# Welcome

## Contents


- Round-off Error
- Linear systems
- Interpolation and Curve-fitting
- Root finding
- Optimization
- Differentiation and integration
- Differential equations
- The Finite Element Method


This is an introduction to the book. Below are the main sections:

Decompose into simpler problems.

- Boil everything down to a simple task (linear system of equations)
- Apprioximte functions as weighted sums of simpler functions (a linear basis)
- Expand functions as Taylor series
- Achieve higher order accuracy by
  - approximating higher order derivatives.
  - cancelling over and under prediction by averaging.

AI is fast. Use your head before the computer or it will run away with you! Nondimensionalize Symmetries

# Round-off Error

Round-off errors are possible whenever you have to write down a number. This is especially problematic for irrational numbers, but also true for rationals!

```
from numpy import pi, e, sqrt, binary_repr
print(pi, e, 1/3, sqrt(2))
```

The problem is that we can only write down a finite number of digits before we get tired. So we round the last digit or chop it.

This error is inherant to *finite precision numerics*, not necessarily computations. The problem is that if computations include millions of calculations, these little mistakes can add up.

> The average human is capable of around 1 mistake per second, but computers are capable of millions of mistakes a second!

NB: If we could use infinite digits, or represent numbers symbolically (e.g. 1/3, \(\pi\), \(\sqrt 2\)), this wouldn't be an issue. This is why tools like Mathematica, Maple, or the sympy module, don't evaluate anything until they have too.

Q: What are the pros and cons of keeping symbols?

The impact of roundoff error is related to the precision (the number of digits we write down), and the way (base) we write numbers in.

# Finite precision representations

## Binary representation

Humans use a base-10 numbering system called ‘decimal’, probably because that’s how many fingers we (typically) have.

> Except for Mayans who used base-20: ‘vegesimal’ system! I guess they could only do math sitting down!

The number 1305 is expressed in decimal with each column indicating a power of the base ($10$):
$1305_{10} = 5 \times 10^0 + 0 \times 10^1 + 3 \times 10^2 + 1 \times 10^3$

NB: We wrote the order of digits backwards so we could go in increasing powers.

Computers use base 2 (binary) since a bit can only be 0 or 1. The same number is written as:

```
# prompt: convert 1305 into binary

binary_repr(1305)
```

which we can check:

$101000011001_2 = 1 \times 2^0 + 0 \times 2^1 + ... + 1 \times 2^3 + 1 \times 2^4+... + 1 \times 2^8 + 0 \times 2^{9} + 1 \times 2^{10}$
$=1305_{10}$

We can also use a decimal point with binary.

$54.75_{10} = 5\times 10^{-2} + 7\times 10^{-1} + 4 \times 10^{0} + 5 \times 10^{1}$

```
# prompt: express 54.75 in binary

from numpy import binary_repr

def decimal_to_binary(number):
  # Convert the integer part to binary
  integer_part = binary_repr(int(number))

  # Convert the fractional part to binary
  fractional_part = number - int(number)
  binary_fractional_part = ""
  for i in range(20):
      fractional_part *= 2
      if fractional_part >= 1:
          binary_fractional_part += "1"
          fractional_part -= 1
      else:
          binary_fractional_part += "0"
      if fractional_part == 0:
          break

# Combine the integer and fractional parts
  binary_representation = integer_part + "." + binary_fractional_part

  print(binary_representation)

decimal_to_binary(54.75)
```

Check:
$1 \times 2^{-2} + 1 \times 2^{-1} + 0 \times 2^{0} + 1 \times 2^{1} + 1 \times 2^{2} + 0 \times 2^{3} + 1 \times 2^{4} + 1 \times 2^{5} = 54.75_{10}$

### Example Convert 0.1 to binary

```
decimal_to_binary(0.1)
```

The binary representation of 0.1 is a repeating number!

## Precision

Computers use a standard data unit, called a *word*. The number of bits in each word is called the *precision* and is, by IEEE convention, in increments of 32:

| Precision | # bits |
|---|---|
| single | 32 |
| double | 64 |
| quad | 128 |

For comparison, the previous number 10100011001 takes 11 bits.

The most common precision in modern computing, and the standard in python3, is double precision. Quad precision is occasionally accessible for precise calculation.

## Integers

Integers are a fundamental data type if you don't need fractions, and **do not suffer from roundoff error**! However, since they have a finite number of digits (bits) their size is limited.

The range of values an integer can store is \( 2^{bits}\). Integers are signed, so we must include if the number is +'ve or -'ve. Furthermore, there is a redundancy where -0 = +0, leading to the range of -'ves being larger than that of +'ve.

The min and max numbers an integer can represent is therefore:

\(min = -2^{bits-1}\)

\(max = 2^{bits-1} -1\)

> You may be tempted to use a bit to represent the sign. This is not modern practice for integers, which instead use a method called *Two's complement*.

### Example: What is the largest integer a double precision variable can store?

```
print('min ', -2**63, '\nmax  ', 2**63-1)

print('\nCheck with the built-in numpy examiner')
import numpy as np

print(np.iinfo(np.int64))
```

### Example 2: Let's break it!

```
# Works
print(np.int64(2**62))

#Overflow error
print(np.int64(2**63))
```

```
#Works
print(np.int64(1000000000000000000))

#Overflow error
print(np.int64(10000000000000000000))
```

#Floating point numbers

Writing a number like \(10 000 000 000 000 000 000\) isn't really useful. It is much better to isolate the magnitude in units, or as an exponent:

\(10 000 000 000 000 000 000 = 10^{19}\)

## Floating point Decimal numbers (aka: Engineering notation)

Remove leading zeros and *placeholder* trailing zeros using a *floating point* to separate the fractional part (mantissa / significand) from the order of magnitude (exponent).

Engineering notation = \(mantisa \times 10^{exponent}\)

| Decimal | Engineering | Mantissa | Exponent |
|---|---|---|---|
| \(265.73\) | \(2.6573 \times 10^2\) | 2.6573 | 2 |
| \(.0001\) | \(1 \times 10^{-4}\) | 1 | -4 |
| \(-0.0034123\) | \(-3.4123 \times 10^{-3}\) | -3.4123 | -3 |
| \(1500^*\) | \(1.5 \times 10^3\) | 1.5 | 3 |

*only if the trailing zeros are not actually measured.

Note:

1. The mantissa is a fraction, but if we *normalize* the fraction to have the decimal after the first digit, we can represent it as an integer.
2. The exponent is the power of the number system *base*, in this case \(10\).

```
# prompt: Convert a number to engineering notation

def to_engineering_notation(number):
  if number == 0:
    return "0"

  exponent = 0
  while abs(number) < 1:
    number *= 10
    exponent -= 1
  while abs(number) >= 10:
    number /= 10
    exponent += 1

  print(f"{number}E{exponent}")

to_engineering_notation(265.73)
to_engineering_notation(0.0001)
to_engineering_notation(-.0034123)
to_engineering_notation(1500)
to_engineering_notation(0)

# You can also use Log10 to calculate this
```

## Floating point Binary numbers

The same technique can be applied to binary by using base 2:

$mantisa \times 2^{exponent}$

### Example convert 54.75 into floating point binary

$54.75_{10} = 110110.11_2$

$= 1.1011011_2 \times 2^5$

$= 1.1011011_2 \times 2^{101}$

## Precision in floating point numbers

If the mantissa and exponent have infinite range, we can represent all numbers using floating point. However we are once again limited by the number of bits (precision). Now, the bits are divided into sign, mantissa, and exponent by convent: IEEE Standard for Floating-Point Arithmetic (IEEE 754):

| Precision | # bits | Sign | Exponent | Mantissa |
|---|---|---|---|---|
| Single | 1/8/23 | S | EEEEEEEE | FFFFFFFFFFFFFFFFFFFFFFF |
| Double | 1/11/52 | S | EEEEEEEEEEE | FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF |
| Quad | 1/15/112 | S | EEEEEEEEEEEEEEE | FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF |

Note that the ‘sign’ bit is now here and is the sign of the number. The sign of the exponent is one of the bits in the ‘exponent’ block.

