ-th row of the j-th column.*

3. Kill all the terms below the leading entry.
4. Repeat step 1 until $j = n - 1$.

We choose pivots up until the second last row (no more terms to kill if you are at the last row.)

Remark. Note that we find the maximal entry with respect to the absolute value. Essentially, this means that such a maximal value could very well be negative. Most importantly, however, is that it is nonzero.

Moreover, it is worth commenting that by our hypothesis it is not possible to have more than one zero in a single column $\mathbf{A}_j$. In such cases, the rows of A are not linearly independent and hence, A is singular which we have assumed not since the beginning.

The steps we wrote gave us a nice pseudocode; let's see the Python implementation. We will first implement the code for step 1, which is to find $\max \operatorname{abs} \mathbf{A}_j$.

Algorithm. Finding maximal rows.

```
def find_maximal_row(mat_column: List[float]):
    """Finds the row (element) with maximal value in `mat_column'
    and returns its index.

    Args:
        mat_column: A matrix column, viewed as n * 1 column vector.
    """
    abs_col = [abs(entry) for entry in mat_column]
    max_entry = max(abs_col)
    idx_max_entry = abs_col.index(max_entry)
    return idx_max_entry
```

Then we will implement the code for step 2, which is to swap the rows. We will do the swap in place i.e. we pass the matrix A and vector $\mathbf{b}$ by reference to the swap function.

Algorithm. Swapping rows.

```
def swap_two_rows(A: Matrix, b: Vector, i: mat_index, j: mat_index):
    """Swaps in place the i-th and j-th row of A and b."""
    A[i], A[j] = A[j].copy(), A[i].copy()
    b[i], b[j] = b[j], b[i]
```

Step 3 is really the standard Gaussian elimination which we have implemented. The only thing left is to integrate these new functions into our original code. As per the discussion, the problem really arises in the elimination phase so we don't have to do anything to the backward substitution phase code. In fact, we only need to modify the original `row_echelon()` function by adding [13] four more lines of code.

[13] *Four lines are for readability. This could in fact be written in a single line!*

Algorithm. Python implementation.

```
def row_echelon_pivot(A: Matrix, b: Vector):
    """Put the augmented matrix [A|b] into
    row echelon form using the pivot technique.
    """
    n = len(b)
```

```
    A, b = A.copy(), b.copy()

    # Iterate through all pivots.
    for j in range(0, n-1):

        # <--- Pivot technique applied --->
        leading_entries = A[j:, j]
        idx_maximal_leading_entry = find_maximal_row(leading_entries)
        idx_maximal_row = j + idx_maximal_leading_entry
        swap_two_rows(A, b, idx_maximal_row, j)

        # Kill all the terms below the leading entry of pivot.
        for i in range(j+1, n):
            elim_param = A[i, j] / A[j, j]

            A[i] = A[i] - elim_param * A[j]
            b[i] = b[i] - elim_param * b[j]

    return A, b
```

Remark. The value assigned to `idx_maximal_row` might be a bit confusing so let's elaborate on that.

In the outer loop over `range(0, n-1)`, the index j corresponds to the first row[14] of the j-th step subsystem we are considering. So there are $n - j$ rows (such rows are captured by `leading_entries = A[j:, j]`) below it and we want to choose the row whose leading entry is maximal. The index of such value is then captured by the `idx_maximal_leading_entry` variable.

[14] *In our non-pivot implementation, we chose this row as the default pivot on each step.*

The problem now is that this index is relative to the number of rows/columns[15] of the j-th subsystem matrix. If such index is k, then in the original matrix A, it is in fact the index $j + k$. And we want to swap these rows as rows of A, and not the subsystem. This explains why we added j in the following code:

[15] *doesn't matter as this is a square matrix*

```
idx_maximal_row = j + idx_maximal_leading_entry.
```

We promised earlier that by using this new pivot technique, we are also able to improve the original method. Here is the reason why. Suppose we are at the j-th step. By choosing the pivot with leading entry as large as possible, say, a_{jj}, the elimination parameter we compute

$$\lambda_{R_j}(a_{ij}) = \frac{a_{ij}}{a_{jj}},$$

for $i = j + 1, j + 2, \ldots, n$ will be as small as possible. This is because a_{jj} appears in the denominator. Consequently, we are able to make the smallest number of modifications and hence, able to keep the highest possible precision (remember that floating-point division sucks and mess up the accuracy of your result).

4 Numerical differentiation

In this chapter, we want to find an algorithm that allows us to compute the derivative of a function numerically at a point (and hence, on $\mathbf{R}$). The question is how do we tell a computer to do this?

4.1 Finite difference approximation

Let $f(x)$ be a function. Then its derivative is, by definition, just the limit

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

So in approximation, we have

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \tag{4.1}$$

for some $h > 0$ small. So the derivative of a function f at x in approximation can really be written in a computer verbatim from its definition.

```
def df(f: Function, x: Real, h: Real):
    return float(f(x+h) - f(x)) / h
```

This method is known as **finite difference approximation**; in particular (4.1) is known as the **right difference approximation** of $f'(x)$. The the word finite comes from the fact that we are not taking limits in h. Rather, we make a fixed choice of h, say, 10^{-20}. Obviously, we want to take h to be as small as possible because in the limit, this gives the exact derivative of f as we know it. But how small can we go down? Here's the problem: If h is super super small, then

think $h = 10^{-20}$ as before.

- The difference $f(x+h) - f(x)$ will be small, and so the numerator is small.
- The denominator is small — it is h itself.

So tiny divide by tiny, a disaster for a subject that demands precision. We will start to lose digits due to machine precision and hence, our evaluated result will be imprecise. The question now is how precise can we be? How small can we take h? Are there any other ways to calculate the derivative aside from the naive definition so that highest precision is attained?

4.2 The precision problem

The answer to the questions above has to do with how precise the numbers are on our computer. This is because a computer has only finite precision. Assume the precision on the computer is of the range $\pm\varepsilon$, where $|\varepsilon| > 0$. As per our discussion earlier, one could see that the issue with finite difference approximation is not about "differentiating" per se but more about precision in evaluation of very small numbers. So we could lose precision when adding, subtracting, multiplying and dividing small numbers. In our case, when we make h small, the difference $f(x+h) - f(x)$ is also super small so we may lose precision during subtraction, before even dividing by h. This is something we really need to take care of.

For example, $\varepsilon = 10^{-8}$; and this means that the computer, has in its memory, 8 digits for each number evaluated.

4.2.1 Precision during evaluation

Now let's go back to the equation defining differentiation and look at the terms. By Taylor expansion we have

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \mathcal{O}(h^3),$$

or if we view this up to the first order term only

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

This gives the derivative quotient

$$\frac{f(x+h) - f(x)}{h} = \frac{\mathcal{O}(h)}{h} = \mathcal{O}(1). \tag{4.2}$$

Now note that $f(x)$ itself can be very complicated[16]. It could be, say, defined using a quotient or sums of quotient. So in fact, as mentioned in the intro, our finite precision problem could come from this evaluation of f at x. Since we evaluated the function twice, once at x and another at $x+h$, this gives that $f(x)$ and $f(x+h)$ both have an imprecision of order[17] $\pm\varepsilon$. So the difference $f(x+h)-f(x)$ will have an imprecision of order $\pm 2\varepsilon \sim \pm\varepsilon$. Thus writing $\Delta_h(f) = f(x+h) - f(x)$, we have the quotient

$$\frac{\Delta_h f \pm \varepsilon}{h \pm \varepsilon},$$

which we can write as

$$\frac{\Delta_h f + c_1\varepsilon}{h + c_2\varepsilon},$$

for some $c_1, c_2 \in [-1, 1]$. Now, let us assume that ε is super small such that $\varepsilon \ll h$. Then, we can Taylor expand the quotient above in powers of ε, we have

$$\frac{\Delta_h f \left(1 + \frac{c_1}{\Delta_h f}\varepsilon\right)}{h\left(1 + \frac{c_2}{h}\varepsilon\right)}.$$

By expanding $1/\left(1 + \frac{c_2}{h}\varepsilon\right)$ using the power series identity

$$\frac{1}{1+x} = \sum_{i=0}^{\infty}(-1)^i x^i$$

up to second order, we obtain

$$\frac{\Delta_h f}{h}\left(1 + \frac{c_1}{\Delta_h f}\varepsilon\right)\left(1 - \frac{c_2}{h}\varepsilon\right) + \mathcal{O}(\varepsilon^2).$$

After expanding parenthesis, this gives us

$$\frac{\Delta_h f}{h} + \varepsilon\frac{\Delta_h f}{h}\left(\frac{c_1}{\Delta_h f} - \frac{c_2}{h}\right) + \mathcal{O}(\varepsilon^2)$$

Now let us observe the order of each term that has ε as a factor:

- $\frac{\Delta_h f}{h}$ has order $\mathcal{O}(1)$ by (4.2).
- $\frac{c_1}{\Delta_h f}$ has order $\mathcal{O}(1/h)$, as c_1 is just a constant and $\Delta_h f$ has order $\mathcal{O}(h)$.

[16] *That is, the algorithm that defines it is complicated*

[17] ***Note.*** *This is an assumption! We assume that this imprecision is of the same order as machine imprecision we fixed earlier. We make this assumption for simplicity.*

- $\frac{c_2}{h}$ has order $\mathcal{O}(1/h)$, as c_2 is just a constant.

This implies that we have

$$\frac{\Delta_h f}{h} + \mathcal{O}(1/h)\varepsilon + \mathcal{O}(\varepsilon^2),$$

or equivalently,

$$\frac{\Delta_h f}{h} + \mathcal{O}(\varepsilon/h) + \mathcal{O}(\varepsilon^2).$$

This says that the evaluation error is of order $\mathcal{O}(\varepsilon/h)$.

But now observe that the term $\mathcal{O}(\varepsilon/h)$ is small iff $\varepsilon \ll h$. This is consistent with the assumption we made which allows us to do the Taylor expansion in ε. The important conclusion is that $\Delta_h f/h$ is precise up to $\pm\varepsilon/h$ and for h very small, ε/h can be a bigger error than ε itself. This is problematic.

> **Slogan.** $f(x)$ and $f(x+h)$ are both precise up to order $\pm\varepsilon$, but $\Delta_h f/h$ is precise up to order $\pm\varepsilon/h$.

4.2.2 Optimal precision for differentiation

The question now is: what is the optimal value of h to compute the derivative? Clearly, h must be small enough $\ll 1$ (as otherwise, are we even differentiating the function)? And our latest discovery shows that h cannot be too small due to precision error during function evaluation so we need $h \gg \varepsilon$. So what is the optimal value h that satisfies these constraints given an ε?

> **Slogan.** Need $\varepsilon \ll h \ll 1$.

To answer this, we need to go back to the Taylor expansion. We have that

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

Using our definition and moving terms around, we have

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

So in calculating the quotient $\Delta_h f/h$, we get not exactly the derivative we want, but $f'(x)$ plus some error $\mathcal{O}(h)$. Let's now compare the two errors we got so far.

1. The error due to derivative approximation is $\mathcal{O}(h)$.
2. The error due to function evaluation[18] is of order $\mathcal{O}(\varepsilon/h)$.

[18] *at two places: x and $x+h$*

If h is too large, then the function evaluation error is very small but the error due to h is very large. If h is very small, then the opposite happens. So we need something in between. One way to mitigate this issue is by making these two errors about equal to each other. That is, we attain an optimal value[19] of h whenever

[19] *in terms of order*

$$h \approx \frac{\varepsilon}{h} \implies h \approx \sqrt{\varepsilon}$$

or in terms of order, whenever

$$\mathcal{O}(h) = \mathcal{O}(\varepsilon/h) \implies \mathcal{O}(h) = \mathcal{O}(\sqrt{\varepsilon}).$$

Slogan. Optimal h value (in terms of order) is whenever $\mathcal{O}(h) = \mathcal{O}(\sqrt{\varepsilon})$.

A typical machine precision error would be $\varepsilon \sim 10^{-16}$. So when we calculate the derivative with this optimal h value, we would attain an error of order 10^{-8}. So we cannot get more than 8 significant digits in the the numerical evaluation of a derivative.

4.3 An even more optimal h

The question now is, can we improve this optimal value? The answer is yes. We just need to be more clever when defining the derivative. One can convince oneself that the following **central difference approximation** defines a derivative as well

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x-h)}{2h}.$$

Again, we consider the finite difference and get the derivative quotient

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

Observe that the derivative error due to h is now of order $\mathcal{O}(h^2)$. Since machine precision error really doesn't change, we still have an error of order $\mathcal{O}(\varepsilon/h)$. So the optimal value of h here is whenever

$$h^2 \approx \frac{\varepsilon}{h} \implies h \approx \varepsilon^{1/3}.$$

This gives an overall error of

$$\mathcal{O}(h^2) = \mathcal{O}(\varepsilon/h) = \mathcal{O}(\varepsilon^{2/3}) \ll \mathcal{O}(\sqrt{\varepsilon}).$$

That is, with this h we would, at h^2 precision, attain an error of order $\varepsilon^{2/3} = (10^{-16})^{2/3} \sim 10^{-12}$. We can do better by considering more exotic differences like

$$\frac{f(x-2h) - f(x+2h) + 8f(x+h) - 8f(x-h)}{12h} = f'(x) + \mathcal{O}(h^4),$$

where the optimal value of h here is whenever $h \approx \varepsilon^{1/5}$ which at h^4 precision gives an overall error of order $\varepsilon^{4/5} \sim 10^{-14}$. In general we have the following.

Theorem. Let $c, h \neq 0$, $k \in \mathbf{N}$ and $\Delta_h f/ch$ be a finite difference approximation of $f'(x)$ such that

$$\frac{\Delta_h f}{ch} = f'(x) + \mathcal{O}(h^k).$$

Then the optimal value of h is whenever

$$h \approx \varepsilon^{1/(k+1)},$$

where ε is the machine precision.

A general theory about generating such exotic differences is given by the Richardson extrapolation which we will not cover.

5 Numerical integration

Just like numerical differentiation, we can use the naive definition of an integral straight from analysis. That is, we use the notion of limit of step functions. But just like in numerical differentiation, we also have more clever ways to do an integration.

As we have established when discussing differentiation, we really can't take limits the way we are used to do it. What we can do instead is modify how we define the limit. In numerical differentiation, we did this by introducing exotic finite difference quotients. Here, instead of approximation using step functions[20], we will play around with linear and quadratic polynomials.

[20] *which are constants i.e. degree* 0 *polynomials*

5.1 Midpoint method

Suppose we are integrating over $I = [a, b]$. The way we did it with Riemann sums is to approximate using rectangles and taking limits. Let's just do exactly that. First, we partition the interval we are integrating over into rectangles. So, let us divide I into n intervals of length h. Then we have the partition

$$[a, b] = \bigcup_{i=0}^{n-1} [a + ih, a + (i + 1)h].$$

The height of a rectangle in the Riemann sum is usually defined by its "geometric intersection" with the function we are trying to integrate at the rectangle's midpoint (at the x-axis). Thus, If we are integrating over the subinterval, say, $[a, a + h]$ then the midpoint is $a + h/2$, the "geometric intersection" is at $f(a + h/2)$ and so the integral of the function in this approximation is:

$$\int_a^{a+h} f(x) \approx hf\,(a + h/2) = \text{approx. area of 1 rectangle.}$$

This gives the following approximation for the integral over $[a, b]$.

Definition (*Midpoint method*)**.** Let $f \in \mathcal{R}[a, b]$, let $N = \#$subdivisions of $[a, b]$ and $h = (b - a)/N$ be the step size. Then

$$\int_a^b f(x)\,\mathrm{d}x \approx h \sum_{i=0}^{N-1} f(x_i + h/2) =: M_N(f),$$

where $x_i = a + ih$.

The subscript, N on the method M denotes the number of points we are summing to (not over!). In particular, if $N = 0$, then we are considering the first subinterval. This will be true for other methods as well.

Algorithm. Midpoint method.

```
def midpoint(f: Function, a: Real, b: Real, n: int):
    """Integration using the midpoint rule."""
    dx = float(b-a) / n
    xs = np.arange(start=a+dx/2, stop=b+dx/2, step=dx)
    fs = list(map(lambda x: f(x), xs))
    return sum(fs) * dx
```

In terms of implementation, we are done. However, since we will be dealing with a lot more numerical integration methods, it will be nice if we can compare errors between methods. This will be something that we will look into later on.

5.2 Trapezoidal rule

The trapezoidal rule is similar to the midpoint method, but instead of taking a rectangular block, we take a trapezoidal block. Analytically speaking, we are now taking linear approximations instead of constant approximations. However, "linear" here is not a tangent line at the point, rather it is more of a secant line through two points which defines our trapezoid.

Let us formalize this. Let A be the area of a trapezoidal block, then elementary geometry gives us $A = \text{height} \times \text{average base length}$. Thinking in terms of the graph of f, we have the area

$$A = \frac{h}{2}(f(a) + f(a+h)),$$

for a single trapezoidal block under the curve f. This gives us an approximation of the integral of the function using trapezoidal blocks.