The actual storage is a bit complicated, but the key for us is the finite precision of the mantissa.

### Example How is 0.1 actually stored?

```
print(format(0.1, '.55f'))
```

```
error = 0.00000000000000000555111512312578270211815834045410156 25
eps_r = error / 0.1
print(eps_r)
```

## Summary

In practice, we have to be careful when we mixing the order of terms. i.e. adding terms of different magnitude, or subtracting terms of slightly-varying magnitude.

We cannot count on the associative property:

```
print(-1+(1+1e-20))
print((-1+1)+1e-20)
```

Beware of subtractive cancellation!

```
# Define two nearly equal numbers
a = np.float32(1.23456789)
b = np.float32(1.23456780)

# Perform subtraction
result = a - b

# Print the results with higher precision
print("a =", format(a, '.20f'))
print("b =", format(b, '.20f'))
print("a - b =", result)
```

# Quantifying error

## True Absolute Error

The difference between the true answer and an approximate answer is called the True Absolute Error:

$E_t = |True - Approx|$

Only really useful when the *true* value is known.

### Example: What is the derivative of $sin(x)$ at $x=0$?

Recall: $\frac{d \sin (x)}{dx} = lim_{\Delta x->0} \frac{sin(x+\Delta x)-sin(x)}{\Delta x}$

True: $\frac{d \sin (x)}{dx} = cos(x)$

```
from numpy import cos, sin, abs
value_true = cos(0)
print(value_true)
```

Define a function (for later use) approx which takes a parameter, $\Delta x$:

$approx(\Delta x) = \frac{sin(x+\Delta x)-sin(x)}{\Delta x}$

```
def approx(delta_x):
    return (sin(0+delta_x)-sin(0))/delta_x
```

For $\Delta x = .1$,

```
value_approx = approx(delta_x = .1)
print(value_approx)
```

The true absolute error is therefore,

```
E_t = abs(value_true - value_approx)
print(E_t)
```

## True relative error

Often the absolute error is not so useful since it doesn't consider the magnitude of the value.

E.g. GPS has an error of about ~1m. Is that important for calculating distance for a long road trip? What about a self-driving the car?

Also note that absolute error has units which complicates generalization.

The true relative error is defined as,

$\epsilon_t = \frac{E_t}{True}$

or as a percent,

$\frac{E_t}{True} \times 100 \%$

### Example: What is the relative error from the previous calculation?

```
eps_t = E_t/value_true
print(eps_t)
```

## Approximate absolute and relative error

What if we don't have the true value?

Numerical methods typically have a tunable parameter that controls accuracy (viz. $\Delta x$ above). We can estimate the error for sequential approximations, using the *better* approximation in place of the *True* one.

$E_a = |Better\ approx - approx|$

$\epsilon_a = \frac{E_a}{Better\ approx}$

### Example: Use smaller step sizes to find the approximate error and fracitonal error.

```
E_a = abs(approx(.01) - approx(.1))
print(E_a)
```

```
epsilon_a = E_a/approx(.01)
print(epsilon_a)
```

## Tolerance

Since we don't know the true answer, we can never say that we have reached it. What we can say is that the answer *isn't getting any better*.

Programmatically, we say that the error / fractional error has crossed a *tolerance*.

Define:

The absolute tolerance, \(Tol_a\), is the threshold past which the absolute error is *small enough*.

The relative tolerance, \(Tol_r\), is the threshold past which the relative error is *small enough*.

## Pseudo code concept

```
Run an algorithm for a given parameter
loop:
  reduce parameter
  run algorithm
  Calculate error and relative error
  Exit when tolerance is met
```

### Example: Lets explore \(E_a\) and \(\epsilon_a\) as a function of \(\Delta x\):

```
# prompt: Plot E_a and epsilon_a vs delta_x for decending values of delta_x to 1e-10 on a log-log with ma

import matplotlib.pyplot as plt
import numpy as np

delta_x = np.logspace(0,-10,11)
E_a = np.zeros(delta_x.size)
epsilon_a = np.zeros(delta_x.size)

for i, dx in enumerate(delta_x):
  E_a[i] = approx(dx/10) - approx(dx)
  epsilon_a[i] = E_a[i]/approx(dx/10)

plt.loglog(delta_x, E_a, marker='o', label='$E_a$')
plt.loglog(delta_x, epsilon_a, marker='s', label='$\epsilon_a$')
plt.xlabel('$\Delta x$')
plt.ylabel('Error')
plt.legend()
plt.show()
```

Note:

- The two errors overlap. Why?
- The plot is a straight line. What does this mean?
- There is a precipitous drop-off at \(10^{-7}\). What is that?

THESE QUESTIONS AND MORE IN THE NEXT EXCITING EPISODE OF NUMERICAL METHODS FOR ENGINEERING!

# Truncation error

Truncation error occurs when we *approximate* a mathematical function.

## Taylor series

Recall the *EVER SO USEFUL* Taylor series:

$$f(x+\Delta x) = f(x) + f'(x) \Delta x + f''(x) \frac{\Delta x ^2}{2} + f'''(x) \frac{\Delta x ^3}{6} + ...$$

$$= \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!} \Delta x ^n$$

but this is not useful unless we have an infinite amount of time and resources.

If $\Delta x$ is *small*, $\Delta x ^2$ is smaller, and $\Delta x ^3$ smaller still. In fact, as long as $f(x)$ is *well behaved* (loosely defined as continuous, smooth, differentiable, not infinite, etc) the derivatives don't explode exponentially and the rightmost terms get very small.

So let's *truncate* the series and only keep the first $k$ terms:

$$f(x+\Delta x) = f(x) + f'(x) \Delta x + f''(x) \frac{\Delta x ^2}{2} + E_k$$

where

$$E_k = \sum_{n=k}^{\infty} \frac{f^{(n)}(x)}{n!} \Delta x ^n$$

is the **truncation error**.

This quantity is akin to a True Error in that if we knew what $E_k$ was exactly, we would have the true function $f$!

It is more useful to define the *order* of the error. Noting that the leading term is $\propto \Delta x ^k$, we would say:

$$f(x_0+\Delta x) \approx f(x_0) + f'(x_0) \Delta x + f''(x_0) \frac{\Delta x ^2}{2} + \mathcal{O}(\Delta x ^3)$$

or that this is a *third* order approximation.