Definition (*Trapezoidal rule*)**.** Let $f \in \mathcal{R}[a,b]$, let $N = \#\text{subdivisions of } [a,b]$ and $h = (b-a)/N$ be the step size. Then

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{2}\sum_{i=0}^{N-1}(f(x_i) + f(x_{i+1})) =: T_N(f),$$

where $x_i = a + ih$.

Here's is the code implementation of the method.

Algorithm. Trapezoidal method.

```
import numpy as np
def trapezoidal(f: Callable, a: float, b: float, n: int) -> float:
  dx = float(b-a) / n
  xs = np.arange(start=a, stop=b+dx, step=dx)
  fs = list(map(lambda x: f(x), xs))
  return (sum(fs) - 0.5*(fs[0] + fs[-1])) * dx
```

where we have returned:

$$\sum_{i=0}^{n-1} f(a+ih) - \frac{h}{2}\left[f(a) + f(b)\right]$$

5.3 Simpson's 1/3 rule

The Simpson's rule is similar to the midpoint and trapezoidal rule. The only difference is that we are using a second-order polynomial approximation. So, we need 3 points (corresponding to each coefficients).

Definition (*Simpson's 1/3 rule*)**.** Let $f \in \mathcal{R}[a,b]$, let $N = \#\text{subdivisions of } [a,b]$ and $h = (b-a)/N$ be the step size. Then

$$\int_a^b f(x)\,dx \approx \frac{h}{6}\sum_{i=0}^{N-1}\left[f(x_i) + 4f\left(x_i + \frac{h}{2}\right) + f(x_{i+1})\right] =: S_N(f),$$

where $x_i = a + ih$.

For the code implementation, let's look a little closer at our summand

$$\frac{h}{6}\left[f(a) + 4f\left(a + \frac{h}{2}\right) + f(a+h)\right]$$

What we can see, the endpoints of the intervals will occur in two terms $f(a)$ and $f(a+h)$, so they will have a factor of $1/3$ except for the very first and last point (a and b). While the points which are in the middle of the interval will occur just once, so they have a factor of $2/3$. In summary:

Start & endpoints: $1/6$
Points that separates subdivisions: $1/3$
Midpoints of subdivisions: $2/3$.

This gives the formula

$$\frac{h}{3}\sum_{i=1}^{N-1} f(x_i) + \frac{2h}{3}\sum_{j=0}^{N-1} f\left(x_j + \frac{h}{2}\right) + \frac{h}{6}[f(a) + f(b)],$$

where $x_i = a + ih$.

Algorithm. Simpson's 1/3 rule.

```
def simpson(f: Function, a: Real, b: Real, n: int) -> Real:
    # Step size, h
    dx = float(b-a) / n

    # First sum term, f(a + ih)
    xs1 = np.arange(start=a+dx, stop=b, step=dx)
    fs1 = list(map(lambda x: f(x), xs1))

    # Second sum term, f(a + jh + h/2)
    xs2 = np.arange(start=a+dx/2, stop=b+dx/2, step=dx)
    fs2 = list(map(lambda x: f(x), xs2))

    return (sum(fs1) + 2*sum(fs2) + 0.5*(f(a) + f(b))) * dx/3
```

5.4 Error analysis of integration methods

Let us compare the sum (rhs) and the actual analytic integration of the functions. This is not straightforward to do as we have not mentioned a specific function f. However, as per our major assumption in this module, all functions are analytic and therefore has a Taylor expansion.

The steps for our error analysis will be similar for all integration methods:

1. Taylor expand the true function and integrate this expansion over a small interval $[a, a+h]$.
2. Taylor expand the approximation for a single approximation block[21].
3. Take the difference of the two expansions; we will get an error term.
4. Now consider what happens for n intervals.

[21] *this agrees with integrating over the interval* $[a, a+h]$ *in step 1*

Before we begin analyzing the midpoint method, it is worth formalizing the notion of error for a numerical integration method.

Definition (*Error of integration method*)**.** Let $f \in \mathcal{R}[a,b]$, and let n be the number of subdivisions of $[a,b]$. Let $\mathcal{I}_k(f)$ be the numerical integration method acted on f, and evaluated over $k \leqslant n$ subdivisions of $[a,b]$. Then the **error of $\mathcal{I}_k(f)$**, denoted $E_k(f;\mathcal{I})$ is defined to be the difference

$$E_k(f;\mathcal{I}) = \left| \int_a^{a+kh} f(x)\,dx - \mathcal{I}_k(f) \right|.$$

If the method is clear, we will write $E_k(f)$. In particular, we are interested when $k = n$. In this case, the error is

Note that when $k = n$, then $a + kh = a + nh = b$.

$$E_n(f;\mathcal{I}) = \left| \int_a^{b} f(x)\,dx - \mathcal{I}_n(f) \right|.$$

Error (*Midpoint error*)**.** The order of error in the midpoint method is $\mathcal{O}(h^2)$. Furthermore,

$$E_N(f;M) \leqslant \frac{h^2|b-a|}{24} \max_{x\in[a,b]} |f''(x)|.$$

Note that error here is for the integral over $[a,b]$. We started the analysis with $(a, a+h)$ but then eventually deal with the whole interval case.

Proof. Let us make an analysis of error around $(a, a+h)$. By Taylor expanding f around a, we have

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

True integral. Integrating this over the small interval $(a, a+h)$,

$$\int_a^{a+h} f(x)\ \mathrm{d}x = hf(a) + \frac{h^2}{2}f'(a) + \frac{h^3}{6}f''(a) + \mathcal{O}(h^4).$$

Approximation block. If we (Taylor) expand our earlier approximation of this integral:

$$hf(a+h/2) = hf(a) + \frac{h^2}{2}f'(a) + \frac{h^3}{8}f''(a) + \mathcal{O}(h^4).$$

Error for 1 term. If we take the difference,

$$\int_a^{a+h} f(x)\ \mathrm{d}x - hf(a+h/2) = \frac{1}{24}h^3 f''(a) + \cdots,$$

that is, the total error of difference is $\mathcal{O}(h^3)$. So the error around a in absolute value is bounded by

$$E_1(f) \leqslant \frac{1}{24}h^3|f''(a)| + \mathcal{O}(h^4).$$

Error for N terms. So over N terms, the bound is

$$E_N(f) \leqslant \frac{N}{24}h^3 \max_{x\in[a,b]}|f''(x)| + \mathcal{O}(h^4).$$

But $b-a = Nh$ and so $N = (b-a)/h$. So the bound of the error is in fact

$$E_N(f) \leqslant \frac{|b-a|}{24}h^2 \max_{x\in[a,b]}|f''(x)| + \mathcal{O}(h^3).$$

which is $\mathcal{O}(h^2)$, as desired. ■

In the trapezoidal rule, we discover something interesting. It turns out that a linear approximation is worse than a constant approximation!

Error (*Trapezoidal rule*)**.** The order of error in the trapezoidal rule is $\mathcal{O}(h^2)$; and

$$E_N(f;T) \leqslant \frac{h^2|b-a|}{12} \max_{x\in[a,b]}|f''(x)|.$$

Moreover, it is less precise than the midpoint method.

Proof. The Taylor series expansion of f gives

$$f(x) = \sum_{k=0}^{\infty}\frac{(x-a)^k}{k!}f^{(k)}(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \cdots$$

True integral. Since one can easily prove that

$$\int_a^{a+h}\frac{(x-a)^n}{n!}f^{(n)}(a) = \frac{h^{n+1}}{(n+1)!}f^{(n)}(a),$$

and in this course every function is perfect (smooth, uniform convergence holds) then

$$\int_a^{a+h} f(x) = \sum_{k=0}^{\infty}\frac{h^{k+1}}{(k+1)!}f^{(k)}(a) = hf(a) + \frac{h^2}{2!}f'(a) + \frac{h^3}{3!}f''(a) + \mathcal{O}(h^4).$$

Approximation block. By considering the taylor expansion of $f(a+h)$, we have

$$\begin{aligned}\frac{h}{2}(f(a)+f(a+h)) &= \frac{h}{2}\left(f(a) + \sum_{k=0}^{\infty}\frac{h^k}{k!}f^{(k)}(a)\right)\\ &= hf(a) + \sum_{k=1}^{\infty}\frac{h^{k+1}}{2k!}f^{(k)}(a)\\ &= hf(a) + \frac{h^2}{2}f'(a) + \frac{h^3}{4}f''(a) + \mathcal{O}(h^4).\end{aligned}$$

Error for 1 term. Finally we look at the difference between the true and numerical

approximation.

$$\int_a^{a+h} f(x) - \frac{h}{2}(f(a) + f(a+h)) = \frac{h^3}{12} f''(a) + \mathcal{O}(h^4).$$

Error for n terms. This is the error made for 1 interval. So for N intervals with $N = (b-a)/h$, the error around a in absolute value is bounded by

$$E_N(f) \leqslant \frac{|b-a|}{12} h^2 \max_{x \in [a,b]} |f''(x)|.$$

So the error is indeed of order $\mathcal{O}(h^2)$. To see that this method is less precise, just note that the error here is bounded by a factor of $1/12$ whereas the midpoint method was bounded by a factor of $1/24 < 1/12$. ■

Finally, let's do an error analysis for the Simpson's $1/3$ rule.

Error (*Simpson's 1/3 rule*)**.** The order of error in Simpson's rule is $\mathcal{O}(h^4)$. Furthermore,

$$E_N(f; S) \leqslant \frac{h^4}{6!4} \max_{x \in [a,b]} \left| f^{(4)}(x) \right|,$$

Proof. **True integral.** Choose an interval $[a, a+h] \subseteq [a, b]$. Then

$$\int_a^{a+h} f(x) = \sum_{k=0}^{\infty} \frac{h^{k+1}}{(k+1)!} f^{(k)}(a) = hf(a) + \cdots + \frac{h^5}{5!} f^{(4)}(a) + \mathcal{O}(h^6).$$

Approximation block. Simpson's numerical approximation gives:

$$\begin{aligned} S_0(f) &= \frac{h}{6}\left[f(a) + 4f\left(a + \frac{h}{2}\right) + f(a+h) \right] \\ &= hf(a) + \cdots + \frac{h^5}{5!}\frac{25}{24} f^{(4)}(a) + \mathcal{O}(h^6). \end{aligned}$$

Error for 1 term. This implies that the difference has error bounded by

$$E_1(f) \leqslant \frac{h^5}{5!}\frac{1}{24} \left| f^{(4)}(a) \right| + \mathcal{O}(h^5).$$

Error for n terms. Since $E_N(f) = nE_1(f)$, we have

$$E_N(f) \leqslant \frac{h^4}{5!}\frac{1}{24} \max_{x \in [a,b]} \left| f^{(4)}(x) \right| + \mathcal{O}(h^5),$$

and thus the order of error is indeed $\mathcal{O}(h^4)$. ■

5.5 Gaussian quadrature

Let us focus on approximating integrals of the form

$$\int_a^b w(x) f(x)\, dx,$$

Disclaimer: This is not a section dedicated to integrating the Gaussian function e^{-x^2}. Gaussian quadrature is just another integrating method where quadrature is a fancy word for numerical integration.

where $w(x)$ is called a **weighting function** and it is *given*. The numerical approximation of this integral can be seen to take the form

$$\mathcal{G}_N(f) = \sum_{i=0}^{N} A_i f(x_i),$$

where A_i are **weights**[22] and x_i are the so-called **nodal abscissas** or just nodes. These nodes are nothing more than the nodes which divided our rectangle and trapezoid in previous methods. There is, however, a key difference for the Gaussian quadrature. In the first three methods, we chose our nodes to be evenly spaced[23]. Here, we will choose the nodes x_i and the weights A_i so that $G_N(f)$ yields the exact integral we wanted. That is, so that

$$\int_a^b w(x) f(x)\, dx = \mathcal{G}_N(f).$$

[22] *This is different from the weighting function $w(x)$, but there is a non-immediate relation.*

[23] *where each node differs from a consecutive node by "length" h.*

Remark. We will solve this problem by a reverse engineering approach i.e. the way we normally would do it when we assume things are true as an "if and only if".

Think finding δ in $\varepsilon-\delta$ proofs. When we play around with the equations, we start by saying "if this is less than ε then..." whereas the usual logic should end by the expression being less than ε.

Since functions can be generally written as a linear combination of basis functions, let us choose a basis $\{g_0, \ldots, g_n\}$ and assume the f that we are interested in can be constructed from this choice of basis. The problem is then equivalent to wanting x_i and A_i (when given $w(x)$ the weighting function) so that

$$\int_a^b w(x) g_m\, dx = \mathcal{G}_N(g_m).$$

But this is too general (and difficult) for this course, so let's make things simpler. Let us choose the standard monomial basis $\{1, x, \ldots, x^n\}$ so that our problem becomes wanting to choose x_i and A_i (when given $w(x)$ the weighting function) so that

$$\int_a^b w(x) x^m\, dx = \mathcal{G}_N(x^m).$$

Let's pause and note the amount of free parameters to choose/find here. Since we are summing from 0 up until N, there are $N+1$ nodes x_i and another $N+1$ weights A_i to be chosen. So we can impose a total of $2(N+1) = 2N+2$ conditions. That is, we can choose values of m from 0 up until $2N+1$. So the problem is now wanting to find x_i and A_i (when given a weighting function) such that

$$\int_a^b w(x) x^m\, dx = \mathcal{G}_N(x^m) = \sum_{i=0}^{N} A_i x_i^m, \quad \text{for } m = 0, \ldots, 2N+1.$$

That is, we want to solve the following system of $2N+2$ equations in $N+1$ variables x_i and $N+1$ variables A_i

$$\begin{aligned}
\int_a^b w(x) x^0\, dx &= \mathcal{G}_N(x^0) = \sum_{i=0}^{N} A_i x_i^0, \\
&\vdots \\
\int_a^b w(x) x^{2N+1}\, dx &= \mathcal{G}_N(x^{2N+1}) = \sum_{i=0}^{N} A_i x_i^{2N+1}.
\end{aligned}$$

Of course, if we are able to solve this system analytically, we will get a numerical approximation that is *exact* for all polynomials of degree $2N+1$ or less; and an approximation for functions that are not (based on) such polynomials. But the problem now is the following: while these equations (focus at rhs) are linear in A_i, they are very nonlinear in x_i. So how de wo solve these equations? It turns out that we need some cleverness to solve the problem. The trick is to choose x_i to be the zeros of certain polynomials.

Let $\phi_{N+1}(x)$ be a polynomial of degree $N+1$; and assume that x_i are all the zeros of this polynomial where $i = 0, \dots, N$. Now, consider the polynomial $P_k(x) = \phi_{N+1}(x)x^k$ for $k = 0, \dots, N$. This polynomial has degree less than $2N+1$:

$$\deg P = \deg(\phi_{N+1}x^k) \leqslant \deg \phi_{N+1} + \deg x^k = N + 1 + k \leqslant 2N + 1.$$

Consequently, we should then have

$$\int_a^b w(x)P_k(x)dx = \mathcal{G}_N(P_k).$$

We are reverse engineering our way to finding A_i and x_i. So with our hypothetical choice of A_i and x_i, the Gaussian quadrature on $\phi_{N+1}(x)$ gives the following *exact* numerical approximation because its degree is less than $2N+1$.

But by hypothesis $\phi_{N+1}(x_i) = 0$ and so $P(x_i) = 0$. Thus in fact, we have the condition

$$\mathcal{G}_N(P_k) = 0, \quad \text{for } k = 0, \dots, N.$$

We finally have something that we can sensibly solve. WLOG[24], let us assume that $\phi_{N+1}(x)$ is a monic polynomial and write

$$\phi_{N+1}(x) = \sum_{i=0}^{N+1} c_i x^i,$$

[24] *otherwise, factor out the leading coefficient and absorb it into the other coefficients like A_i and coefficients of $w(x)$.*

where $c_{N+1} = 1$. Then our condition above gives us

$$0 = \mathcal{G}_N(P_k) = \int_a^b w(x)x^k \sum_{i=0}^{N+1} c_i x^i \, dx,$$

for $k = 0, \dots, N$. That is, we now have the system of $N+1$ equations

$$\begin{aligned} \int_a^b w(x)x^0 \sum_{i=0}^{N+1} c_i x^i \, dx &= 0, \\ &\vdots \\ \int_a^b w(x)x^{N+1} \sum_{i=0}^{N+1} c_i x^i \, dx &= 0; \end{aligned}$$

and we can solve these easily to get all the coefficients c_j which defined our polynomial $\phi_{N+1}(x)$. We will omit the process here, but just believe us that we now have the relation

$$\sum_{i=0}^{N} c_i h_{i+k} = -h_{N+1+k},$$

This is really just a rewriting our equations in terms of c_i. Nothing more, nothing less.

for $k = 0, \dots, N$ (so again, $N+1$ equations) where h_i is the so-called **moment** defined by the integral

$$h_i = \int_a^b w(x)x^i \, dx, \quad i = 0, \dots, N.$$

But since $w(x)$ was given, the moment is something that we can evaluate. So we know h_i. Remember that our goal was to find the weights A_i and nodes x_i. We chose our x_i as the roots of $\phi_{N+1}(x)$ so we are left to find the weights. But now notice that our moments (which we know their value) are in fact just $h_m = \mathcal{G}_N(x^m)$ by definition and so we may write

$$h_m = \sum_{i=0}^{N} A_i x_i^m, \quad m = 0, \dots, N.$$

Thus, getting A_i is just simple linear algebra now.