This is a useful statement, since it indicates the payoff for tuning the numerical parameters. In this case, halving the step size halves the error.

### Example: Find the order of approximate derivative we calculated previously (known as the *forward* difference).

$$f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

We can substitute the Taylor series for $f(x+\Delta x)$

$$f'(x) \approx \frac{f(x) + f'(x_0) \Delta x + f''(x_0) \frac{\Delta x ^2}{2} ... - f(x)}{\Delta x}$$

$$\approx \frac{f'(x_0) \Delta x + f''(x_0) \frac{\Delta x ^2}{2} ...}{\Delta x}$$

$$\approx f'(x_0) + f''(x_0) \frac{\Delta x}{2} ...$$

$$\approx f'(x_0) + f''(x_0) \frac{\Delta x}{2} ...$$

$$\approx f'(x_0) + \mathcal{O}(\Delta x)$$

Therefore this approximation, call the *forward difference* is a first order algorithm.

### Example 2: Find the order of approximatino of the central difference formula,

$$f'(x) \approx \frac{f(x+\Delta x) - f(x-\Delta x)}{2 \Delta x}$$

Substituting the (3rd order) Taylor series for $f(x+\Delta x)$ and $f(x+[-\Delta x])$

$$f'(x) \approx \frac{f(x) + f'(x) \Delta x + f''(x) \frac{\Delta x ^2}{2} + f'''(x_0) \frac{\Delta x ^3}{6} ... - [f(x) + f'(x) [-\Delta x] + f''(x) \frac{-\Delta x ^2}{2} + f'''(x_0) \frac{-\Delta x ^3}{6}...]}{2 \Delta x}$$

$$\approx \frac{f(x) + f'(x) \Delta x + f''(x) \frac{\Delta x ^2}{2} + f'''(x_0) \frac{\Delta x ^3}{6} ... - f(x) + f'(x) \Delta x - f''(x) \frac{-\Delta x ^2}{2} + f'''(x_0) \frac{-\Delta x ^3}{6}...}{2 \Delta x}$$

$\approx \frac{2 f'(x) \Delta x + 2 f'''(x_0) \frac{\Delta x ^3}{6} ...}{2 \Delta x}$

$\approx f'(x) + f'''(x_0) \frac{\Delta x ^2}{6} ...$

$\approx f'(x) + \mathcal{O}(\Delta x ^2)$

Therefore, the central difference approximation is a second order algorithm (for the same number of function calls!). By halving the step size, the error is quartered!

## Common mathematical functions

Computers are very good at addition / subtraction, multiplication / division, and exponentiation. How should we calculate other functions?

Let's examine some Taylor expansions:

| Function | Taylor Expansion |
|---|---|
| $\sin(x)$ | $x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$ |
| $\cos(x)$ | $1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$ |
| $\exp(x)$ | $1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$ |
| $\ln(1+x)$ | $x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots$ |

On the surface these look good, and in *infinite* precision they are globally convergent.

But we are not in *infinite* precision.

### Example: Examine the terms of $sin(x)$ for small x

```
from numpy import pi, e, sqrt, binary_repr
import numpy as np
import math

def taylor_series_sin(x, n_terms):
    """
    Calculate the Taylor series expansion of sin(x) up to n_terms.

    Parameters:
    x (float): The point at which to evaluate the Taylor series.
    n_terms (int): The number of terms to include in the expansion.

    Returns:
    list: A list of terms in the Taylor series expansion.
    """
    terms = []
    for n in range(n_terms):
        term = ((-1)**n * x**(2*n + 1)) / math.factorial(2*n + 1)
        terms.append(term)
    print("The terms are as follows:")
    for i, term in enumerate(terms):
        print(f"Term {i+1}: {term:.10f}")
    print(f"Approximate sin({x}): {sum(terms):.10f}, and it should be {math.sin(x):.10f}")
```

For a small x:

```
taylor_series_sin(1, 10)
```

NB: The terms are flipping signs (potential for roundoff error), but more importantly they are decresing.

-> No problem

### Example: Examine the terms of sin(x) for large x

```
taylor_series_sin(10, 10)
```

Getting a bit funny…

```
taylor_series_sin(100, 10)
```

Completely wrong.

## Why you should use a package

In this case, the remedy is fairly simple, but if you are not careful, these function can behave very strangely. In practice, the means of calculation are very sophisticated for performance and stability, including other expansion techniques and sometimes even look-up tables.

This is why we use packages! :-)

The Taylor expansion is still useful to consider limiting behaviour.

For small $x$,

$exp(x) \approx 1+x$ which is subject to roundoff error. Therefore packages like numpy provide special functions like $expm1 = exp(x)-1$

```
# Poorer approximation
print(np.exp(1e-10) - 1)

#Better approximation
print(np.expm1(1e-10))
```

# Linear systems

Linear system solvers are the workhorse of scientific computing, and includes the canonical *solve*.

E.g. the trivial equation for $x$ with scalars $a$ and $b$:

$ax = b$

$x=b/a$

> In contrast: $x^2 = 1$ is nonlinear.

In general, we will have $n$ unknowns that must be solved *simultaneously* and a set of $m$ linear equations. The group of such equations is termed a *linear system* and is written compactly in matrix form:

$A x = b$

where

$A$ is the *coefficient matrix* of dimension $m \times n$

$x$ is the *variable / unknown vector* of dimension (length) $n$

$b$ is the *constant / right-hand side* vector of dimension $m$.

In most cases in this course we will be dealing with \(n\) equations and \(n\) unkowns. In that case, \(m=n\) and \(A\) is called \(square\).

# Solvability

You are organizing a fundraising event and need to buy chairs and tables. Chairs cost $20 each and tables cost $50 each. You have a budget of $700 and need a total of 20 pieces of furniture (chairs and tables combined). How many chairs and tables should you buy?

## A system with one solution

Let \(c\) and \(t\) be the number of chairs and tables respectively. The budget and pieces equations are,

(1) \(20 c + 50 t = 700\)

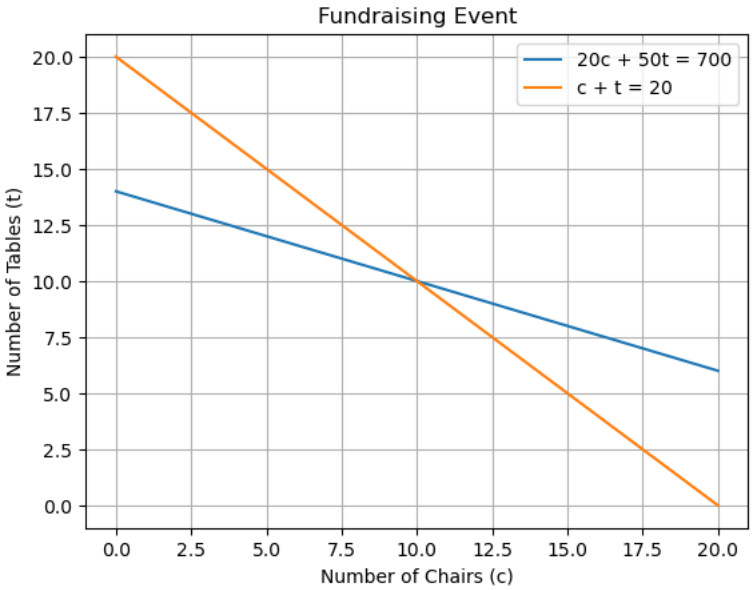
(2) \( c+t = 20\)

There are a few ways to solve these equations.

### Solve graphically

Since these are lines, let's plot them!

```
# prompt: Plot the two lines with a grid

import matplotlib.pyplot as plt
import numpy as np
# Define the x values
x = np.linspace(0, 20, 100)

# Calculate the y values for the first equation (20c + 50t = 700)
y1 = (700 - 20 * x) / 50

# Calculate the y values for the second equation (c + t = 20)
y2 = 20 - x

# Plot the lines
plt.plot(x, y1, label='20c + 50t = 700')
plt.plot(x, y2, label='c + t = 20')

# Add labels and title
plt.xlabel('Number of Chairs (c)')
plt.ylabel('Number of Tables (t)')
plt.title('Fundraising Event')

# Add a grid
plt.grid(True)

# Add a legend
plt.legend()

# Display the plot
plt.show()
```




The point where the lines intersect satisfy both equations and is therefore a solution. Since lines only cross once, it is the unique solution.

## Solve through elimination

Multiply the second equation, (2), by \(20\):

(3) \(20c+20t = 400\).

Subtract (3) from (1) and simplify:

\(30t = 300\)

\(t=10\)

Substitute answer into (2):

\(c = 10\)

## Matrix formulation and solution

Writting these as a matrix equation becomes:

\(\begin{pmatrix} 20 & 50 \\ 1 &1 \end{pmatrix} \begin{pmatrix} c \\ t \end{pmatrix} = \begin{pmatrix} 700 \\ 20 \end{pmatrix}\)

or in standard form,

\(A x = b\)

with \(A = \begin{pmatrix} 20 & 50 \\ 1 &1 \end{pmatrix}\)

\(x = \begin{pmatrix} c \\ t \end{pmatrix}\)

\(b = \begin{pmatrix} 700 \\ 20 \end{pmatrix}\)

Let's find \(A^{-1}\) such that \(x = A^{-1}b\). For a square matrix of dimensions 2x2:

\(\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{|A|} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}\)

where \(|A| = ad-bc\) is the *determinant*.

> The prefactor of $\frac{1}{|A|}$ is systemic to inversion. In general, $A^{-1} = \frac{1}{|A|} adj(A)$ for square matricies of any dimension.

For our case,

$|A| = -30$, and

$A^{-1} = \frac{1}{-30} \begin{pmatrix} 1 & -50 \\ -1 & 20 \end{pmatrix}$

thus, $A^{-1} b$:

$x = \begin{pmatrix} 10 \\ 10 \end{pmatrix}$

## Infinite solutions

Lets tweak our problem and see what happens.

There is now a discount on tables down to $20 each. The customer heard about it and cut your budget to $400.

The problem is now:

$20 c + 20 t = 400$

$c+t = 20$

### Graphically

```
# prompt: Plot the two lines with a grid making the first line thicker

import matplotlib.pyplot as plt
import numpy as np
# Define the x values
x = np.linspace(0, 20, 100)

# Calculate the y values for the first equation (20c + 20t = 400)
y1 = (400 - 20 * x) / 20

# Calculate the y values for the second equation (c + t = 20)
y2 = 20 - x

# Plot the lines
plt.plot(x, y1, label='20c + 20t = 400', linewidth=3)  # Make the first line thicker
plt.plot(x, y2, label='c + t = 20')

# Add labels and title
plt.xlabel('Number of Chairs (c)')
plt.ylabel('Number of Tables (t)')
plt.title('Fundraising Event (Revised)')

# Add a grid
plt.grid(True)

# Add a legend
plt.legend()

# Display the plot
plt.show()
```

The lines overlap! What does this mean?

### Elimination

Multiple second row by 20:

$20c+20t = 400$

subtracting the first we get,

$0=0$

:-(

Solve the second equation to find $c = 20/t$

and that's it! For all $t$ there is a $c$ that is a solution!

### The matrix equation

$|A| = ad−bc = 0$

What does this mean for the inverse $A^{-1}$?

## No solutions

WOOPS! The customer meant to say $500; no more, no less!

The problem is now

$20 c + 20 t = 500$

$ c+t = 20$

### Graphically

```
# prompt: Graph it again please!

import matplotlib.pyplot as plt
import numpy as np
# Define the x values
x = np.linspace(0, 20, 100)

# Calculate the y values for the first equation (20c + 20t = 500)
y1 = (500 - 20 * x) / 20

# Calculate the y values for the second equation (c + t = 20)
y2 = 20 - x

# Plot the lines
plt.plot(x, y1, label='20c + 20t = 500', linewidth=3)  # Make the first line thicker
plt.plot(x, y2, label='c + t = 20')

# Add labels and title
plt.xlabel('Number of Chairs (c)')
plt.ylabel('Number of Tables (t)')
plt.title('Fundraising Event (Revised Again)')

# Add a grid
plt.grid(True)

# Add a legend
plt.legend()

# Display the plot
plt.show()
```

Now they are parralel! What does THIS mean?

### Elimination

The second row multiplied by 20 is still $20c+20t = 400$ (!!). Now subtracting the first becomes:

$20c + 20t = 500$

-$20c - 20t = -400$

$----------$

$0+0=100$

… >:-(

### And the matrix equation?

Unchanged since only $b$ has change!

(What does **this** tell you?!?)

## Putting it together

Linear equations (in 2 unkowns) are lines in 2D. The solution is the intersection of those lines. 2 Lines can intersect either

1. in one place (the first example)
2. everywhere (the second example)
3. nowhere (the third example)

For 2 and 3, these lines are parallel, i.e. you can slide one to lie ontop of the other.Such lines are called *linearly dependent*.

Example 1 has *linear independent* equations which intersect in one place and can be solved.

Scenarios 1 and 2 are called a *consistent* linear system since an answer can be obtained. Scenario 3 is *inconsistent* since there is no solution.

### The matrix interpretation

The coefficient matrix $A$ depends on the nature of the lines, *not the constant*. When

$|A| = 0$,

The matrix $A$ is termed *singular*. The lines are parallel, which means the equations / rows in $A$ and linear dependant and you will not be able to solve for a unique $x$.

This is true regardless of the values of $b$!

# Computational complexity

The *computational complexity* of an algorithm characterizes the number of operations it requires (thus comparing the algorithm instead of the hardware). Linear algebra algorithms are characterized in terms of the dimension of their arguments (vector length, matrix size, etc). Complexity is represented in big O notation (similar to the accuracy of function approximations). I.e.: Only the leading order is kept, and constants disregarded.