5.5.1 Everything simplified and code

Let's put all of this back together but let the logic flow correctly this time. We can condense everything into only 6 steps.

1. Choose a value $N \in \mathbf{N}$. This determines how many nodes x_i and weights A_i we would have.

   Obviously, the higher the N the better precision we get. But it is more computationally expensive.

2. Choose a weighting function $w(x) > 0$.

3. Calculate the moments h_i. There are $N + 1$ of them to be calculated:

$$h_i = \int_a^b w(x) x^i \, dx, \quad i = 0, \dots, N.$$

```
from scipy.integrate import quad
def h(i: int, a: Real, b: Real) -> Real:
    """The moment at index i.

    Args:
        i: Index ranging from 0 up to n.
        a: Lower bound of interval to integrate over.
        b: Upper bound of interval to integrate over.
    """
    # Choose a weight function. Here we take w(x) = 1.
    WEIGHT_FUNC = lambda x: 1
    integrand = lambda x: WEIGHT_FUNC(x) * x**i
    return quad(integrand, a, b)[0]
```

   In general we do not need the `quad()` method from the `scipy.integrate` module. The moments are usually something we can compute analytically by hand.

4. Calculate the c_i to get the polynomial $\phi_{N+1}(x)$. This can be done by solving the system of $N + 1$ linear equations

   recall that the c_i were the coefficients of our polyomial $\phi_{N+1}(x)$.

$$\sum_{i=0}^{N} c_i h_{i+k} = -h_{N+1+k}, \quad k = 0, \dots, N.$$

Remark. We can view this as solving a matrix equation. Let $\mathcal{H} = (H_{ki})$ be the symmetric matrix $H_{ki} = h_{k+i} = h_{i+k}$ and $\mathbf{h} = (-h_{N+1}, -h_{N+2}, \dots, -h_{2N+1})$. Then we want to find the vector $\mathbf{x} = (c_1, \dots, c_N)$ such that

$$\mathcal{H}\mathbf{x} = \mathbf{h}.$$

The solution is given by $\mathbf{x} = \mathcal{H}^{-1}\mathbf{h}$.

5. Find the zeros of $\phi_{N+1}(x)$. This gives our nodes x_i.

This is perhaps the most difficult part to solve as it is not a linear step and so no linear algebra :(

6. Calculate the weights A_i. This can be found by solving the system of $N+1$ linear equations

$$h_m = \sum_{i=0}^{N} A_i x_i^m, \quad m = 0, \ldots, N.$$

Remark. Again we can view this as solving a matrix equation. Let $\mathcal{X} = (X_{mi})$ be the matrix defined by $X_{mi} = x_i^m$ and $\mathbf{h}' = (h_0, \ldots, h_N)$. Then we want to find the vector $\mathbf{A} = (A_1, \ldots, A_N)$ such that

$$\mathcal{X}\mathbf{A} = \mathbf{h}'.$$

The solution is given by matrix inversion $\mathbf{A} = \mathcal{X}^{-1}\mathbf{h}'$.

The code for step 4, 5 and 6 are given below.

```
def nodes_and_weights(n: int, a: Real, b: Real):
    """Computes ci, xi and Ai.

    Args:
        n: Number of nodes and weights to generate.
        a: Lower bound of interval to integrate over.
        b: Upper bound of interval to integrate over.

    Returns:
        xs: The nodes xi as an (n+1) * 1 np.array
        As: The weights Ai as an (n+1) * 1 np.array
    """

    # STEP 4: Finding ci
    B = np.zeros((n+1, n+1))
    u = np.zeros(n+1)

    # Rewrite as a matrix equation; then solve.
    for k in range(n+1):
        u[k] = - h(n+1+k, a, b)
        for i in range(n+1):
            B[k,i] = h(k+i, a, b)
    cs = np.linalg.solve(B, u)

    # c_(n+1) = 1; we assumed Phi_(n+1) to be monic.
    cs = np.append(cs, [1])

    # STEP 5: Finding xi
    xs = np.roots(cs[::-1]).real

    # STEP 6: Finding Ai
    As = np.zeros(n+1)
```

```
        # Rewrite as a matrix equation; then solve.
        for k in range(n+1):
            u[k] = h(k, a, b)
            for i in range(n+1):
                B[k,i] = xs[i]**k
        As = np.linalg.solve(B, u)

        return xs, As
```

Note that in the code for step 6 above, we have reused the `B` matrix and `u` vector. We could have created new ones but this is superfluous as we need a matrix (resp. vector) of the same dimension and each entry of the matrix (resp. vector) will be overwritten anyways.

The Gaussian quadrature is really just a simple sum once we know the weights A_i and nodes x_i. So the code implementation is straightforward once we have our two methods above.

Algorithm. Gaussian quadrature.

```
def gauss_quadrature(f: Function, a: Real, b: Real, n: int) -> Real:
    """Integration using the Gauss's quadrature rule.

    Args:
        f: Function to be integrated.
        n: Number of nodes and weights to generate.
        a: Lower bound of interval to integrate over.
        b: Upper bound of interval to integrate over.
    """
    nodes, weights = nodes_and_weights(n, a, b)
    return sum(weights * f(nodes))
```

Before we finish with this section, let's make one last note.

Remark. A final remark on the Gaussian quadrature.

1. All of our computations depends on our choice of N and $w(x)$.
2. Note that these steps also rely on the choice of boundary condition i.e. the value a and b of the interval $[a, b]$ that we want to integrate over.

   Typically, it is possible to do a change of variable and rescale everything so that $a = -1$ and $b = 1$ i.e. we do the integral on $[-1, 1]$.
3. If $w(x) = 1$, the polynomial $\phi_{N+1}(x)$ we get is the so-called **Legendre polynomial**. They have the general formula

$$\phi_N(x) = \frac{N!}{(2N)!} \frac{\partial^N}{\partial x^N} (x^2 - 1)^N$$

5.6 Monte-Carlo method

The Monte-Carlo method is completely different to what we have seen so far: it is based on repeated random sampling. However, it is the simplest in terms of idea and code. Moreover, it has several advantages like being extremely efficient. Remember that we had to specify a length h for our abscissas? We don't need it anymore. This allows the method to be generalized to higher dimensions quite easily.

Let us fix a dense[25] rectangular grid, $R \subseteq \mathbf{R}^2$ which we will call a **canvas** which will act as a universal set (in a set theory sense). Now further suppose that we have a dense surface $S_1 \subseteq R$ whose area we are interested in calculating. Then the complement of S_1 relative to R is $S_2 = R \backslash S_1$ and assume that it is also dense. So we have that

$$\mathcal{A}(R) = \mathcal{A}(S_1) + \mathcal{A}(S_2),$$

where $\mathcal{A}(\cdot)$ is the area of of the surface. Let's set a counter[26] and choose a random point in the canvas R in a uniform distribution[27] fashion. If the point is in S_1, then we increase the counter by 1. Otherwise, we don't do anything. Suppose that after N times of repeating this procedure (where N is sufficiently large), the counter now has value n. Then intuitively, we should have

$$\frac{n}{N} \approx \frac{\mathcal{A}(S_1)}{\mathcal{A}(R)} \implies \mathcal{A}(S_1) \approx \mathcal{A}(R)\frac{n}{N}.$$

The idea is that in the limit, we should have an exact value, although of course, this is numerically impossible. So the Monte-Carlo method in a single sentence is:

choose N uniformly distributed points and take ratios.

And if we need more accuracy, we choose $N+1$ points; so on and so forth.

[25] *dense in a human sense is sufficient, but you can think about it in a topological sense as well; and of course with $\mathbf{R}^2$ given the usual topology.*

[26] *starting at 0.*

[27] *That is, there's no tendency to choose one point in R more than another point (also in R).*

5.6.1 Probability analysis

Let us prove that our intuition is indeed correct. For this, we need to make a careful probability analysis. Let X be the number of points we get inside S_1 out of N independent trials. Then clearly $X \sim \text{Bin}(N, p)$ i.e. it follows a binomial distribution with probability function

$$\mathbb{P}_k(N) = \binom{N}{k} p^k \, q^{N-k},$$

where p is the probability that the point chosen is inside S_1 and $q = 1-p$ is the probability that the point chosen is not inside S_1 (and hence, in S_2). Clearly, we have

$$p = \frac{\mathcal{A}(S_1)}{\mathcal{A}(R)}, \quad q = \frac{\mathcal{A}(S_2)}{\mathcal{A}(R)}.$$

Convince yourself by considering $N = 1$; this is just a Bernoulli distribution. If you are still not convinced, google "geometric probability".

Let $\alpha = k/N$ so that we can write $k = \alpha N$. Now let us consider what happens as $N \to \infty$. By Stirling's approximation, we have that $x! \sim \exp(x \log x - x)$ for x large. So for k and N large, we have

$$\binom{N}{k} = \frac{N!}{k!(N-k)!} \sim e^{N[(\alpha-1)\log(1-\alpha) - \alpha \log \alpha]}$$

This is a very painful computation. Rewriting only in terms of k and N, the approximation gives

$$e^{N \log(N/k) + (k-N)\log(N/k-1)}$$

Writing $p^k = e^{k \log p}$ and $q^{N-k} = e^{(N-k)\log q}$, we thus have

$$\mathbb{P}_k(N) \sim e^{N g(\alpha)},$$

where we have

$$g(\alpha) = (\alpha - 1)\log(1-\alpha) - \alpha\log\alpha + \alpha\log p + (1-\alpha)\log q.$$

There's a more interesting observation: in fact, we can prove that this choosing uniform points inside of S_1 business follows exactly a Gaussian distribution. How? By uniqueness of density functions, it suffices to show that $\mathbb{P}_k(N)$ follows the Gaussian's (distribution) density function. To do this, we need to make some small observations.

We claim that $f(\alpha) := e^{Ng(\alpha)} \sim \mathbb{P}_k(N)$ has a maximum at $\alpha = p$. To do this, we simply find zeros of $f'(\alpha)$. We first differentiate f:

$$f(\alpha) = e^{Ng(\alpha)} \implies \frac{f'(\alpha)}{f(\alpha)} = Ng'(\alpha) \iff f'(\alpha) = Nf(\alpha)g'(\alpha).$$

By hypothesis, $N > 0$ and by definition $f(\alpha) > 0$. So zeros of $f'(\alpha)$ is given by zeros of $g'(\alpha)$. It is trivial to see that

$$g'(\alpha) = \log(1-\alpha) - \log\alpha + \log p - \log q, \qquad g''(\alpha) = \frac{1}{\alpha - 1} - \frac{1}{\alpha}.$$

This implies that g' vanishes at p, and so f' vanishes at p. To see that this is a maxima, we compute f'':

$$f''(\alpha) = N(f'(\alpha)g'(\alpha) + f(\alpha)g''(\alpha)) = Nf(\alpha)(Ng'(\alpha)^2 + g''(\alpha)).$$

But observe that $g''(p) < 0$ and so $f''(p) < 0$ i.e. $f''(p)$ is indeed a maxima. Moreover, it is clear that $g(\alpha) < 0$ for $\alpha \neq p$ and so, $0 \leqslant f(\alpha) \leqslant 1$. That is, $f(\alpha)$ really defines a probability measure. So we have shown that for N large, it is most likely that we have

The moral of showing f has a maxima at p is that $f(\alpha)$ behaves like the Gaussian pdf. Next, we will see that it really looks like it.

$$\frac{k}{N} = \alpha = p = \frac{\mathcal{A}(S_1)}{\mathcal{A}(R)},$$

which was our intuition earlier!

5.6.2 A hidden Gaussian and the $\mathcal{O}(1/\sqrt{N})$ error

Next, we Taylor expand g around p and neglect $\mathcal{O}(\alpha^3)$ terms to have

$$g(\alpha) \approx g(p) + g'(p)(\alpha - p) + \frac{(\alpha-p)^2}{2}g''(p) = \frac{(\alpha-p)^2}{2}g''(p),$$

where the last equality comes from the fact that $g(p) = 0$ by definition and $g'(p) = 0$ by maximality at p. Now what is $g''(p)$? Well, we get this simply by plugging p into $g''(\alpha)$ to have

$$g''(\alpha) = \frac{1}{q} - \frac{1}{p} = -\frac{1}{pq},$$

and so

$$g(\alpha) \approx -\frac{(\alpha-p)^2}{2pq} = -\left(\frac{k}{N} - p\right)^2 \frac{1}{2pq} = -\frac{1}{2}\left(\frac{k - pN}{N\sqrt{pq}}\right)^2.$$

Recalling that $\mathbb{P}_k(N) \sim e^{Ng(\alpha)}$, we thus have the following theorem.

Theorem. For sufficiently large N, we have

$$\mathbb{P}_k(N) \sim e^{-\frac{1}{2}\left(\frac{k-pN}{\sqrt{Npq}}\right)^2}.$$

That is $k \sim \mathcal{N}(\mu, \sigma^2)$ is a Gaussian distributed random variable, where $\mu = pN$ and $\sigma = \sqrt{Npq}$.

actually, we are missing a $1/\sqrt{2\pi}$ term for this to be a Gaussian pdf, but this is only an approximation in the limit so this term is quite negligible.

Remark. In the lecture, Ben viewed the resulting equation as the density function for the Gaussian distribution of k/N. This is quite similar to the Gaussian distribution of k, but there's a subtle difference.

Error analysis. To see the error of the Monte-Carlo method, let's look at the behaviour of k around the center of the distribution i.e. at pN. We observe that

$$k \approx pN \pm \sigma = pN \pm \sqrt{Npq}.$$

This implies that

$$\frac{k}{N} \approx p \pm \frac{\sqrt{Npq}}{N} = p \pm \sqrt{\frac{pq}{N}},$$

which further implies that

$$p \approx \frac{k}{N} \pm \sqrt{\frac{pq}{N}}.$$

Recalling that $\mathcal{A}(S_1) = p\mathcal{A}(R)$, we have that

$$\mathcal{A}(S_1) = \mathcal{A}(R)\left(\frac{k}{N} \pm \sqrt{\frac{pq}{N}}\right).$$

But recalling $\mathcal{A}(S_2) = \mathcal{A}(R)q$, we have

$$\mathcal{A}(S_1) = \mathcal{A}(R)\frac{k}{N} \pm \underbrace{\sqrt{\frac{\mathcal{A}(S_1)\mathcal{A}(S_2)}{N}}}_{\text{error term}}.$$

From here, we can see that the error of the Monte-Carlo method is of order $\mathcal{O}(1/\sqrt{N})$.

Error (*Monte-Carlo error*)**.** The error of the Monte-Carlo method over N trials is of order $\mathcal{O}(1/\sqrt{N})$.

This is larger relative to errors of other integration methods we have seen so far. Previously, we usually have errors of order $\mathcal{O}(h^p)$ for some $p > 0$ and $h = 1/N$ is small. So, the Monte-Carlo method is not as precise as the other methods. However, it does have its own advantages; a lot in fact. Here are a few:

(1). It is very simple to implement[28].

(2). There's an obvious way to gain more precision: just increase N (i.e. choose more points).

(3). It is very flexible: can easily generalize to higher dimensions.

[28] *if you understand the core idea of the Monte-Carlo method, you can implement it right after waking up, in your "mamai" state; or perhaps, even in your dreams.*

Here's a bonus flex: You can brag to your friends by computing the value of π in code. How? Just execute the two-dimensional Monte-Carlo to calculate the area of a unit circle.

We now have enough knowledge to implement the Monte-Carlo in code. Below is an example code of the method where we find the area of the disc $x^2 + y^2 < 1$.

Algorithm. Monte-Carlo integration for the disc $x^2 + y^2 < 1$.

```
def monte_carlo_circle(N: int):
    n = 0
    canvas_area = 2*2

    for _ in range(N):
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)

        if (x**2 + y**2 < 1):
            n += 1

    p = n / float(N)
    q = 1 - p
    area = p * canvas_area
    error = canvas_area * np.sqrt(p*q/N)

    return area, error
```

A generalized code for the Monte-Carlo method is not too difficult to write if we assume that we always sample out of a rectangular canvas. Below is an implementation of a two-dimensional case.

Algorithm. Monte-Carlo integration.

```
def monte_carlo(N: int,
                x_bound: Tuple,
                y_bound: Tuple,
                boundary_cond: Function):
    """Integration using the Monte-Carlo method

    Args:
        N: Number of times to sample.
        x_bound: Lower and upper bound in the x-axis.
        y_bound: Lower and upper bound in the y-axis.
        boundary_cond: Boundary condition of the surface;
          a function which returns a bool.
    """
    n = 0
    x_low, x_hi = x_bound
    y_low, y_hi = y_bound
    canvas_area = (abs(x_low)+abs(x_hi)) * (abs(y_low)+abs(y_hi))

    for _ in range(N):
        x = random.uniform(x_low, x_hi)
        y = random.uniform(y_low, y_hi)
```

```
        if boundary_cond(x, y):
            n += 1

    p = n / float(N)
    q = 1 - p
    area = p * canvas_area
    error = canvas_area * np.sqrt(p*q/N)

    return area, error
```

Remark. Note that the `boundary_cond` parameter must be a function with two parameters `x` and `y` which returns a `bool`. Using the disk example above, we can use the boundary function defined in code by

```
def boundary_cond(x, y) -> bool:
    return x**2 + y**2 < 1
```

Using the two-dimensional Monte-Carlo method, we can easily compute Riemann integrals of real-valued function in one real variable $f : \mathbb{R} \to \mathbb{R}$ since the graph of f lies in the real plane $\mathbb{R}^2$.

Example. To compute

$$\int_{-1}^{2} x^2 \, \mathrm{d}x,$$

we sample from the rectangular canvas

$$[-1, 2] \times [\min_{x \in [-1,2]} x^2, \max_{x \in [-1,2]} x^2] = [-1, 2] \times [0, 4].$$

In code, this is

```
x = random.uniform(-1, 2)
y = random.uniform(0, 4)
```

To compute the area under the graph of x^2, we use the boundary condition

```
def boundary_cond(x, y) -> bool:
    return y < x**2
```

so that we increment `n += 1` whenever `y` (chosen from a uniform distribution) is under the graph of `x**2` i.e. `y < x**2`.

To generalize to three dimension, we need another parameter `z_bound` and now compute the volume `canvas_volume` in the obvious way (i.e. the way we usually compute volumes of cuboids). Then sample from a uniform distribution within this `z_bound`. And finally, the `boundary_cond` should now have a third parameter.

6 Ordinary differential equations

The goal of this chapter is easy: solve an initial value problem. That is, we want to solve an n-th order ordinary differential equation (ODE) given n initial value conditions. Let us recall what this all means.

Definition (*ODE*). An **n-th order ODE** is an equation of the form

$$y^{(n)}(x) = f(x, y, y', \dots, y^{(n-1)})$$

where y is a function of x and $y^{(i)}(x) = d^i y/dx^i$.

Note that f can be as nonlinear as you like. For example $f = x^2 + y^3 + \cos(y'')$.

Example (*Simplest ode*). The equation $y' = x$ has the solution $y = x^2/2 + C$, where C is an integration constant.

Example (*Second-order ode*). The equation $y'' = -y$ has the solution $y = A\sin(x) + B\cos(x)$, where $A, B \in \mathbf{R}$ are integration constants.

Note that an n-th order ODE can always be transformed into n first-order ODE. Using our definition above, this transformation can be done in the following way.

$$\begin{aligned} y_0 &:= y, \\ y_1 &:= y', \\ y_2 &:= y'', \\ &\vdots \\ y_{n-1} &:= y^{(n-1)}. \end{aligned}$$

Note that these are all functions.

Under this transformation, we obtain n first-order ODE by differentiating everything once.

$$\begin{aligned} y_0' &= y_1, \\ y_1' &= y_2, \\ y_2' &= y_3, \\ &\vdots \\ y_{n-2}' &= y_{n-1}, \\ y_{n-1}' &= f(x, y_0, y_1, \dots, y_{n-1}). \end{aligned}$$

In general, we get $y_i' = dy_i/dx = y_{i+1}$. This is an easy first order ODE.

To see this equality, note that $y_{n-1}' = y^{(n)}$ and now stare at our ODE blue box definition above.

Compactly and conveniently, we may rewrite this as a vector function

$$\mathbf{y}'(x) = \mathbf{F}(x, \mathbf{y}),$$

where $\mathbf{F}(x, \mathbf{y}) = (y_1, y_2, \dots, y_{n-1}, f(x, \mathbf{y}))$ and $\mathbf{y} = (y_0, \dots, y_{n-1})$. Conceptually, we can think of solving only "one" first order ODE here as in fact each entry of $\mathbf{y}'(\mathbf{x}) = \mathbf{F}(x, \mathbf{y})$ defines a first order ODE.

Remark. It is worth mentioning that $\mathbf{F}$ is a function of x and $\mathbf{y}$ because of the last component $f(x, \mathbf{y})$. On the contrary, we write $f(x, \mathbf{y})$ to mean something different. Here, we mean that f is a function of the variables $\mathbf{y} = (y_0, y_1, \dots, y_{n-1})$. Think the notation as appending to the list of variables. And yes, this is an abuse of notation.

This may be unnecessary, but again we emphasize that this vector notation is just

rewriting what we had earlier — the n first order ODE — where using the notation $\mathbf{y}'_{(i)} = \mathbf{F}_{(i)}$ to mean the i-th component of the vector equation, we have n first-order ODEs

$$\begin{aligned} y_0' = \mathbf{y}'_{(0)} &= \mathbf{F}_{(0)}(x, \mathbf{y}) = y_1, \\ y_1' = \mathbf{y}'_{(1)} &= \mathbf{F}_{(1)}(x, \mathbf{y}) = y_2, \\ &\vdots \\ y_{n-2}' = \mathbf{y}'_{(n-2)} &= \mathbf{F}_{(n-2)}(x, \mathbf{y}) = y_{n-1}, \\ y_{n-1}' = \mathbf{y}'_{(n-1)} &= \mathbf{F}_{(n-1)}(x, \mathbf{y}) = f(x, \mathbf{y}). \end{aligned}$$

In general, solving an ODE would give an integration constant i.e. we have a family of solutions. However, most of the time we want a unique solution. This is done by supplementing the ODE with initial value conditions. Here, the initial condition is a vector

$$\mathbf{y}(a) = \mathbf{c}.$$

An ODE together with its initial value condition is called an **initial value problem**. As we will see, we tackle initial value problems quite generally. This means that we develop methods without caring what f is. We also don't care whether we know how to solve it analytically or not. For example, we have no problems solving the following ODE

$$y' = \sin(y) + \cos(x)$$

numerically, albeit this is a nightmare if we would want to solve it analytically.

6.1 Euler method

Let us consider an easier method first, the Euler method. From the definition of a derivative, we know that

$$\mathbf{y}'(x) \approx \frac{\mathbf{y}(x+h) - \mathbf{y}(x)}{h},$$

for some h small. But instead of using this to evaluate the approximate derivative, we will use it to approximate the value of $\mathbf{y}(x+h)$ given that we know what is $\mathbf{y}(x)$. Recalling that $\mathbf{y}' = \mathbf{F}(x, \mathbf{y})$, we thus have the approximation

$$\mathbf{y}(x+h) \approx \mathbf{y}(x) + h\mathbf{F}(x, \mathbf{y}).$$

We can do this in a recursive fashion. Let us suppose that we are given the initial condition $\mathbf{y}(a) = (c_0, \ldots, c_{n-1})$ at $x = a$. It is thus a natural choice to start this process at $x = a$.

$$\mathbf{y}(a+h) \approx \mathbf{y}(a) + hF(a, \mathbf{y}(a)).$$

Observe that we can calculate everthing on the rhs, so we now know what is $\mathbf{y}(a+h)$. Now do the same thing at $x = a + h$.

$$\mathbf{y}(a+2h) \approx \mathbf{y}(a+h) + h\mathbf{F}(a+h, \mathbf{y}(a+h)),$$

Again, observe that we know everything on the rhs, so we get what was unknown on the lhs. Continuing this recursion, we have that

$$\mathbf{y}(a+(n+1)h) \approx \mathbf{y}(a+nh) + h\mathbf{F}(a+nh, \mathbf{y}(a+nh)).$$

Definition (*Euler's method*)**.** Let $\mathbf{y}'(x) = \mathbf{F}(x, \mathbf{y})$ be an n-th order ODE with initial value conditions $\mathbf{y}(a) = \mathbf{c}$ for some $a \in \mathbf{R}$, $\mathbf{c} \in \mathbf{R}^n$. The solution to this initial value problem is given by the recursion

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{F}(x_n, \mathbf{y}_n),$$

where $\mathbf{y}_n = \mathbf{y}(x_n)$ and $x_n = a + nh$; and $h \in \mathbf{R}$ is a constant chosen to be small.

n-th order ODE in its n first-order ODE form.

Remark. Two important remarks.

1. Note that h must be sufficiently small so that our approximation is good enough, just as in the case with derivatives.
2. Note that at every step, we are actually approximating and we never had an equality. This means the error of this approximation accumulates over time!

Using the Euler method, we are able to evaluate the function $\mathbf{y}$ which solves, at least numerically, the ODE at a myriad of points. Once we have the value $\mathbf{y}$ at many points, we can apply various methods, like Lagrange interpolation and curve fitting, to get the full solution curve. Let's first see an easy direct implementation of Euler's method.

Example. Consider the first-order differential equation $y' = \cos x$ with the initial value condition $y(0) = 1$. Let $h = 1/2$ be the step size. We will do two steps of the Euler's method.

Step 1.

$$y\left(\frac{1}{2}\right) = y(0) + \frac{1}{2}\cos 0 = \frac{3}{2}.$$

Step 2.

$$y(1) = y\left(\frac{1}{2}\right) + \frac{1}{2}\cos\left(\frac{1}{2}\right) = \frac{3}{2} + \frac{1}{2}\cos\left(\frac{1}{2}\right) \approx 1.93879.$$

Observe that this differential equation can be easily solved analytically as it is separable. It has the solution $y = \sin(x) + 1$ and observe that

$$y(1) = \sin(1) + 1 \approx 1.841147.$$

So the error of the Euler's method (in two steps) is $\varepsilon \approx 0.097643$, as it should:

$$\varepsilon \leqslant 0.1 = \frac{1}{5}h,$$

so indeed we have $\varepsilon = \mathcal{O}(h)$.

Moral of the example above? Use smaller h to reduce error if you want to use the Euler's method. Of course, this comes at the expense of more computational power; but what can you do, life's not always fair after all. Now, we'll see a more complicated example which utilizes all the formalization we've done so far to get a better overall picture.

Example. Consider the second-order ordinary differential equation $y'' = -y$ with the initial value condition $y(0) = 2$ and $y'(0) = 1$. In our ODE definition, we can rewrite this as

$$y'' = f(x, y, y'),$$

with $f(x, y, y') = -y$. By using the transformation $y^{(i)} \mapsto y_i$, we can further rewrite

this as

$$\mathbf{y}'(x) = \mathbf{F}(x, \mathbf{y}),$$

with $\mathbf{y} = (y_0, y_1) = (y, y')$ and $\mathbf{F}(x, \mathbf{y}) = (y', f(x, \mathbf{y}))$. Now, we execute Euler's method to find the solution to this ODE. Fix $h > 0$ very small. We will start at $x = 0 + h$ since we know what the values are for $x = 0$ due to the initial conditions.

$$\begin{aligned}
\begin{pmatrix} y_0(h) \\ y_1(h) \end{pmatrix} = \mathbf{y}(h) &= \mathbf{y}(0) + h\mathbf{F}(0, \mathbf{y}(0)) \\
&= \begin{pmatrix} y_0(0) \\ y_1(0) \end{pmatrix} + h \begin{pmatrix} y'(0) \\ f(0, \mathbf{y}(0)) \end{pmatrix} \\
&= \begin{pmatrix} y(0) \\ y'(0) \end{pmatrix} + h \begin{pmatrix} y'(0) \\ -y(0) \end{pmatrix} \\
&= \begin{pmatrix} 2 \\ 1 \end{pmatrix} + h \begin{pmatrix} 1 \\ -2 \end{pmatrix} \\
&= \begin{pmatrix} 2 + h \\ 1 - 2h \end{pmatrix}.
\end{aligned}$$

We then proceed to see approximate $\mathbf{y}(0 + 2h)$.

$$\begin{aligned}
\mathbf{y}(2h) &= \mathbf{y}(h) + h\mathbf{F}(h, \mathbf{y}(h)) \\
&= \begin{pmatrix} 2 + h \\ 1 - 2h \end{pmatrix} + h \begin{pmatrix} y'(h) \\ -y(h) \end{pmatrix} \\
&= \begin{pmatrix} 2 + h \\ 1 - 2h \end{pmatrix} + h \begin{pmatrix} y_1(h) \\ -y_0(h) \end{pmatrix} \\
&= \begin{pmatrix} 2 + h \\ 1 - 2h \end{pmatrix} + h \begin{pmatrix} 1 - 2h \\ -2 - h \end{pmatrix} \\
&= \begin{pmatrix} 2 + 2h - 2h^2 \\ 1 - 4h - h^2 \end{pmatrix}.
\end{aligned}$$

So on and so forth. The process can go on indefinitely until we want to stop. If one would to plot the solution curve, we would get approximately the function $y(x) = 2\cos(x) + \sin(x)$ for small iterations. This agrees with our analytical solution.

Let's think about the code implementation. Recall that we have the following vectors

$$\mathbf{y} = \begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix}, \quad \text{and} \quad \mathbf{F}(x, \mathbf{y}) = \begin{pmatrix} y_1 \\ \vdots \\ y_{n-1} \\ f(x, \mathbf{y}) \end{pmatrix}.$$

If $\mathbf{y}$ is given, then we can simply think of constructing $\mathbf{F}(x, \mathbf{y})$ by popping the first element in $\mathbf{y}$ and appending $f(x, \mathbf{y})$. This is easy to code.

```
def F_vec(x: Real, y_vec: Vector, f: Function):
    F_res = y_vec.tolist()
    F_res.pop(0)
    F_res.append(f(x, y_vec))
    return np.array(F_res)
```

Remark. This is important. It is worth stressing that f must admit the special form

```
def f(x, y_vec):
    # some code
```

where it represents **exactly** $f = (x, \mathbf{y}) = (x, y, y', y'', ...)$. So the `y_vec` is viewed as an ordered list. As an example, if we want to define the system $y'' = -y$, this has the form $y'' = f(x, y, y')$ with $f = -y$. Accordingly, in code we define this as

```
def f(x, y_vec):
    return -y_vec[0]
```

A more complicated example: if we want to define the system $y''' = y' + 2y + x^2 + 3$, then we identify f with the rhs and so in code we have

```
def f(x, y_vec):
    return y_vec[1] + 2*y_vec[0] + x**2 + 3
```

In general, there is a correspondence

$$\texttt{y\_vec[n]} \longleftrightarrow y^{(n)} \longleftrightarrow y_n.$$

Great, we now have a description for $\mathbf{F}(x, \mathbf{y})$ in code. We proceed by making a simple observation: if we are solving for a first order ODE $y' = f(x, y)$, we supplement it with one initial condition to have a unique solution. If it is a second order ODE $y'' = f(x, y, y')$, then we would need two initial conditions. In general, an n-th order ODE needs n initial conditions to have a unique solution.

Now, these initial conditions together gives a starting point for our algorithm as can be seen in the previous example where we had $y(0) = 2$ and $y'(0) = 1$. In fact, they are the only thing we know at the start of the iteration. If we view the conditions $y(0) = 2, y'(0) = 1$ as an **ordered list** `[2., 1.]`, we can apply Euler's method to this row, and get a new row. This is the prediction of Euler's method at time $a + h$ (as per the example above). If we apply the Euler's method again, we get a new row which is the prediction at time $a + 2h$, etc. Our code will use this idea.

can be viewed as the column vector $\begin{pmatrix} 2 \\ 1 \end{pmatrix}$.

Algorithm. Euler's method.

```
def euler(
      init_conds: List[Real], a: Real, b: Real, N: int, f: Function):
    """Solves an ODE using the Euler method.

    Args:
       init_conds: List of initial value conditions.
       a: Initial value of x, also used as startpoint of interval.
       b: Endpoint of interval.
       N: Number of subintervals.
       f: The n-1 th order system if this is an n-th
            order ODE. It admits the form f(x, y_vec).
    """
    h = float(b-a) / N
    ys = np.zeros((N+1, len(init_conds)))
    ys[0] = init_conds
    xs = np.arange(start=a, stop=b+h, step=h)
```

```
    for i in range(N):
        xi, yi = a+i*h, ys[i]
        ys[i+1] = yi + h*F_vec(xi, yi, f)
    return xs, ys
```

Remark. Note how the `init_conds` array, which is the initial conditions **must be ordered**. The `F_vec` function pops and append `f(x, y_vec)` to `y_vec` whose initial row is exactly the `init_conds` array! Even the reverse array would give a whole different result.

TLDR: If we want to solve an ODE with initial conditions, say,

$$\mathbf{y}(a) = (1, 4, 6)$$

corresponding to initial conditions here on y, y' and y'', then we must put

```
init_conds = [1., 4., 6.].
```

Slogan. `init_conds` array must be ordered!