## Example

Traditional matrix multiplication of two $n \times n$ matricies - involves $n^3$ operations and is thus $O(n^3)$

NB: This is algorithm-dependant - The Strassen algorithm has $O(n^{2.81})$

```
def time_matmult(n):
  import numpy as np
  A = np.random.rand(n, n)
  B = np.random.rand(n, n)
  %timeit C = A @ B

time_matmult(100)
time_matmult(1000)
time_matmult(2000)
time_matmult(4000)
```

```
123 µs ± 51.3 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
98.4 ms ± 37.9 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
514 ms ± 11 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
4.64 s ± 736 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Some complexities of common linear algebra operations:

| Operation | Description | Complexity |
|---|---|---|
| Matrix addition | Adding two matrices of the same size | $O(n^2)$ |
| Matrix multiplication | Multiplying two matrices | $O(n^3)$ (standard algorithm) |
| Matrix-vector multiplication | Multiplying a matrix by a vector | $O(n^2)$ |
| Matrix inversion | Finding the inverse of a matrix | $O(n^3)$ |
| Determinant calculation | Computing the determinant of a matrix | $O(n^3)$ |
| Transpose | Swapping rows and columns of a matrix | $O(n^2)$ |
| Matrix norm | Sqrt of sum of squares | $O(n^2)$ |

High-performance computing introduces a new element to complexity which deals with how algorithms parallelize. This is called *scaling* and measured in $O(number \ of \ nodes)$. The goal is usually $O(n)$ but is hard to achieve!

# Conditioning

The matrix interpretation is troubling. The solvability of these systems is related to $det(A)$ and we have 2 data points:

1. $|A| = -30$ and the system is solvable.
2. $|A| = 0$ and the system is unsolvable.

What is the significance of $-30$?. What happens when $|A| \rightarrow 0$ ?

We saw $|A| \rightarrow 0$ when we adjusted the price of tables. Let's play with that! Set the cost of tables to $20-\epsilon$.

$|A| = 20-20+\epsilon = \epsilon$

Now look at the inverse equation:

$A^{-1} \sim \frac{1}{|A|} \sim \frac{1}{\epsilon}$

As $\epsilon \rightarrow 0$, this fraction goes to infinity!

The problem is:

$20 c + [20+\epsilon] t = 700$

$ c + t = 10$

## Graphically,

```
import matplotlib.pyplot as plt
import numpy as np

# Define the x values
x = np.linspace(-200, 30, 100)

# Define the epsilon values
epsilons = [0, -1, -10, -30]

plt.plot(x, 20 - x, label='c + t = 20', linewidth = 3)

# Plot the lines for each epsilon value
for epsilon in epsilons:
  y1 = (700 - 20* x) / (20 - epsilon)

  # Plot the lines
  plt.plot(x, y1, label=f'epsilon = {epsilon}')

# Add labels and title
plt.xlabel('Number of Chairs (c)')
plt.ylabel('Number of Tables (t)')
plt.title('Fundraising Event with Varying Epsilon')

# Add a grid
plt.grid(True)

# Add a legend
plt.legend()

# Display the plot
plt.show()
```

The lines are getting more and more parallel / linearly dependent.

NOTE: There is a solution until $\epsilon =0$, but the answer is both moving away and moving at an increasing rate!

## Elimination

Let's use $c = 20-t$ and substitute into the first equation:

$$ 20 [20 -t] + [20-\epsilon] t = 700$$

$$\epsilon t= 300$$

$$ t = 300 / \epsilon$$

and again we see an infinite answer.

Ex:

| $\epsilon$ | t | c |
|---|---|---|
| -30 | 10 | 10 |
| -10 | 30 | -10 |
| -1 | 300 | -280 |
| 0.1 | 3000 | -2980 |
| 1e-20 | 3e+22 | -3e+22 |
| 0 | $\infty$ | $-\infty$ |

This should sound alarm bells!

1. Recall roundoff error - 1e-20 is well within the range of roundoff error. Here, even just *recording* the matrix element value can cause astronomical changes in the solution!
2. These coefficients typically have some uncertainty which compounds the numberical uncertainty.

## The condition number

Let's try to quantify this issue by returning to the importance of $|A| = -30$. We could multiply the first equation by an arbitrary scalar § and the determinant would scale accordingly, so obviously the magnitude of $|A|$ can't be important.

It is only important relative to the *magnitude* of the equation coefficients (the elements of $A$).

### Matrix norms

For vectors we know the magnitude (called the *norm*) is: $||v|| = \sqrt{\sum_n v_i^2} = \sqrt{v\cdot v}$.

> In fact this is more than one type of vector norm. In general, $||v||_p = \sum_i \left(v_i^p\right) ^{1/p}$. The p=2 norm is the 'Euclidian norm', for $p\rightarrow \infty$, $|v|_{\infty} = max(v_i)$.

The Euclidian / Frobenius norm is simply,

$||A||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |A_{ij}|^2}$

$||A||_F = \sqrt{A:A}$

Infinity Norm (maximum row-sum norm)

$||A||_\infty = \max_{1 \le i \le m} \sum_{j=1}^{n} |A_{ij}|$

Spectral, or 2-norm $||A||_2 = \max(eigen value)$

The determinant is considered *small* in comparison with the matrix norm $|A| << ||A||$.

In actuality, C.N. = $\frac{max(eigenvalue)}{min(eigenvalue)}$ but hard to find.

The *matrix condition number*

$cond(A) = ||A|| ||A^{-1}||$

is the formal measure of conditioning. For $cond(A) ~1$ the matrix is well-conditioned and its inversion is numerically stable. Otherwise it is *ill-conditioned* and we will need to do something else.

If the eigenvalues can be calculated (or estimated) then we can do the spectral condition number,

$cond(A) = \frac{\max(e.v.)}{\min(e.v.)}$

(generallized to singular values).

# BONUS head-scratchers!

Lets go back to the original problem

(1) $20 c + 50 t = 700$

(2) $ c+t = 20$

and add some other requirement that amounts to:

(3) \( 50 c + 20 t = 700\)

Some quick math will show these lines are linearly independant, and graphically we see there is a solution,

```
# prompt: Plot the 3 lines above

import matplotlib.pyplot as plt
import numpy as np

# Define the x values
x = np.linspace(0, 20, 100)

# Calculate the y values for the first equation (20c + 50t = 700)
y1 = (700 - 20 * x) / 50

# Calculate the y values for the second equation (c + t = 20)
y2 = 20 - x

# Calculate the y values for the third equation (50c + 20t = 700)
y3 = (700 - 50 * x) / 20

# Plot the lines
plt.plot(x, y1, label='20c + 50t = 700')
plt.plot(x, y2, label='c + t = 20')
plt.plot(x, y3, label='50c + 20t = 700')

# Add labels and title
plt.xlabel('Number of Chairs (c)')
plt.ylabel('Number of Tables (t)')
plt.title('Fundraising Event with 3 Equations')