Let's do a rough[29] error analysis on the Euler's method. We know that the finite difference approximation gives an error of order $\mathcal{O}(h)$ due to machine precision. This was given by the Taylor's expansion

[29] *not so rigorous*

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

It is then trivial to see that

$$\mathbf{y}(x+h) = \mathbf{y}(x) + h\mathbf{y}'(x) + \mathcal{O}(h^2) = \mathbf{y}(x) + h\mathbf{F}(x, \mathbf{y}) + \mathcal{O}(h^2).$$

In other words, a single iteration of Euler's method gives an error of order $\mathcal{O}(h^2)$. If we do the iteration N times over some interval $[a, b]$, we thus accumulate an error of $\mathcal{O}(Nh^2)$. But since $N = (b-a)/h$, this error is in fact of order $\mathcal{O}(h)$. This means that our numerical approximation's final value $y(b)_{\text{approx}}$ via Euler's method and our theoretical final value $y(b)_{\text{exact}}$ has a difference

$$|y(b)_{\text{approx}} - y(b)_{\text{exact}}| = \mathcal{O}(h).$$

Of course as $h \to 0$, the numerical approximation becomes closer to the analytical value but since it approaches it in linear time, the approximation tends to the exact value very slowly. If we want to gain one digit, we have to decrease h by a factor of 10, and we do this by increasing the subdivision factor N by a factor of 10. This is not good and slow, so how do we mitigate this? The answer is to go to higher powers of h. This is the method of Runge-Kutta.

6.2 Runge-Kutta method

As usual in this course, the way we gain more precision is by looking at higher orders in the Taylor expansion.

6.2.1 Second-order Runge-Kutta

Let's take a look at just a single next order for now.

Observe that there is nothing stopping us from going to all the higher orders aside from computational power.

$$\mathbf{y}(x+h) = \mathbf{y}(x) + h\mathbf{y}'(x) + \frac{h^2}{2}\mathbf{y}''(x) + \mathcal{O}(h^3).$$

By using $\mathbf{y}(x) = \mathbf{F}(x, \mathbf{y})$, we further have

$$\mathbf{y}(x+h) = \mathbf{y}(x) + h\mathbf{F}(x, \mathbf{y}) + \frac{h^2}{2}\frac{d}{dx}\mathbf{F}(x, \mathbf{y}) + \mathcal{O}(h^3).$$

But real analysis tells us

$$\frac{d}{dx}\mathbf{F}(x, \mathbf{y}) = \frac{\mathbf{F}(x+h, \mathbf{y}(x+h)) - \mathbf{F}(x, \mathbf{y}(x))}{h} + \mathcal{O}(h),$$

so that

$$\frac{h^2}{2}\frac{d}{dx}\mathbf{F}(x, \mathbf{y}) = \frac{h}{2}\left[\mathbf{F}(x+h, \mathbf{y}(x+h)) - \mathbf{F}(x, \mathbf{y}(x))\right] + \mathcal{O}(h^3).$$

So we can substitute this value into our equation for $\mathbf{y}(x+h)$ above to have the equation

$$\begin{aligned}\mathbf{y}(x+h) &= \mathbf{y}(x) + h\mathbf{F}(x, \mathbf{y}) + \frac{h}{2}\left[\mathbf{F}(x+h, \mathbf{y}(x+h)) - \mathbf{F}(x, \mathbf{y}(x))\right] + \mathcal{O}(h^3). \\ &= \mathbf{y}(x) + \frac{h}{2}\mathbf{F}(x, \mathbf{y}) + \frac{h}{2}\mathbf{F}(x+h, \mathbf{y}(x+h)) + \mathcal{O}(h^3).\end{aligned}$$

Finally, we note that there is sort of a paradox here as we have $\mathbf{y}(x+h)$ in both sides of the equation. This can easily be solved by taking the linear approximation of $\mathbf{y}(x+h)$ in the rhs. That is, we take $\mathbf{y}(x+h) \approx \mathbf{y}(x) + h\mathbf{y}'(x) = \mathbf{y}(x) + h\mathbf{F}(x, \mathbf{y})$. So we now have, in approximation,

just like we did for the Euler's method as a whole

$$\mathbf{y}(x+h) \approx \mathbf{y}(x) + \frac{h}{2}\mathbf{F}(x, \mathbf{y}) + \frac{h}{2}\mathbf{F}(x+h, \mathbf{y} + h\mathbf{F}(x, \mathbf{y})) + \mathcal{O}(h^3).$$

As before in the Euler's method, we start at a point $x = a$ for which we know something about the function — it's initial conditions. Then look at $x = a + h$, then $x = a + 2h$, so on and so forth. We thus have a recursion.

Definition (*Runge-Kutta second-order method*)**.** Let $\mathbf{y}'(x) = \mathbf{F}(x, \mathbf{y})$ be an n-th order ODE with initial value conditions $\mathbf{y}(a) = \mathbf{c}$ for some $a \in \mathbf{R}$, $\mathbf{c} \in \mathbf{R}^n$. The solution to this initial value problem is given by the recursion

n-th order ODE in its n first-order ODE form.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{2}\mathbf{F}(x_n, \mathbf{y}_n) + \frac{h}{2}\mathbf{F}(x_{n+1}, \mathbf{y}_n + h\mathbf{F}(x_n, \mathbf{y}_n)).$$

where $\mathbf{y}_n = \mathbf{y}(x_n)$ and $x_n = a + nh$; and $h \in \mathbf{R}$ is a constant chosen to be small.

As you can guess by now, the code implementation of the second-order Runge-Kutta method is as simple as tweaking a line of code in the Euler method with keeping everything else, such as `F_vec`, as it is. We will however add another line of code to make things more readable.

Algorithm. Second-order Runge-Kutta method.

```
def runge_kutta_2nd(
      init_conds: List[Real], a: Real, b: Real, N: int, f: Function):
```

```
    """Solves an ODE using the second-order Runge-Kutta method.

    Args:
       init_conds: List of initial value conditions.
       a: Initial value of x, also used as startpoint of interval.
       b: Endpoint of interval.
       N: Number of subintervals.
       f: The n-1 th order system if this is an n-th
            order ODE. It admits the form f(x, y_vec).
    """
    h = float(b-a) / N
    ys = np.zeros((N+1, len(init_conds)))
    ys[0] = init_conds
    xs = np.arange(start=a, stop=b+h, step=h)

    for i in range(N):
        xi, yi = a+i*h, ys[i]
        F_vec_i = F_vec(xi, yi, f)
        ys[i+1] = (yi
                   + h/2*F_vec_i
                   + h/2*F_vec(xi+h, yi + h*F_vec_i, f))
    return xs, ys
```

By the same argument in our error analysis for Euler's method, the second-order Runge-Kutta method has an error of order $\mathcal{O}(h^2)$, which is better than Euler's method. This is also the reason why we called it *second-order* Runge-Kutta corresponding to the second power in h.

6.2.2 Fourth-order Runge-Kutta

Like we mentioned earlier, we can do better simply by looking at even higher powers of h, and this would give us an error of order $\mathcal{O}(h^3)$, $\mathcal{O}(h^4)$, so on and so forth. But as can be seen in the code, the amount of computation increases as well, so there is a compromise. It turns out that an *optimal* power of h to look at is the fourth power. This gives the fourth-order Runge-Kutta which is, in Ben's words, "the most used method to solve ODEs" due to its simplicity and efficiency. Define the following variables

$$\begin{aligned}
\mathbf{K}_0 &= h\mathbf{F}(x, \mathbf{y}), \\
\mathbf{K}_1 &= h\mathbf{F}\left(x + \frac{h}{2}, \mathbf{y} + \frac{\mathbf{K}_0}{2}\right), \\
\mathbf{K}_2 &= h\mathbf{F}\left(x + \frac{h}{2}, \mathbf{y} + \frac{\mathbf{K}_1}{2}\right), \\
\mathbf{K}_3 &= h\mathbf{F}(x + h, \mathbf{y} + \mathbf{K}_2).
\end{aligned}$$

Using this notation, we can write

$$\mathbf{y}(x + h) = \mathbf{y}(x) + \frac{1}{6}\left(\mathbf{K}_0 + 2\mathbf{K}_1 + 2\mathbf{K}_2 + \mathbf{K}_3\right) + \mathcal{O}(h^5).$$

By considering the fact that we are doing this over N subdivisions of some interval $[a, b]$, we thus have the order of error to be $\mathcal{O}(h^4)$ as desired.

Definition (*Runge-Kutta fourth-order method*)**.** Let $\mathbf{y}'(x) = \mathbf{F}(x, \mathbf{y})$ be an n-th order ODE with initial value conditions $\mathbf{y}(a) = \mathbf{c}$ for some $a \in \mathbf{R}$, $\mathbf{c} \in \mathbf{R}^n$. The solution to this initial value problem is given by the recursion

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{1}{6}\left(\mathbf{K}_0 + 2\mathbf{K}_1 + 2\mathbf{K}_2 + \mathbf{K}_3\right),$$

where $\mathbf{y}_n = \mathbf{y}(x_n)$ with $x_n = a + nh$; and $h \in \mathbf{R}$ is a constant chosen to be small.

n-th order ODE in its n first-order ODE form.

Algorithm. Fourth-order Runge-Kutta method.

```
def runge_kutta_4th(
    init_conds: List[Real], a: Real, b: Real, N: int, f: Function):
  """Solves an ODE using the fourth-order Runge-Kutta method.

  Args:
     init_conds: List of initial value conditions.
     a: Initial value of x, also used as startpoint of interval.
     b: Endpoint of interval.
     N: Number of subintervals.
     f: The n-1 th order system if this is an n-th
          order ODE. It admits the form f(x, y_vec).
  """
  h = float(b-a) / N
  ys = np.zeros((N+1, len(init_conds)))
  ys[0] = init_conds
  xs = np.arange(start=a, stop=b+h, step=h)

  for i in range(N):
      xi, yi = a+i*h, ys[i]
      K0 = h * F_vec(xi, yi, f)
      K1 = h * F_vec(xi+h/2, yi+K0/2, f)
      K2 = h * F_vec(xi+h/2, yi+K1/2, f)
      K3 = h * F_vec(xi+h, yi+K2, f)
      ys[i+1] = yi + 1./6*(K0 + 2*K1 + 2*K2 + K3)
  return xs, ys
```

6.3 Boundary value problem

Observe that what we have done so far are initial value problems where we have initial conditions $\mathbf{y}(a) = \mathbf{c}$ for some $\mathbf{c} \in \mathbf{R}^n$. That is, we know what the values of $y, y', y'', \ldots$ are at some fixed $x = a$ initially.

However, the motion of a pendulum[30] is not the only equation in the world. For example, suppose we ask the following question: if I drop a swing whose initial height at time zero is 10 metres and its height at time π is 7.39 metres, whose motion is governed by some differential equation $y''(x) = f(x, y, y')$, then how do we solve this? This is now something that we call a **boundary value problem**, with initial boundary conditions $y(0) = 10$ and $y(\pi) = 7.39$. The boundary value problem is ubiquitous in math and

[30] *This is a reference to the famous initial value problem we've seen earlier* $y'' = -y$.

physics, especially in the context of partial differential equations. It is thus worth to try to solve them first when they are simply an ordinary differential equation.

6.3.1 Shooting method

In this course, we will only solve the easiest yet nontrivial case of all boundary value problems: second-order ODE with two boundary conditions. Generally, we can write this as finding the solution to

$$y'' = f(x, y, y'), \quad y(a) = \alpha, \quad y(b) = \beta, \tag{6.1}$$

Note that as before, f can be as nonlinear as you'd like.

for some $a, b, \alpha, \beta \in \mathbf{R}$.

One way to solve this problem is to convert it into an initial value problem which we now know how to solve, for example, by using Runge-Kutta. So we consider the initial value problem

$$y'' = f(x, y, y'), \quad y(a) = \alpha, \quad y'(a) = u, \tag{6.2}$$

Notice the only thing that has changed is the right-most term.

where $u \in \mathbf{R}$ and solve it within the interval $[a, b]$. The solution to this problem is a curve $y(x)$, whose value at $x = b$ is known. If $y(b) = \beta$, we are done. Otherwise, change u to, say, $u^* \in \mathbf{R}$ and try again. This time we would get another solution $y^*(x)$ whose value at $b = \beta$ is again known. If $y^*(b) = \beta$, we are done, otherwise keep trying.

Trial and error will get us the solution eventually but this will take a lot of time. Since we have Naruto and Netflix episodes to finish, we have to think differently. Let us denote the set of all possible solution curves to (6.2) by

$$\mathcal{C} = \{y_u(x) \mid y_u(x) \text{ solves (6.2) for some } u \in \mathbf{R}\}.$$

Note that a and α are fixed. We let only u to vary.

Remark. One can think of this set as trajectories that goes through the point $x = a$ which obeys the initial value problem, and there are infinitely many of them. Since a solution to a second-order ODE with two initial conditions on y and y' is necessarily unique, there is exactly one solution for each $u \in \mathbf{R}$. So we have to find the unique $u \in \mathbf{R}$ such that $y_u \in \mathcal{C}$ and $y_u(b) = \beta$. In other words, we have to find the unique trajectory y_u which solves $y'' = f(x, y, y')$ that goes through both α and β. This is where the name *shooting* method comes from.

Note that if it so happens that $y_u(b) = \beta$, it is the unique such y_u that satisfies this condition.

Then construct a map $\phi : \mathbf{R} \to \mathbf{R}$ defined by $u \mapsto y_u(b)$ where $y_u \in \mathcal{C}$. Our problem now translates to finding (the unique) $v \in \mathbf{R}$ such that $\phi(v) = y_v(b) = \beta$. Such a v is thus a root of the equation $\Phi(u) = 0$, where $\Phi(u) := \phi(u) - \beta$; and we can find this root by using methods that we already know such as the Newton, secant and bisection method. Since the Newton method requires knowledge of the function's derivative and the bisection method is so slow, we will settle for the secant method. So in summary, the steps involved are the following:

1. Solve the equation $\Phi(u) = 0$ using the secant method.
2. Say, we finally get $u^* \in \mathbf{R}$ as a solution.
3. Apply an ODE method to solve (6.2) with $u = u^*$.
4. We get our $y(x)$ which solves (6.1), yeay!

which involves solving an ODE **a lot of times**.

The algorithm should then be straightforward from here.

Algorithm. Shooting method.

```
def shooting(a: Real,
             b: Real,
             alpha_cond: Real,
             beta_cond: Real,
             f: Function):
    """Solves the boundary value problem for second-order ODE.

    Args:
       a: Initial value of x, also used as startpoint of interval.
       b: Endpoint of interval.
       alpha_cond: Initial condition so that y(a) = alpha_cond.
       beta_cond: Initial condition so that y(b) = beta_cond.
       f: The n-1 th order system if this is an n-th
            order ODE. It admits the form f(x, y_vec).
    """
    _partition_len = 2000
    def Phi(u: Indeterminate):
        """The function y(b) - beta_cond."""
        _, ys = runge_kutta_4th(init_conds=[alpha_cond, u],
                                a=a, b=b, N=_partition_len, f=f)
        last_ys = ys[-1]
        last_y = last_ys[0]
        return last_y - beta_cond

    root = secant(Phi, a, b)

    xs, ys = runge_kutta_4th(init_conds=[alpha_cond, root],
                             a=a, b=b, N=_partition_len, f=f)
    y_vals = ys[:,0]
    return xs, y_vals
```

Note how we have used the `secant` and `runge_kutta_4th` methods, both of which are algorithms we have implemented before.

Remark. A final remark on ODE. Note that the numerical methods we have developed for solving initial and boundary value problems all work regardless of what the ODE looks like. We don't care whether it's linear or not; and we certainly don't care whether we know how to solve it analytically or not.

7 Stochastic differential equation

Stochastic just means random, and stochastic differential equations (SDE) are just differential equations with some random noise. Stochastic differential equations are prominent in a lot of subjects such as biology, physics and finance. We will discuss some specific examples in physics and finance.

Example. In physics, a particle going around in a gas or in a fluid will be affected by the collisions with molecules in that fluid. These collisions can be seen as random as they happen time to time in arbitrary directions.

In order to represent the motion of this particle with this additional random noise, we may add a random term to the Newton's 2nd law $m\ddot{x} = F(t)$ which is a standard ODE.

Example. In finance, almost every risky asset (think stocks, forex, crypto, put and call options) is seen as a random variable even in its simplest one-period binomial model form. One way to model this is to think about "*who trades what at when*". So the random noise here is in fact governed by traders and investors.

Of course, this is not an exact representation as there are only finitely many ways of decision making in buying and selling assets. But since modelling every single decision would be messy and complicated, we can treat it as random.

7.1 A quick hand-waving intro

A proper mathematical formulation of SDEs requires some prerequisites such as probability theory. We will therefore take a hand-waving approach so that the things we talk about are still precise and make sense to those not previously exposed to the theory of stochastic calculus.

Definition (*Stochastic process*)**.** A **stochastic process** is a sequence $(X_i)_{i\in I}$ of random variables.

Note that I is an arbitrary index set here — possible has no ordering or infinite

Definition (*Wiener process*)**.** A (standard) **Wiener process** is a stochastic process $(W_t)_{t\geqslant 0}$ such that

(1) $W_0 = 0$.

(2) W_t has independent increments.

(3) W_t is continuous in t.

(4) Let $\Delta W_t = W_{t+\Delta t} - W_t$. Then $\Delta W_t \sim \mathcal{N}(0, \Delta t)$.

We will only care about (1) and (4) so we will not spend time explaining what does *independent increments* or being continuous as a discrete process means. What is crucial here is to remember that $\Delta W_t \sim \mathcal{N}(0, \Delta t)$. That is, the step increment on W_t is i.i.d normal with expectation zero and variance Δt. We now have sufficient vocabulary to describe a stochastic differential equation.

Definition (*Stochastic differential equation*)**.** A **stochastic differential equation** is an equation of the form

$$dX_t = a(X_t)\,dt + b(X_t)\,dW_t,$$

where a, b are given functions and W_t is a Wiener process.

7.2 Euler-Maruyama method

A numerical approximation of the solution of SDEs is given by the so-called Euler-Maruyama method. This method can be formulated by mimicking what we did for ODEs before this.

Theorem (*Euler-Maruyama method*)**.** Consider the stochastic differential equation

$$dX_t = a(X_t)\,dt + b(X_t)\,dW_t,$$

on the time interval $\mathcal{T} = [0, T]$ with initial value condition $X_0 = x_0$. Choose $N \in \mathbf{N}$ to partition the interval $\mathcal{T}$ into equally spaced intervals of length $\Delta t = T/N$. Then the solution to this initial value problem is given by the recursion

$$x_{n+1} = x_n + a(x_n)\,\Delta t + b(x_n)\,\Delta W_n$$

where $x_n = x(t_n)$ with $t_n = t_0 + n\Delta t$, and $\Delta W_n = W_{t_{n+1}} - W_{t_n}$.

and again, we emphasize that $\Delta W_n \sim \mathcal{N}(0, \Delta t)$.

For simplicity, let us look at the special case of $a(X_t) = 0$ (also known as the zero-drift case) and $b(X_t) = \sigma$ for some constant $\sigma \in \mathbf{R}$ which gives us the so-called **Brownian motion**. In this case, we have the following SDE

$$dX_t = \sigma\,dW_t,$$

whose approximate solution, via the Euler-Maruyama method, is given by the recursion

$$x_{n+1} = x_n + \sigma\,\Delta W_n.$$

The code for solving Brownian motion should then be obvious. However, we want to let things be more general. In the Euler-Maruyama method above, we talk about a time interval $[0, T]$. There's nothing stopping us from considering a more general interval $[\tau, T]$ and partition this interval into equally spaced intervals of length $(T - \tau)/N$. Finally, we want to remind you (for the third time) that $\Delta W_n \sim \mathcal{N}(0, \Delta t)$. In particular we have $\operatorname{var}(\Delta W_n) = \Delta t$ and so the standard deviation is given by $\sqrt{\operatorname{var}(\Delta W_n)} = \sqrt{\Delta t}$.

Algorithm. Euler-Maruyama method for Brownian motion.

```
def brownian(x_init: Real,
             t_init: Real,
             t_end: Real,
             N: int,
             sigma: Real):
    """
    Solves an SDE using the Euler-Maruyama method.

    Args:
        x_init: Initial starting point of stochastic process.
        t_init: Initial time of evolution, expect > 0.
        t_end: Total time of evolution, expect > 0.
        N: Number of subintervals to partition [t_init, t_end].
        sigma: A real constant value.
    """
```

```
    # Initialize sequence of time, ts.
    dt = (t_end-t_init) / float(N)
    ts = np.arange(start=t_init, stop=t_end, step=dt)

    # Initialize sequence of solution, xs.
    xs = np.zeros(N)
    xs[0] = x_init

    # Solve using Euler-Maruyama method.
    for i in range(1, N):
        dW = np.random.normal(loc=0., scale=np.sqrt(dt))
        xs[i] = xs[i-1] + sigma*dW

    return ts, xs
```

The code for the general Euler-Maruyama method should then be obvious and is left as an exercise for the reader.

Remark. Notice that we feed `np.random.normal()` with the square root of `dt` instead of the variance `dt` itself. This is because the makers of `numpy` made the normal distribution implementation in `numpy` to accept the standard deviation instead of variance. So nothing funny is really happening here, it's just a choice made by the developers.

8 Partial differential equations

Partial differential equations (PDE) are naturally much much harder than ordinary differential equations. We will not go too much into the theory, but more into just solving it numerically. We will look at the **relaxation method** to solve the Laplace equation in two independent variables

also known as the steady-state heat equation.

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0,$$

where $f : U \to \mathbf{R}$ is a twice-differentiable scalar function, and $U \subseteq \mathbf{R}^2$ is an open subset (w.r.t the usual topology). We will typically choose U to be a box, say, $[a, b] \times [c, d]$ for some $a, b, c, d \in \mathbf{R}$. Using the standard finite difference approximation for second derivatives, we have the system of equations

$$\frac{\partial^2 f}{\partial x^2} = \frac{f(x+h, y) + f(x-h, y) - 2f(x, y)}{h^2} + \mathcal{O}(h^2),$$

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

Evaluating $\nabla^2 f$ we have

$$\nabla^2 f = \frac{1}{h^2} \left[f(x+h, y) + f(x-h, y) + f(x, y+h) + f(x, y-h) - 4f(x, y) \right] + \mathcal{O}(h^2).$$

So an approximation to the solution of the Laplace equation $\nabla^2 f = 0$ is given by

$$f(x, y) \approx \frac{1}{4} \left[f(x+h, y) + f(x-h, y) + f(x, y+h) + f(x, y-h) \right].$$

Definition (*Discrete neighbourhood*)**.** Let $f : U \to \mathbf{R}$ be a twice-differentiable function, where $U \subseteq \mathbf{R}^2$ is open; and let $h > 0$. The **discrete h-neighbourhood** of a point $f(x_0, y_0) \in f(U)$, when it exists, is the four-point set

$$\mathcal{F}_{(x_0, y_0)} = \{ f(x_0 \pm h, y_0), f(x_0, y_0 \pm h) \}.$$

Definition (*Discrete average*)**.** Let $f : U \to \mathbf{R}$ be a twice-differentiable function, where $U \subseteq \mathbf{R}^2$ is open. Let $\mathcal{F}$ be the discrete h-neighbourhood of $f(x_0, y_0) \in f(U)$. The **h-average** of $f(x_0, y_0)$ is defined to be

We will say that (x_0, y_0) has a discrete h-neighbourhood if it's corresponding $f(x_0, y_0)$ point has a discrete h-neighbourhood. Likewise for the h-average.

$$A_{(x_0, y_0)} = \frac{1}{4} \sum_{e \in \mathcal{F}} e.$$

So a solution f of the Laplace equation must satisfy the condition that $f(x_0, y_0)$ is equal to its h-average for each $(x_0, y_0) \in U$. This induces a natural algorithm to solve the problem called the *relaxation method*.

8.1 Relaxation method

If you are familiar with applying the box blur algorithm on images, then this algorithm is similar except that we are now considering a "diamond" neighbourhood and disregarding boundary points. For simplicity, consider the box $U = [0, a] \times [0, b]$ for some $a, b \in \mathbf{R}$.

What we do now is discretise points of U (and hence, the image $f(U)$) and view it as a lattice which we will call a **configuration space** or just configuration.

Example. For example, we can discretise U by using a sequence of integer points $P \times Q \subseteq U \cap \mathbf{Z}^2$. A concrete example is the following: let $a = b = 2$ in $U = [0, a] \times [0, b]$. Then $U \cap \mathbf{Z}^2 = \{0, 1, 2\} \times \{0, 1, 2\} =: P \times Q$. So we have a lattice of integers that looks like Figure 3. We then care about the image $f(P \times Q)$ which can also now be viewed as a lattice — simply by assigning $f(x, y) \to (x, y)$ for each $(x, y) \in P \times Q$.

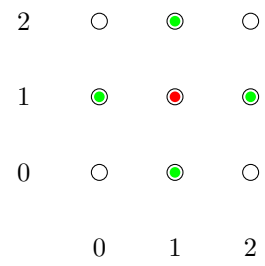

Figure 3. Lattice of integers

In general, we would want to discretise U by a sequence of rational numbers equally spaced by a step $h \in \mathbb{Q}$. We can then scan through this lattice and replace each point $f(x, y)$ by its h-average. This is the so-called **relaxation method**.

Example. Referring to Figure 3 and taking $h = 1$, then the relaxation method applied to this configuration means replacing f(red point) with the average of all f(green point).

Immediately from Figure 3, we can see that there is a caveat to this algorithm. The boundary points ∂U does not have a (complete) discrete 1-neighbourhood. As an example, consider the point $(0, 2) \in \partial U$. Then with $h = 1$, the point $(0 - h, 2) = (-1, 2) \notin U$. Consequently, it is not well-defined to talk about $f(-1, 2)$ as f is defined only on U. We solve this by not applying the algorithm at all to these boundary points. So for the lattice in Figure 3, we actually do the algorithm on the single red point and we stop.

The point of the algorithm is that we do it iteratively and after some number of iterations, the original configuration eventually reaches a stable configuration i.e. it converges. This explains the name *relaxation*. If we assume that we have an initial configuration, that is, a discretised version of U, then the code implementation should be immediate.

We will not discuss the convergence of this method, but what we can say is that we are guaranteed that we have a solution every time for the Laplace equation. This method can be applied to various other problems. However, convergence is not necessarily guaranteed for all these problems. Thankfully, the method works really well for the Laplace equation.

Algorithm. Relaxation method Python implementation.

```
def relaxation(init_config: Matrix):
    """Solves the Laplace equation in two variables using
    the relaxation method.

    Args:
        init_config: The initial discretised configuration.
    """
    res = init_config.copy()
    sizex, sizey = res.shape
    for x in range(1, sizex-1):
        for y in range(1, sizey-1):
            res[x, y] = (init_config[x+1, y]
                         + init_config[x-1, y]
                         + init_config[x, y+1]
                         + init_config[x, y-1])
            res[x, y] /= 4
    return res
```

9 Eigenvalue problem

Let's turn our attention to some numerical linear algebra. In particular, we will look at computing eigenvalues of a linear map. Before anything, let us first agree on some conventions.

Conventions. We will only talk about matrices, but always keeping in mind that they represent a linear map $T : V \to V$ of finite-dimensional vector spaces over $\mathbf{R}$. In fact, we will only talk about the special case $V = \mathbf{R}^d$. Suppose $\dim V = n$, then these matrices would be square $n \times n$ matrices. Moreover, we will assume that A is a **real symmetric matrix** unless otherwise stated. If you remember from linear algebra, this ensures that the eigenvalues of A are real. If A is a matrix, we will always write its entries as (a_{ij}), that is the i-th row and j-th column element of A is a_{ij}. In general, the lowercased version of the letter used to write the matrix represents the entries of the matrix. An immediate example is $A = (a_{ij})$ above.

fancier way: T is an endomorphism of V.

Definition (*Eigenstuffs*)**.** Let A be any square matrix. A scalar $\lambda \in \mathbf{R}$ is said to be an **eigenvalue** of A if there is a nonzero (column) vector v such that $Av = \lambda v$. The corresponding v is then called an **eigenvector** of A.

N.B. The eigenvector, when it exists, is nonzero! This is important.

Observe that the equation $Av = \lambda v$ is equivalent to the equation $(A - \lambda I)v = 0$, where I is the identity matrix. Further observe that the map $A - \lambda I$ is not *injective*[31] as it maps both v and 0 to 0, but $v \neq 0$. By the well-known fact of linear algebra, the matrix $A - \lambda I$ is therefore singular and has zero determinant. The determinant $f_A(x) := \det(A - \lambda I)$ is called the **characteristic polynomial of** A.

[31] *by this, we meant the linear operator that it represents is injective.*

Theorem. Let A be a square matrix and $\lambda \in \mathbf{R}$. Then λ is an eigenvalue of A if and only if $\det(A - \lambda I) = 0$.

9.1 Naive method

If we try to solve the equation $\det(A - \lambda I) = 0$, then we're back to solving roots of polynomials given that we know how to calculate determinants. Let's be lazy and just use the `linalg.det` method from `numpy` to compute determinants and the `linalg.identity` method to cook up an identity matrix. We thus have an obvious algorithm for the characteristic polynomial.

Algorithm. The characteristic polynomial $\det(A - xI)$.

We replace the eigenvalue λ with x now since it is more readable in code.

```
def char_poly(A: Matrix):
    def char_poly_x(x):
        eigenvalue_matrix = A - x*np.identity(len(A))
        return np.linalg.det(eigenvalue_matrix)
    return char_poly_x
```

This gives us a Python function in the variable x for which we can compute its roots using previous methods. But here comes a problem. Before this[32], we usually have a clear interval/range to hunt the roots of the function. We now don't have that luxury anymore! Let us try to solve for the roots of the characteristic polynomial using the Newton method.

[32] *for example, in implementing the shooting method*

Example (Naive implementation and root guessing problem)**.** Consider the real symmetric 3×3 matrix

$$A = \begin{pmatrix} 1 & -1 & 0 \\ -1 & 4 & -2 \\ 0 & -2 & 2 \end{pmatrix}.$$

The characteristic polynomial, $f_A(x)$ of A is given by $\det(A - xI)$. In code, this matrix and its characteristic polynomial is the following.

```
A = np.array([[1, -1, 0], [-1, 4, -2], [0, -2, 2]])
char_poly_A = char_poly(A)
```

If we want to find the roots of f_A, we are forced to first estimate (or even guess) the position of the roots. This can be done, say, by plotting. It turns out that the roots of f_A is nearby $x = 0, 1, 5$. So we apply a root-hunting algorithm to find the approximate roots. Because we feel super lazy today, we will use the Newton's method, `newton` from the `scipy.optimize` module.

```
import scipy as sc
roots = []
for x in [0, 1, 5]:
    root = sc.optimize.newton(char_poly_A, x)
    roots.append(root)
```

The roots are then found in `roots`.

So our naive approach was straightforward just applying what we've known so far, at the expense of having to inspect the neighbourhood of the roots. We have even more subtle problems.

Example (Degenerate eigenvalue problem)**.** Consider the polynomial

$$\lambda^2 - 2\lambda + 1 = (1 - \lambda)^2.$$

This is the characteristic polynomial of the 2×2 matrix $I - \lambda I$. Observe that this polynomial has a *double root* $\lambda = 1$ and our naive method could not simply "understand" this.

Example (Super huge matrix problem)**.** In practice, we usually care about very huge matrices. For example, consider a machine learning problem in the subdomain of natural language processing (NLP). It is possible that we will be dealing with a $\sim 10^4 \times 10^4$ matrix when processing, say, a collection of Wikipedia pages. Would you want to inspect the position of all roots? By the Fundamental Theorem of Algebra, there are exactly 10^3 complex roots and at most 10^3 real roots to look at!

9.2 Jacobi method

Let us first recall the definition of two matrices being similar and diagonalizable.

Definition (*Similar matrix*)**.** Let A be a square matrix. We say B is **similar** to A if there exists an invertible matrix P such that $B = P^{-1}AP$.

Definition (*Diagonalizable*)**.** Let A be a square matrix. We say A is **diagonalizable** if there exists an invertible matrix P such that $D = P^{-1}AP$ is diagonal.

Note that the matrix P in both definitions is nothing else but the change-of-basis matrix. Now recall (or make the simple observation) that the eigenvalues of a diagonal matrix is simply its diagonal entries. So if we are dealing with real symmetric matrices, this is good because we have the following theorem from linear algebra.

Theorem (*Diagonalizability for real symmetric matrices*)**.** Let A be a real symmetric matrix. Then there exists an orthogonal matrix P such that $D = P^{-1}AP$ is diagonal.

Since the eigenvalues of a matrix A is invariant under a change of basis[33], we can diagonalize A to have a diagonal matrix D and read off the diagonal entries of D. The Jacobi method we are going to look at is essentially diagonalizing A in the special way of using the orthogonal Jacobi rotation matrices $Q_{k\ell}$. The above theorem guarantees the existence of an orthogonal matrix P which diagonalizes A. The Jacobi method forces a sequence of Jacobi rotation matrix to behave and act like P.

[33] *easy to proof: If $Ax = \lambda x$, then choose a basis so that $x = Py$ and observe that $By = \lambda y$ where $B = P^{-1}AP$.*

Definition (*Jacobi rotation matrix*)**.** The **$k\ell$-Jacobi rotation matrix** is the square orthogonal matrix

$$Q_{k\ell}(\theta) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & 0 & \\ & & \cos\theta & \cdots & \sin\theta & & \\ & & \vdots & 1 & \vdots & & \\ & & -\sin\theta & \cdots & \cos\theta & & \\ & 0 & & & & \ddots & \\ & & & & & & 1 \end{bmatrix},$$

where $\theta \in \mathbf{R}$. The kk and $\ell\ell$ entries are $\cos\theta$, the $k\ell$ entry is $\sin\theta$ and the ℓk entry is $-\sin\theta$. This matrix describes a rotation of θ in the $k\ell$-plane. For simplicity, we will usually drop the dependence on θ when writing and just write $Q_{k\ell}$. Also, we may refer to it as the Jacobi rotation matrix without the $k\ell$.

Note that the Jacobi rotation matrix, unlike the usual rotation matrix, doesn't always rotate the plane in one direction only. But we don't really care about this.

Example. The standard (clockwise) rotation matrix

$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

in $\mathbf{R}^2$, gives a clockwise rotation by θ in the xy-plane. This is the Jacobi rotation matrix Q_{12} in $\mathbf{R}^2$.

Note that our definition for a Jacobi rotation matrix depends on the vector space V we are acting on. So Q_{12} in $\mathbf{R}^2$ need not be the same as Q_{12} in $\mathbf{R}^3$ (see next example). This make sense as otherwise our definition is not a very good one. For example, even in 4-dimensional, we would still want to rotate in the xy-plane.

Example. The 3×3 (clockwise) Jacobi rotation matrix in the xy plane

$$\begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

is the Jacobi rotation matrix Q_{12} in $\mathbf{R}^3$.

Slogan. Jacobi rotation matrix is an identity matrix except possibly at four symmetrical entries, the diagonal having $\cos\theta$ terms and the off-diagonal having $\sin\theta$ terms.

Using the Jacobi rotation matrices to do a similarity transformation is okay because of the following proposition.

Proposition. Let A be a real symmetric $n \times n$ matrix and let $Q = Q_{k\ell}(\theta)$ be the $k\ell$-Jacobi rotation matrix, with $1 \leqslant k, \ell \leqslant n$. Then the matrix $B = Q^\top AQ$ is similar to A and also real symmetric.

The reason why they are useful for diagonalizing is the following. For suitable $\theta \in \mathbf{R}$, the Jacobi rotation matrix $Q_{k\ell}(\theta)$ kills the off-diagonal[34] entries $a_{k\ell}$ and $a_{\ell k}$. So if we apply the Jacobi rotation matrices to all off-diagonal entries of A iteratively, we would eventually end up with a diagonalized version of A. This whole process is called the Jacobi transformation.

[34] *non-diagonal entries*

Definition (*Jacobi transformation*)**.** Let A be a real symmetric matrix and write the $k\ell$-Jacobi rotation matrix as $Q = Q_{k\ell}(\theta)$. The matrix $B = (b_{ij}) = Q^\top AQ$ is said to be a **$k\ell$-Jacobi transformation of A** if $\theta \in \mathbf{R}$ is such that $b_{k\ell} = 0$. The corresponding θ is called the **$k\ell$-annihilator of A**.

N.B. This is not a standard definition, even in the module!

The problem now is, given an entry $a_{k\ell}$ of A, what is the $k\ell$-annihilator θ? Is there a precise formula? Since we don't want to get involved with inverse cosines and sines, it is sufficient to look at $\cos\theta$ only[35]. For simplicity, write $Q = Q_{k\ell}$ for the $k\ell$-Jacobi rotation matrix. Now let $B = (b_{ij})$ be the matrix $B = Q^\top AQ$ with $Q = (q_{ij})$. Then let's look at $b_{k\ell}$.

[35] *for the same θ, we can get $\sin\theta$ from $\cos\theta$ by the Pythagorean identity. In fact, we will get $\sin\theta$ in a more clever way.*

$$\begin{aligned} b_{k\ell} = \sum_{i,j} q_{ik}a_{ij}q_{j\ell} &= \sum_j (q_{kk}a_{kj}q_{j\ell} + q_{\ell k}a_{\ell j}q_{j\ell}) \\ &= \sum_j (\cos\theta a_{kj}q_{j\ell} - \sin\theta)a_{\ell j}q_{j\ell}. \end{aligned}$$

Notice that we have q_{ik} instead of q_{ki} in the first sum because we transposed the matrix Q!

The second equality comes from the fact there are only two places over i for which the terms are nonzero, namely at $i = k$ and $i = \ell$. Similary over j, there are only two places for which the terms are nonzero namely at $j = k$ and $j = \ell$. So, we further have

$$\begin{aligned} b_{k\ell} &= (a_{kk}q_{k\ell} + a_{k\ell}q_{\ell\ell})\cos\theta - (a_{\ell k}q_{k\ell} - a_{\ell\ell}q_{\ell\ell})\sin\theta \\ &= a_{kk}\sin\theta\cos\theta + a_{k\ell}\cos^2\theta - a_{\ell k}\sin^2\theta - a_{\ell\ell}\sin\theta\cos\theta. \end{aligned}$$

Since A is symmetric, then $a_{k\ell} = a_{\ell k}$ and writing $c = \cos\theta$ and $s = \sin\theta$, we have the equation

$$b_{k\ell} = (c^2 - s^2)a_{k\ell} + cs(a_{kk} - a_{\ell\ell}). \tag{9.1}$$

By using the Pythagorean identity $c^2 + s^2 = 1$, we can look only[36] at cosine terms in $b_{k\ell}$:

$$b_{k\ell} = (2c^2 - 1)a_{k\ell} + c\sqrt{1 - c^2}(a_{kk} - a_{\ell\ell}).$$

[36] *Note that there is nothing special about $\cos\theta$, we could have just opted to look at $\sin\theta$.*

We know that you've been too caught up with the algebra so let us remind you what we're trying to do here. Our goal is to find the $k\ell$-annihilator $\theta \in \mathbf{R}$. If we choose a suitable θ so that $a_{k\ell}$ is annihilated, then this means that the resulting product $b_{k\ell}$ must be zero. It suffices to choose a suitable $\cos\theta$, so, our next step is to solve the equation $b_{k\ell} = 0$ for c,

which is dependent on θ. We have

$$b_{k\ell} = 0 \implies (2c^2 - 1)a_{k\ell} = c\sqrt{1-c^2}(a_{kk} - a_{\ell\ell}).$$

Squaring[37] both sides of the equation, we get a quadratic equation in c^2 (not c).

$$(2c^2 - 1)^2 a_{k\ell}^2 = c^2(1-c^2)(a_{\ell\ell} - a_{kk})^2. \tag{9.2}$$

[37] *There is a subtle problem with squaring. We have lost information about the sign of* $\cos(x)$. *This gives rise to a sign ambiguity later on which we will tackle.*

This can be solved trivially using the standard formula for roots of quadratic equation. We get

$$c^2 = \frac{1}{2}\left(1 + \sqrt{\frac{1}{1+4\tau^2}}\right), \quad \text{where} \quad \tau = \frac{a_{\ell k}}{a_{kk} - a_{\ell\ell}}. \tag{9.3}$$

So we now know that θ is a $k\ell$-annihilator if it satisfies equation (9.3). For a complete algorithm, we would want to get a full description of the matrix B after the $k\ell$-Jacobi transformation of A. This demands for knowledge of the value of $s = \sin\theta$ as well. So how do we do this?

It is tempting to say that $s = \sqrt{1-c^2}$, and this is derived from the Pythagorean identity $s^2 = 1 - c^2$. While this is true, we have actually just opened up a can of worms by getting s this way! Recall that we lose information of sign when we square both sides of $b_{k\ell} = 0$ in (9.2). Since signs are important in determining relationship between sin and cos, doing this does not necessarily gives us the right answer. What we can instead do is to look back at equation (9.1), write $c^2 - s^2 = 2c^2 - 1$, leave the second c term untouched and isolate s on one side. All of this gives us

$$s = \frac{\tau}{c}(1 - 2c^2), \quad \text{where} \quad \tau = \frac{a_{\ell k}}{a_{kk} - a_{\ell\ell}}.$$

observe that this τ and the previous τ coincides.

Proposition (*Off-diagonal annihilator*)**.** Let $A = (a_{ij})$ be a real symmetric matrix. The number $\theta \in \mathbf{R}$ is an xy-annihilator of A if and only if

$$c^2(\theta) = \frac{1}{2}\left(1 + \sqrt{\frac{1}{1+4\tau^2}}\right), \quad s^2(\theta) = \frac{\tau}{c}(1 - 2c^2),$$

where $c = \cos\theta$, $s = \sin\theta$ and

$$\tau = \frac{a_{xy}}{a_{yy} - a_{xx}}.$$

This is obvious but it suffices for θ to satisfy the first equation since the second equation depends on the first equation.

The algorithm for this process is also immediate.

Algorithm. The function to get cosine and sine terms so that a_{xy} is annihilated.

```
def get_cs(x: int, y: int, A: Matrix):
    """Returns the cosine and sine of the angle which annihilates
    the (x,y) entry of A under the Jacobi transformation.

    Args:
        x: The row of the entry (x,y).
        y: The column of the entry (x,y).
        A: The matrix to be diagonalized in the
            Jacobi transformation.
    """
```

```
    denom_diff = A[y,y] - A[x,x]
    if denom_diff == 0:
        annihilator = np.pi/4.
        c = np.cos(annihilator)
        s = np.sin(annihilator)
    else:
        tau = A[x,y] / denom_diff
        c = np.sqrt(0.5* (1+np.sqrt(1/(1+4*tau**2))))
        s = tau/c * (1-2*c**2)

    return c, s
```

Since we now know c and s explicitly, we can proceed to evaluate all other entries of b_{ij} for which $(i,j) \neq (k,\ell)$. Overall, we thus get the following.

Theorem (*Jacobi transformation*)**.** Let $A = (a_{ij})$ be a real symmetric matrix. The resulting matrix $B = (b_{ij})$ obtained from the xy-Jacobi transformation of A is given explicitly by:

$$b_{ij} = \begin{cases} 0, & \text{if } i,j = x,y \text{ (annihilated term);} \\ ca_{ix} - sa_{iy}, & \text{if } j = x \text{ and } i \neq x,y, \\ ca_{iy} + sa_{ix}, & \text{if } j = y \text{ and } i \neq x,y, \\ c^2a_{yy} + 2sca_{yx} + s^2a_{xx}, & \text{if } i = j = y \text{ (diagonal term);} \\ c^2a_{xx} - 2sca_{yx} + s^2a_{yy}, & \text{if } i = j = x \text{ (diagonal term);} \\ a_{ij}, & \text{if } i,j \neq x,y, \end{cases}$$

where $c = \cos(\theta_0)$ and $s = \sin(\theta_0)$ with θ_0 being the xy-annihilator of A.

Note that $b_{ix} = b_{xi}$ and $b_{iy} = b_{yi}$, so the second and third cases took care of these as well.

The code implementation for this should then be immediate.

Algorithm. The Jacobi transformation.

```
def jacobi_transform(x: int, y: int, A: Matrix) -> Matrix:
    """Returns the Jacobi transformation of A for entry (x, y).

    Args:
        x: The row of the entry (x,y).
        y: The column of the entry (x,y).
        A: The matrix to be diagonalized in the
            Jacobi transformation.
    """
    B = A.copy()
    c, s = get_cs(x, y, B)

    for i in range(len(B)):
        B[y,i] = c*A[y,i] - s*A[x,i]
        B[i,y] = B[y,i]
        B[x,i] = c*A[x,i] + s*A[y,i]
```

```
            B[i,x] = B[x,i]

        # Must do after for loop, otherwise will be overwritten.
        B[y,x] = B[x,y] = 0
        B[x,x] = c**2*A[x,x] + 2*c*s*A[y,x] + s**2*A[y,y]
        B[y,y] = c**2*A[y,y] - 2*c*s*A[y,x] + s**2*A[x,x]

        return B
```

We thus get a full description of B which is the matrix A after a $k\ell$-Jacobi transformation. Repeating this for other off-diagonal entries of A, we will end up with a (nearly) diagonalized matrix — this is the Jacobi method. But before we complete the code for the Jacobi method, some questions should pop up in your head right now. We have a correct algorithm, but is it efficient? Which off-diagonal entry of A should we begin with? Should we attack it at random? Should we just brute-force row by row?

The answer to these questions is the following: just as with Gaussian elimination, full efficiency of this algorithm is attained by executing the Jacobi transformation on the largest (in absolute value) off-diagonal entry of A at each iterative step. By doing this, we now not only have a correct algorithm, but an efficient one as well; let us summarize the steps for a single Jacobi transformation.

1. (*Full efficiency*). Find the largest (in absolute value) entry of A, say, a_{xy} i.e. at the xy entry.

2. (*Finding annihilator*). Find the xy-annihilator in terms of c and s.

3. (*Jacobi transform*). Compute all b_{ij} under the xy-Jacobi transformation of A.

That is, we compute the full b_{ij} description using the c and s that kills a_{xy} so that we get a matrix B similar to A such that $b_{xy} = 0$.

Here is the code to find the largest off-diagonal entry of A.

Algorithm. The function to find maximal off-diagonal element of A.

```
def find_max_offdiag(A: Matrix):
    """Finds the largest off-diagonal element of A."""
    i_max = 0
    j_max = 1  # We initialize with 1 because off-diagonal!
    max_val = A[i_max,j_max]
    len_A = len(A)
    for i in range(len_A):
        for j in range(i+1, len_A):
            abs_val = abs(A[i,j])
            if abs_val <= max_val:
              continue
            i_max, j_max, max_val = i, j, abs_val

    return i_max, j_max, max_val
```

The Jacobi method is then iteratively applying the Jacobi transformation on the largest off-diagonal entry of A. Here is the full implementation.

Algorithm. The Jacobi method

```
def jacobi(A: Matrix) -> Matrix:
    """Diagonalize the square matrix A using the Jacobi method."""
    B = A.copy()
    max_val = 1
    while max_val > 10**(-6):
        i, j, max_val = find_max_offdiag(B)
        B = jacobi_transform(i, j, B)
    return B
```

The eigenvalues of A can thus be read off from the diagonal entries of its diagonalized form. The method thus solves the eigenvalue problem as desired.

10 Optimization problem

One of the fundamental problems in mathematics is that of optimizing, either minimizing or maximizing a function. Some examples include fitting data as we have seen earlier, maximizing profit in an earnings chart or minimizing loss functions. This is a non-trivial problem as convergence can become exponentially slow when the number of parameters increase or when the function is not smooth. We will look at two methods, one which minimizes a single variable real-valued function called the *golden section search* and another which finds critical points of a multivariate real-valued function, called the *Powell's method*.

google "curse of dimensionality". This is a common problem of machine learning and data science.

10.1 Golden section search

A fancy name for a simple minimizing method. Let us consider a real-valued function $f : [a, b] \to \mathbf{R}$ in one variable x. Here is an overall picture of the method. We want to choose two **section points** $x_1, x_2 \in (a, b)$ such that

1. x_1 is nearer to a than b,
2. x_2 is nearer to b than a,
3. $|x_1 - a| = |x_2 - b|$ is true,

Then, we are going to proceed in a binary search fashion. For this we need a condition to choose a subinterval of $[a, b]$ which we will discuss later. The only thing you need to know now is that we either choose $[a, x_2]$ or $[x_1, b]$. The point is that we will then choose new section points x_1', x_2' of one of these subintervals so that they obey the same assumption above together with one additional assumption that we only add one point. For this to happen, one of these section points must coincide with one of the previous section points. This procedure of choosing the initial section points x_1, x_2 is called the **initial section search**, denoted `1S`, while the procedure of choosing the subsequent section points x_1', x_2' is called the **subsequent section search**, denoted `+S`. We will then do more `+S` steps iteratively until we get a convergence.

Here is how we formulate this mathematically. Fix a constant[38] $\alpha \in (0, 1/2)$ and begin the `1S` step by choosing section points $x_1, x_2 \in (a, b)$ in the following way:

$$x_1 = a + \alpha(b - a), \quad x_2 = b - \alpha(b - a). \tag{10.1}$$

[38] *we want this α precisely because of our reasons above. If $\alpha \geqslant 1/2$, say $1/2 + 0.1$, then x_1 is nearer to b than a. Likewise, x_2 is nearer to a than b.*

It is trivial to check that we have satisfy all of our requirements of a section point, so initial section search done. Now, evaluate $f(x_1)$ and $f(x_2)$ and compare them. Suppose $f(x_1) > f(x_2)$, then, we begin our first `+S` step, choosing a **section** of $[a, b]$ by doing three things:

1. Set $y_0 = x_2$,
2. Discard a: set $a' = x_1$,
3. Don't discard b: set $b' = b$.

That is, we now look at the section $[a', b'] = [x_1, b]$. If it was the reverse case $f(x_1) < f(x_2)$, then invert everything[39] so that we look at the section $[a, x_2]$. We now begin our second `+S` step: choose section points of $[a', b']$ which also obey our requirements with an additional

[39] *Set $y_0 = x_1$, don't discard a by setting $a' = a$, and discard b by setting $b' = x_2$.*

assumption that we only add one point (as per our discussion in the overall picture intro). So choose section points $x_1', x_2' \in (a', b')$ in the following way:

$$x_1' = a' + \alpha(b' - a'), \quad x_2' = b' - \alpha(b' - a').$$

This choice obeys our requirements. Now, to additionally ensure that we add only one point, we force that x_1' coincides with x_2 i.e. we want $x_1' = x_2$. But what does this condition means? Let's unpack their definitions. For x_1', we have

$$\begin{aligned} x_1' &= a' + \alpha(b' - a') \\ &= x_1 + \alpha(b - x_1) \\ &= a + \alpha(b - a) + \alpha(b - (a + \alpha(b - a))). \end{aligned}$$

Recalling the definition of x_2 from equation (10.1) and equating them, we end up with the equation

$$(\alpha^2 - 3\alpha + 1)(a - b) = 0.$$

Solving this using the standard quadratic equation root formula, we have

$$\alpha = \frac{3 \pm \sqrt{5}}{2}.$$

But we require that $\alpha \in (0, 1/2)$, and $3 + \sqrt{5} > 1$, so we must choose $\alpha = (3 - \sqrt{5})/2$. So we discover that

$$x_1' = x_2 \iff \alpha = \frac{3 - \sqrt{5}}{2}.$$

That is, we are forced to use $\alpha = (3 - \sqrt{5})/2$ for every `+S` step! Now, evaluate $f(x_1')$ and $f(x_2')$ and suppose $f(x_1') < f(x_2')$. Then we initiate a new `+S` step, choosing a section of $[a', b']$ by doing three things again:

1. Set $y_1 = x_1'$,
2. Don't discard a: set $a'' = a'$,
3. Discard b: set $b'' = x_2'$.

To complete this `+S` step, we choose $x_2'' = x_1'$. But we now know that this condition is equivalent to choosing $\alpha = (3 - \sqrt{5})/2$. Then begin a new `+S` step and repeat the same thing. The procedure is repeated iteratively. This gives a decreasing sequence $\{f(y_n)\}$ of minima of f; and this sequence is convergent as the mesh of the section reduces (by $1 - \alpha$) on each iteration. We will see below that we get a slow convergence rate. However, just like the bisection method, this method has the advantage of having a guaranteed convergence; with a very small caveat.

Caveat. Note that it is possible that this method does not[40] converge to a local minima of f if we poorly choose our initial interval $[a, b]$. Instead, the method would converge to the minima over the interval $[a, b]$ only. For example, if we would to execute this method on $f(x) = -x^2$ which doesn't have a local minima in the first place!

[40] *but still converges!*

Convergence rate. Finally, let us discuss the rate of convergence. Since this method is similar to the bisection method, we would also expect that it is linearly convergent i.e. 1-convergent. This is true as at every step we reduce the interval by a fixed proportion

$$1 - \alpha = \frac{\sqrt{5} - 1}{2} = \frac{1}{\varphi},$$

This explains why we call it the fancy name of golden section search.

where $\varphi = \frac{\sqrt{5}+1}{2}$ is the so-called golden ratio. This means that after n steps, the interval has length

$$\varepsilon_n = (b-a)\frac{1}{\varphi^n},$$

for $n = 0, 1, 2, \ldots$ and so we have

$$\lim_{n\to\infty} \frac{\varepsilon_{n+1}}{\varepsilon_n^q} = \frac{1}{\varphi},$$

for $q = 1$. So it is indeed linearly convergent with rate of convergence $1/\varphi$. The code implementation for the golden section search method is straightforward as it is similar to the bisection method.

Algorithm. Golden section search algorithm.

```
def golden_search(a: Real, b: Real, f: Function):
    """Returns the minima of f(x) on the interval [a, b]
    using the golden section search method.
    """

    def choose_sec_points():
        _alpha = (3-np.sqrt(5)) / 2.
        x1 = a + _alpha*(b-a)
        x2 = b - _alpha*(b-a)
        return x1, x2

    x1, x2 = choose_sec_points()

    while abs(f(x1)-f(x2)) > 10**(-12):
        if f(x1) > f(x2):
            a, y = x1, x2
        else:
            b, y = x2, x1
        x1, x2 = choose_sec_points()

    return y
```

10.2 Powell's method

The golden section search gives us the local minima of a real-valued function in one variable. How about a real-valued function in several variables $F : \mathbf{R}^n \to \mathbf{R}$? If F is smooth, then we can find its critical points using the **Powell's method** , which is, in some sense, a higher-dimensional analogue of the Newton's method. Let us recall a definition.

Definition (*Critical point*). Let $F : \mathbf{R}^n \to \mathbf{R}$ be a smooth function. We say that $\boldsymbol{x}_0 \in \mathbf{R}^n$ is a **critical point** of F if $\nabla F(\boldsymbol{x}_0) = 0$. Special cases of critical points are: local minima, local maxima and saddle point.

Special quadratic case. Before we look at the general case, let us look at a special case. Let $A = (a_{ij})$ be a real $n \times n$ matrix (which we can assume to be symmetric), let $\mathbf{b} = (b_i) \in \mathbf{R}^n$ be a vector, and let $c \in \mathbf{R}$. Now suppose $F(\boldsymbol{x})$ is a smooth function with

the specific quadratic equation

$$F(\boldsymbol{x}) = c + \mathbf{b}\cdot\boldsymbol{x} + \frac{1}{2}\,\boldsymbol{x}^\top A\boldsymbol{x} \tag{10.2}$$

$$= c + \sum_i b_i x_i + \frac{1}{2}\sum_{i,j} a_{ij} x_i x_j. \tag{10.3}$$

Compare these colour highlighting with the colour highlighting in equation (10.5) below.

If we look at a single partial derivative of (10.3) with respect to x_i, we have

$$\frac{\partial F}{\partial x_i} = b_i + \sum_j a_{ij} x_j.$$

This implies that if we compute the gradient of $F(\boldsymbol{x})$, then we have the equation

$$\nabla F(\boldsymbol{x}) = \mathbf{b} + A\boldsymbol{x}.$$

From calculus, we know that $\boldsymbol{x}$ is a critical point of F if and only if $\nabla F(\boldsymbol{x}) = \mathbf{0}$. So if $\boldsymbol{x}$ is a critical point, it must satisfy

$$A\boldsymbol{x} = -\mathbf{b} \implies \boldsymbol{x} = -A^{-1}\mathbf{b}.$$

Recall that the point $\boldsymbol{x}_0$ is a critical point of F if and only if

$$\nabla F(\boldsymbol{x}_0) = \mathbf{0} \iff \frac{\partial F}{\partial x_i}(\boldsymbol{x}_0) = 0,\ \forall\, i$$

General case. Now, let's look at the general case. The idea is to write F as our special quadratic case. If F is smooth, it has a Taylor expansion around $\boldsymbol{x}_0 \in \mathbf{R}^n$, where $\boldsymbol{x}_0$ is educatedly chosen to not be too far away from the expected critical point i.e. it is a *good point*[41]. Ignoring higher order terms, we can get the quadratic approximation of F around $\boldsymbol{x}_0$. Writing $\boldsymbol{x}_0 = (x_0^i) = (x_0^1, \ldots, x_0^n)$, this approximation is

$$F(\boldsymbol{x}) \approx F(\boldsymbol{x}_0) + \nabla F(\boldsymbol{x}_0)\cdot(\boldsymbol{x}-\boldsymbol{x}_0) + \frac{1}{2}(\boldsymbol{x}-\boldsymbol{x}_0)^\top \mathbf{H}(\boldsymbol{x}_0)(\boldsymbol{x}-\boldsymbol{x}_0), \tag{10.4}$$

[41] *In the sense of chapter 1.*

where $\mathbf{H} = (H_{ij})$ is the Hessian matrix

$$H_{ij} = \frac{\partial^2 F}{\partial x_i \partial x_j}$$

of F. The equation (10.4) is then equivalent to

$$F(\boldsymbol{x}) \approx F(\boldsymbol{x}_0) + \sum_i \frac{\partial F}{\partial x_i}(\boldsymbol{x}_0)(x^i - x_0^i) + \frac{1}{2}\sum_{i,j} \frac{\partial^2 F}{\partial x_i \partial x_j}(\boldsymbol{x}_0)(x^i - x_0^i)(x^j - x_0^j). \tag{10.5}$$

Compare these colour highlighting with the colour highlighting in the equation (10.3) above.

By identifying the following terms

$$b_0^i \longleftrightarrow \frac{\partial F}{\partial x_i}(\boldsymbol{x}_0), \qquad \frac{\partial^2 F}{\partial x_i \partial x_j}(\boldsymbol{x}_0) \longleftrightarrow a_0^{ij},$$

where $\mathbf{b}_0 = (b_0^i)$ is a suitable dimensional vector and $A_0 = (a_0^{ij})$ is a suitable dimensional matrix, we are now really in the position of the special quadratic case. So using the same approach as in the quadratic case, $\boldsymbol{x}$ is a critical point of F in this approximation if and only if

$$\boldsymbol{x} - \boldsymbol{x}_0 = -A_0^{-1}\mathbf{b}_0 \iff \boldsymbol{x} = \boldsymbol{x}_0 - A_0^{-1}\mathbf{b}_0.$$

Of course if $b_0^i = \partial F/\partial x_i(\boldsymbol{x}_0)$ is zero for all i i.e. $\mathbf{b}_0 = \mathbf{0}$, then we have $\boldsymbol{x} = \boldsymbol{x}_0$ as a critical point of F itself[42], so we are done. But if $\mathbf{b} \neq \mathbf{0}$, then write $\boldsymbol{x}_1 = \boldsymbol{x}_0 - A_0^{-1}\mathbf{b}_0$ and repeat the process by quadratically expanding around $\boldsymbol{x}_1$. The process goes on indefinitely until

[42] *and not just for the quadratic approximation.*

we want it to terminate.

Remark. We wrote using the notation A_0 and $\mathbf{b}_0$ to highlight the correspondence between our current general case and the previous only quadratic case. In essence, we really have

$$A_0 = \mathbf{H}(\boldsymbol{x}_0), \quad \mathbf{b}_0 = \nabla F(\boldsymbol{x_0}),$$

for this general case now.

Using our remark above, we give the full complete theorem of Powell's method for finding critical points of multivariable scalar smooth functions.

Theorem (*Powell's method*)**.** Let $\boldsymbol{x}^*$ be a critical point of $F : \mathbf{R}^n \to \mathbf{R}$. If $\boldsymbol{x}_n$ is an approximation of $\boldsymbol{x}^*$, then the next approximation is given by

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \mathbf{H}_n^{-1} \nabla F(\boldsymbol{x}_n),$$

where $\mathbf{H}_n = \mathbf{H}(\boldsymbol{x}_n)$ is the Hessian matrix of F at $\boldsymbol{x}_n$; and $\boldsymbol{x}_0$ is chosen so that it is a *good point*

Remark. Note that the Powell's method does not tell us whether the point we obtained as a solution is a maxima, minima or saddle point. It only tells us that it is a critical point, and so we have to do one more extra step of calculating the discriminant of F at that point.

10.2.1 As a multidimensional Newton method

We mentioned earlier that the Powell's method in some sense is a higher-dimensional analogue of the Newton's method. As we shall see, the idea does not transfer verbatim from our previous discussion on finding minima of a scalar-valued function. However, the construction is essentially the same. The problem is the following: find a solution to the system of (possibly non-linear) equations

The one-dimensional case can be solved using our original Newton method, since this is just finding roots of f_1.

Those familiar with algebraic geometry should notice that the f_i defines a variety. So this method helps us solve varities in the reals.

$$\begin{cases} f_1(\boldsymbol{x}) = 0, \\ f_2(\boldsymbol{x}) = 0, \\ \quad \vdots \\ f_d(\boldsymbol{x}) = 0, \end{cases}$$

where the $f_i : \mathbf{R}^n \to \mathbf{R}$ are scalar-valued smooth functions in n variables. To do this, we first define a vector-valued function $\boldsymbol{V} : \mathbf{R}^n \to \mathbf{R}^d$ in the following way:

$$\boldsymbol{V}(\boldsymbol{x}) = \begin{pmatrix} f_1(\boldsymbol{x}) \\ f_2(\boldsymbol{x}) \\ \vdots \\ f_d(\boldsymbol{x}) \end{pmatrix}$$

Now observe that if $\boldsymbol{x}$ is a solution to the original system, it must be a root of $\boldsymbol{V}$. So, our goal now is to solve $\boldsymbol{V} = \mathbf{0}$. Let $\boldsymbol{x}_0 \in \mathbf{R}^n$ be a *good point*. Then Taylor expanding $\boldsymbol{V}$ around $\boldsymbol{x}_0$ so that we have a linear approximation, we get

$$\boldsymbol{V}(\boldsymbol{x}) \approx \boldsymbol{V}(\boldsymbol{x}_0) + \mathbf{J}_{\boldsymbol{V}}(\boldsymbol{x}_0)(\boldsymbol{x} - \boldsymbol{x}_0) \tag{10.6}$$

where $\mathbf{J}_{\boldsymbol{V}} = (J_{ij})$ is the Jacobian matrix

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

of $\boldsymbol{V}$, which is a $d \times n$ real matrix. Write $\mathbf{J}_i = \mathbf{J}_{\boldsymbol{V}}(\boldsymbol{x}_i)$. If $\boldsymbol{x}_1 \in \mathbf{R}^n$ is a solution to (10.6), then we must have

$$\boldsymbol{V}(\boldsymbol{x}_0) + \mathbf{J}_0(\boldsymbol{x}_1 - \boldsymbol{x}_0) = 0 \implies \mathbf{J}_0\boldsymbol{x}_1 = \mathbf{J}_0\boldsymbol{x}_0 - \boldsymbol{V}(\boldsymbol{x}_0).$$

Multiplying by the left inverse of $\mathbf{J}_0$ (we shall assume that it exist), we thus have

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 - \mathbf{J}_0^{-1}\boldsymbol{V}(\boldsymbol{x}_0).$$

We can then repeat the process by considering the linear approximation of $\boldsymbol{V}$ around $\boldsymbol{x}_1$ where we would then get

$$\boldsymbol{x}_2 = \boldsymbol{x}_1 - \mathbf{J}_1^{-1}\boldsymbol{V}(\boldsymbol{x}_1).$$

We proceed in this way indefinitely. The general equation should be obvious:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \mathbf{J}_n^{-1}\boldsymbol{V}(\boldsymbol{x}_n),$$

which is analogous to the Newton's formula.

Theorem (*Multidimensional Newton method*)**.** Let $\boldsymbol{V} : \mathbf{R}^n \to \mathbf{R}^d$ be differentiable and let $\mathbf{J}$ be its Jacobian which has a left inverse. Let $\boldsymbol{x}^*$ be a solution to $\boldsymbol{V}(\boldsymbol{x}) = \mathbf{0}$. If $\boldsymbol{x}_n$ is an approximation of $\boldsymbol{x}^*$ and $\mathbf{J}^{-1}(\boldsymbol{x}_n) \neq \mathbf{0}$, the next approximation is given by

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \mathbf{J}_n^{-1}\boldsymbol{V}(\boldsymbol{x}_n),$$

where $\mathbf{J}_n^{-1} = \mathbf{J}^{-1}(\boldsymbol{x}_n)$, with initial condition $\boldsymbol{x}_0$.

Let's look at an example — this is from the problem sheets.

Example. Suppose we want to solve the following system of equations

$$\begin{cases} x + 2y^2 = 1, \\ 3x^2 + y^3 = 1. \end{cases}$$

Geometrically, this is the intersection of an elliptic curve and a conic (both over affine space).

Then define the vector-valued function $\boldsymbol{V} : \mathbf{R}^2 \to \mathbf{R}^2$ by

$$\boldsymbol{V}(x, y) = \begin{pmatrix} x + 2y^2 - 1 \\ 3x^2 + y^3 - 1 \end{pmatrix}.$$

The Jacobian of $\boldsymbol{V}$ is then given by

$$\mathbf{J} = \begin{pmatrix} 1 & 4y \\ 6x & 3y^2 \end{pmatrix}.$$

This has determinant $\det \mathbf{J} = 3y(y - 8x)$ and so

$$\mathbf{J}^{-1} = \frac{1}{3y(y - 8x)} \begin{pmatrix} 3y^2 & -4y \\ -6x & 1 \end{pmatrix}.$$

By inspection, the point $\boldsymbol{x}_0 = (0, 1)$ seems like a good point. So, the first approximation is given by

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 - \mathbf{J}^{-1}(\boldsymbol{x}_0)\boldsymbol{V}(\boldsymbol{x}_0) = \begin{pmatrix} 0 \\ 1 \end{pmatrix} - \frac{1}{3}\begin{pmatrix} 3 & -4 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}.$$

The second approximation is then given by

$$\boldsymbol{x}_2 = \begin{pmatrix} -1 \\ 1 \end{pmatrix} - \frac{1}{27}\begin{pmatrix} 3 & -4 \\ 6 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 3 \end{pmatrix} = \frac{1}{9}\begin{pmatrix} -5 \\ 8 \end{pmatrix};$$

so on and so forth. After 5 or more iterations, one can then see that we get approximately $(-0.378, 0.830)$ which is an approximation to one of the solutions to the system of equation.

in general, if you are lost in finding a good point, just start with a basis element.

You should be mindblown by now; we have just solved a system of two (very) non-linear equations with very little effort.