# Add a grid
plt.grid(True)

# Add a legend
plt.legend()

# Display the plot
plt.show()
```

so what do we do now?!? (Attend advanced numerical methods to learn about pseudoinverses!)

# Direct methods

Solve for the solution *directly*, in a set number of steps.


## Solving linear systems with Gauss elimination

Gauss elimination is an algorithm for the most familiar / intuitive solution technique. Let's reexamine our analytical (symbolic) solution to the previous problem and then make it into a numerical algorithm.

Question: You are organizing a fundraising event and need to buy chairs and tables. Chairs cost $20 each and tables cost $50 each. You have a budget of $700 and need a total of 20 pieces of furniture (chairs and tables combined). How many chairs and tables should you buy?

### Symbolic manipulation

Let \(c\) and \(t\) be the number of chairs and tables respectively.

The budget and pieces equations are,

(1) \(20 c + 50 t = 700\)

(2) \( c+t = 20\)

Solve (2) for $c$:

$c = 20-t$

Substitute into (1):

$20 [20-t] + 50t = 700$

$t = 10$

and substitute into (2) or (1) to find $c$.

This works because our first step carries the unkown *symbol* $t$.

## Numerical approach

Lets repeat this without carrying symbols.

(1) $20 c + 50 t = 700$

(2) $ c+t = 20$

Multiply (2) by $20$:

$20 c + 20 t = 400$

and subtract from (1):

$$\begin{eqnarray*} 20 c + &50 t &=& 700 \ -20 c - &20 t &=& -400 \ \hline &30 t &=& 300 \end{eqnarray*}$$

Thus we have simplified the last line until it reaches a trival solution for $t$. Now it is a matter of *substitution* to solve for $c$.

NB: the linear system has been changed (without changing the answer) to:

$$\begin{pmatrix} 20 & 50 \\ 0 & 30 \end{pmatrix} \begin{pmatrix} c \\ t \end{pmatrix} = \begin{pmatrix} 700 \\ 300 \end{pmatrix}$$

In particular, $A$ is *upper triangular*, (aka *row echelon form* for a square matrix) from which the answer is easily obtained through *backward substitution*.

##Upper triangular matricies

An upper triangular matrix, $U$, is a matrix whose elements below the diagonal are zero:

$$\begin{split}\begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4}\\ 0 & u_{2,2}' & u_{2,3}' & u_{2,4}'\\ 0 & 0 & u_{3,3}' & u_{3,4}' \\ 0 & 0 & 0 & u_{4,4}' \end{bmatrix}\end{split}$$

This is useful because, in equation form,

$U x = b'$

turns in to $$\begin{eqnarray*} \begin{array}{} u_{1,1} x_1 &+& u_{1,2} x_2 & + & u_{1,3} x_{3} &+&u_{1,4} x_4 &=& b'*1,\ & & & u*{2,2}' x_{2} &+ & u_{2,3}' x_{3} &+& u_{2,4}' x_4 &=& b_{2}' \ && & & u_{3,3}' x_{3} &+& u_{3,4}' x_4 &=& b_{3}',\ && && && u_{4,4}' x_4 &=& b_{4}'. \end{array} \end{eqnarray*}$$

which can be solved in $O(n^2)$ time!

## Gauss Elimination

Gauss Elimination proceeds in two phases. The first is elimination which transforms

$A x = b$

into

$U x = b'$

where $U$ is an upper triangular matrix and $b'$ is a modified vector of constants. The second phase is back-substitution which is trivial with triangular matricies.

### Gauss elimination algorithm

We first take

$Ax = b$

and build an *Augmented matrix*:

$Ab = [ A | b ]$

We are allowed the following operations which affect $|A|$ but not the solution to the problem:

|Operation | Effect on $|A|$ | |——|——| |Exchange 2 rows | Flips sign| |Multiply a row by § | Multiplied by § | |Subtract 2 rows | Unchanged|

### Steps from our previous example

```
import numpy as np
# Define A and b

A = np.array([[20, 50],
              [1, 1]])

b = np.array([[700], [20]])

#Form the augmented matrix A|b
Ab = np.hstack([A, b])

#The coefficient matrix can be separated by slicing Ab
def print_update():
  print("Augmented matrix is \n", Ab)
  print("Determinant of A: ", np.linalg.det(Ab[:,:-1])),
  print("Norm of A", np.linalg.norm(Ab[:,:-1], 'fro')),
  print("Condition number of A:", np.linalg.cond(Ab[:, :-1]))
  print("\n")

print("Step 0: Show the augmented matrix and the determinant of A")
print_update()

print("Step 1: Multiply the second row by 20")
Ab[1, :] = Ab[1, :] * 20
print_update()

print("Step 2: Subtract the second row from the first")
Ab[1, :] = Ab[1, :] - Ab[0, :]
print_update()
```

```
Step 0: Show the augmented matrix and the determinant of A
Augmented matrix is
 [[ 20  50 700]
 [  1   1  20]]
Determinant of A:  -29.99999999999999
Norm of A 53.87021440462252
Condition number of A: 96.72299453018881


Step 1: Multiply the second row by 20
Augmented matrix is
 [[ 20  50 700]
 [ 20  20 400]]
Determinant of A:  -600.0
Norm of A 60.8276253029822
Condition number of A: 6.000000000000001


Step 2: Subtract the second row from the first
Augmented matrix is
 [[  20   50  700]
 [   0  -30 -300]]
Determinant of A:  -600.0
Norm of A 61.644140029689765
Condition number of A: 6.171292729553326
```

Note: The determinant, norm and condition number are all changing despite the answer remaining the same. This is the basis of preconditioners!

**Algorithm**

Let's describe an algorithm for Gaussian elimination, as a sequence of passes row by row through the matrix.

Each pass we choose a *pivot row* which is used to eliminate the elements in other equations through multiplication of the pivot and subtraction.

Start from the top and only pass downwards, so we end up with an upper triangular matrix.

**Example**

Use Gauss Elimination to solve the following equations.

\begin{eqnarray*} 4x_1 + 3x_2 - 5x_3 &=& 2 \ -2x_1 - 4x_2 + 5x_3 &=& 5 \ 8x_1 + 8x_2 &=& -3 \ \end{eqnarray*}

Step 1: Turn these equations to matrix form \(Ax=y\).

\[\begin{split} \begin{bmatrix} 4 & 3 & -5\\ -2 & -4 & 5\\ 8 & 8 & 0\\ \end{bmatrix}\left[\begin{array}{c} x_1 \\x_2 \\x_3 \end{array}\right] = \left[\begin{array}{c} 2 \\5 \\-3\end{array}\right]\end{split}\]
Step 2: Get the augmented matrix [A, y]

\[\begin{split} [A, y] = \begin{bmatrix} 4 & 3 & -5 & 2\\ -2 & -4 & 5 & 5\\ 8 & 8 & 0 & -3\\ \end{bmatrix}\end{split}\]
Step 3: Choose the first equation as the pivot equation and turn the 2nd row first element to 0. To do this, we can multiply -0.5 for the 1st row (pivot equation) and subtract it from the 2nd row. The multiplier is \(m_{2,1}=-0.5\). We will get

\[\begin{split} \begin{bmatrix} 4 & 3 & -5 & 2\\ 0 & -2.5 & 2.5 & 6\\ 8 & 8 & 0 & -3\\ \end{bmatrix}\end{split}\]
Step 4: Turn the 3rd row first element to 0. We can do something similar, multiply 2 to the 1st row and subtract it from the 3rd row. The multiplier is \(m_{3,1}=2\). We will get

\[\begin{split} \begin{bmatrix} 4 & 3 & -5 & 2\\ 0 & -2.5 & 2.5 & 6\\ 0 & 2 & 10 & -7\\ \end{bmatrix}\end{split}\]
Step 5: Turn the 3rd row 2nd element to 0. We can multiple -4/5 for the 2nd row, and subtract it from the 3rd row. The multiplier is \(m_{3,2}=-0.8\). We will get

\[\begin{split} \begin{bmatrix} 4 & 3 & -5 & 2\\ 0 & -2.5 & 2.5 & 6\\ 0 & 0 & 12 & -2.2\\ \end{bmatrix}\end{split}\]
Elmination is now complete since \(A\) is upper triangular.

Proceed with substitution:

Step 6: Therefore, we can get \(x_3=-2.2/12=-0.183\).

Step 7: Insert \(x_3\) to the 2nd equation, we get \(x_2=-2.583\)

Step 8: Insert \(x_2\) and \(x_3\) to the first equation, we have \(x_1=2.208\).

#### Complexity

Considering the two phases of Gauss elimination:

Elimination: ~\(n^3/3\) operations, therefore, \(O(n^3)\)

Back substitution: ~ \(n^2/2\) operations, therefore \(O(n^2)\)

##Gauss-Jordan Elimination An obvious extension is to conduct each pass both upwards and downwards (G.E. is just down). While doing this we also normalize the row to get reduced row echelon form (abbreviated: rref):

\[\begin{split} \begin{bmatrix} 1 & 0 & 0 & 2.208\\ 0 & 1 & 0 & -2.583\\ 0 & 0 & 1 & -0.183\\ \end{bmatrix}\end{split}\]

Since \(I^{-1} = I\), the answer is just the right hand vector.

#### Complexity

Gauss-Jordan Elimination eliminates the need for back-substitution so one might think that it is more efficient than Gauss Elimination. Unfortunatley, the elimination phase takes ~\(~n^3/2\) operations whereas GE took ~\(n^3/3 + n^2/2\). therefore Gauss Elimination is preferred.

### Package implementation

Gaussian Elimination is fundamental but has largely been surpased by matrix decomposition (We will see the connection soon). Therefore, you will be hard pressed to find a standard implemention in numperical packages (numpy / scipy).

It is still useful in symbolic manipulation, and indeed you will find it in *sympy* under *echelon form* and *rref*:

```
#prompt: Use sympy rref and echelon form to make an example for linear system

from sympy import Matrix, symbols, pprint

a = symbols('a')

# Define the coefficient matrix and the right-hand side vector
A = Matrix([[1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]])
b = Matrix([2, 5, -3])

# Combine the coefficient matrix and the right-hand side vector into an augmented matrix
Ab = A.row_join(Matrix(b))

# Calculate the reduced row echelon form (rref) of the augmented matrix
Ab_rref = Ab.rref()

# Print the result
print("Augmented matrix in rref:")
pprint(Ab_rref[0])

# Calculate the echelon form of the coefficient matrix
Ab_echelon = Ab.echelon_form()

# Print the result
print("\nCoefficient matrix in echelon form:")
pprint(Ab_echelon)
```

```
Augmented matrix in rref:
⎡1  0  -1  0⎤
⎢           ⎥
⎢0  1  2   0⎥
⎢           ⎥
⎣0  0  0   1⎦

Coefficient matrix in echelon form:
⎡1  2   3   2 ⎤
⎢             ⎥
⎢0  -3  -6  -3⎥
⎢             ⎥
⎣0  0   0   33⎦
```

## Pivoting and diagonal dominance

The order of equations, and therefore rows in the matrix, is obviously arbitrary, but this may break Gauss Elimination.

E.g.: a system with augmented matrix

\[\begin{split} \begin{bmatrix} 2 & -1 & 0 & 1\\ -1 & 2 & -1 & 0\\ 0 & -1 & 1 & 0\\ \end{bmatrix}\end{split}\]
will work with our GE scheme, but the same system reordered,

\[\begin{split} \begin{bmatrix} 0 & -1 & 1 & 0\\ -1 & 2 & -1 & 0\\ 2 & -1 & 0 & 1\\ \end{bmatrix}\end{split}\]
will fail due to the \(0\) in the element of the first row. The remedy is to swap rows 3 and 1 to match the first example.

###Permutation matrix Swapping rows in matrix algebra is achieved via a *permutation matrix* (row swap of the *identity matrix*).

To swap rows 3 and 1, we would use:

\[\begin{split} P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ \end{bmatrix}\end{split}\]

### Diagonal dominance

More generally, if the diagonal element is *small* we will encounter roundoff error:

\[\begin{split} \begin{bmatrix} \epsilon & -1 & 1 & 0\\ -1 & 2 & -1 & 0\\ 2 & -1 & 0 & 1\\ \end{bmatrix}\end{split}\]
after the first pass leads to:

\[\begin{split} \begin{bmatrix} \epsilon & -1 & 1 & 0\\ -1 & 2 -1/\epsilon & -1 + 1/\epsilon & 0\\ 2 & -1 -1/\epsilon& + 1/\epsilon & 1\\ \end{bmatrix}\end{split}\]
from which we immediately see the danger as \(\epsilon \rightarrow 0\)!

Thus, we don't want the diagonal to have \(0\)s or *small* numbers. This motivates the definition of *diagonal dominance*.

A matrix is said to be *diagonally dominant* if each diagonal is larger, in absolute value, than the *sum of the other elements in the row*.

\(|A_{ii}| \ge \sum_{j=1,j \ne i}^{n} |A_{ij}|\)

Example: Which is diagonally dominant?

a) $\( \begin{bmatrix} -2 & 4 & -1 \\ -1 & -1 & 3 \\ 4 & -2 & 1 \\ \end{bmatrix}\)$

b) $\( \begin{bmatrix} 4 & -2 & 1 \\ -2 & 4 & -1 \\ -1 & -1 & 3 \\ \end{bmatrix}\)$

### Importance of diagonal dominance

It can be shown that if a matrix is diagonally dominant, *it will not benefit from pivoting!*.

Diagonal dominance contributes generally to numerical stability and accuracy through minimizing roundoff error propogation. This result extends beyond Gauss elimination, and will also be a criteria for matrix factorization methods and the efficient convergence of iterative methods.


## LU decomposition

Commonly we will have to repeatedly solve \(Ax = b\) for multiple \(b_i\). Gauss Elimination for each \(b_i\) would be grossly inefficient. If you knew all the \(b_i\) in advance you could do this in parallel by forming the augmented matrix:

\([A|b_1 \ b_2 \ b_3 \ ...]\)

but this is seldom the case.

It is much more efficient to decompose the matrix \(A\) into a form that is easier to solve.

> There are other reasons to do this for special matrix types and distributed computing which we will discuss later.

We have actually already seen this efficiency boost with back-substitution. The equation \(U x = b\) solves in \(O(n^2)\).

Any square matrix can be decomposed,

\(A = LU\)

where:

\(L\) is a lower triangular matrix

\(U\) is an upper triangular matrix

Now, the linear system becomes:

\[\begin{split} \begin{align} Ax &= b \\ LUx &= b \end{align} \end{split}\]

Now let \(y = Ux\), such that

\[\begin{split}\begin{align} Ly &= b \\ Ux &= y \end{align} \end{split}\]

both of which solve in \(O(n^2)\).

NOTE: L and U are generally *not unique*.

Example: Return to the previouis example:

\begin{align} 4x_1 + 3x_2 - 5x_3 &=& 2 \ -2x_1 - 4x_2 + 5x_3 &=& 5 \ 8x_1 + 8x_2 &=& -3 \ \end{align}

Through Gaussian Elimination, we found

\[\begin{split} U= \begin{bmatrix} 4 & 3 & -5 \\ 0 & -2.5 & 2.5 \\ 0 & 0 & 12 \\ \end{bmatrix}\end{split}\]

by clearing the first column by multiplying the first row by \(-0.5\) for the second row, and \(2\) for the third. The second column was cleared with the second row multiplied by \(-0.8\). These coefficients turn out to be the elemements of the \(L\) matrix (with 1's along the diagonal)!

\[\begin{split} L= \begin{bmatrix} 1 & 0 & 0 \\ -0.5 & 1 & 0 \\ 2 & -0.8 & 1 \\ \end{bmatrix}\end{split}\]

Let's verify:

```
# prompt: Do decomposition on the above matrix

import numpy as np

# Define the matrix A
A = np.array([[4, 3, -5],
              [-2, -4, 5],
              [8, 8, 0]])

print("The original matrix A is:\n", A, "\n")
L = np.array([[1,0,0],
               [-.5, 1,0],
               [2,-.8,1]])

U = np.array([[4,3,-5],
              [0,-2.5,2.5],
              [0,0,12]])

print("The reconstructed matrix is:\n", L@U)
```

```
The original matrix A is:
 [[ 4  3 -5]
 [-2 -4  5]
 [ 8  8  0]]

The reconstructed matrix is:
 [[ 4.  3. -5.]
 [-2. -4.  5.]
 [ 8.  8.  0.]]
```

Let's check the package decomposition!

```
# Calculate the LU decomposition
from scipy.linalg import lu, inv
P, L, U = lu(A)

print("Permutation Matrix (P):\n", P)
print("Lower Triangular Matrix (L):\n", L)
print("Upper Triangular Matrix (U):\n", U)



print("\nMultiply L and U:\n", L@U, "\nwhich is correct but pivoted!")

print("\nMultiply PLU:\n", P@L@U, "\nwhich is the original matrix!")
```

```
Permutation Matrix (P):
 [[0. 0. 1.]
 [0. 1. 0.]
 [1. 0. 0.]]
Lower Triangular Matrix (L):
 [[ 1.    0.    0.  ]
 [-0.25  1.    0.  ]
 [ 0.5   0.5   1.  ]]
Upper Triangular Matrix (U):
 [[ 8.   8.   0. ]
 [ 0.  -2.   5. ]
 [ 0.   0.  -7.5]]

Multiply L and U:
 [[ 8.  8.  0.]
 [-2. -4.  5.]
 [ 4.  3. -5.]]
which is correct but pivoted!

Multiply PLU:
 [[ 4.  3. -5.]
 [-2. -4.  5.]
 [ 8.  8.  0.]]
which is the original matrix!
```

NB: $P$ in the above is the permutation matrix that, when multiplied by LU recovers the original matrix. It is *not* the pivoting operation that is done internally (although that matrix is easily obtained!).

## Dr. Mike's Tips!

- Direct solver are your 'black box' for most of your needs.
- They are the most robust for ill-conditioned systems.
- They scale *terribly* (both in system size and parallelization)
- If you use them, start with a small system and work upwards.
- Generally speaking you won't see a speedup with parallelization until you get a large # of nodes
- Warning: Some implementations (numpy) are sophisticated enough to handle singular matricies as well as non-singular (be careful with the answer!)
- Sparse matricies are your saving grace! Do your best to protect them (hence store the LU factors, not the inverse!)

## Matrix types

LU decomposition has useful properties depending on the types of matricies involved.

| Symbol | Matrix Type | Example 2x2 | Interesting Properties |
|---|---|---|---|
| 0 | Zero | $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ | A-A = 0 |
| I | Identity | $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ | A I = A |
| D | Diagonal | $\begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix}$ | $D^{-1}$ = $\begin{bmatrix} d_1^{-1} & 0 \\ 0 & d_2^{-1} \end{bmatrix}$ |
| U | Upper Triangular | $\begin{bmatrix} 1 & 2 \\ 0 & 4 \end{bmatrix}$ | $U^{-1}$ is another upper triangular matrix |
| L | Lower Triangular | $\begin{bmatrix} 5 & 0 \\ -1 & 2 \end{bmatrix}$ | $L^{-1}$ is another lower triangular matrix |
| P | Permutation | $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ | Its transpose is its inverse ($P^T = P^{-1}$) |
| S | Symmetric | $\begin{bmatrix} 1 & 3 \\ 3 & -4 \end{bmatrix}$ | Equal to its transpose ($S = S^T$). |
| SPD | Symmetric positive definite | $\begin{bmatrix} 1 & 3 \\ 3 & 4 \end{bmatrix}$ | $x^T [SPD] x >0$ - think of a quadratic |

### Sparse matricies and sparsity patterns

Sparsity refers to the fraction of non-zero elements in a matrix. Sparse matricies have a significant fraction of 0 elements, which can **drastically** reduce storage and computational requirements.

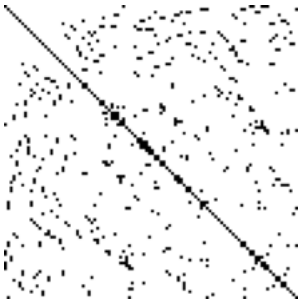
Storage: Typically one (conceptually) stores a matrix as a 2D array. With significant sparsity, one can instead store only non-zero entries. There is a trade-off here between access / modification and memory efficiency.

Computation: (Fill-in) Fill-in is the phenomena where, during computation, zeros become non-zeros. This disrupts the sparsity pattern and the benefits that accompany it, and is hence something to be avoided.

## Sparsity patterns and matrix

Sparsity patterns are condensly viewed as monochrome matricies filled in according to 0 or non-zero.

Example: A sparsity pattern from a finite element problem in 2D.



We already have examples of sparse matricies with

- diagonal
- upper and lower triangular and have seen their importance in controlling complexity and roundoff error.

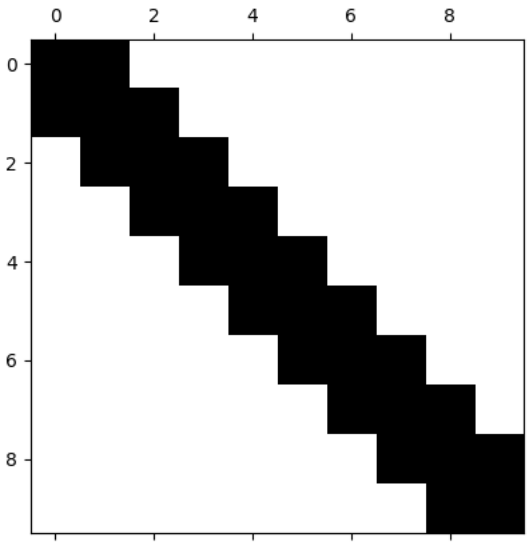
### Banded matricies

Banded matricies only have non-zero elements parallel to the diagonal. Sometimes they are described by the *bandwidth*, the number of parallel bands.

### Tridiagonal (bandwidth 3)

```
# prompt: give me a tridiagonal matrix of dimension 10 and view its sparsity pattern with spy

import numpy as np
import matplotlib.pyplot as plt

# Create a tridiagonal matrix of dimension 10
n = 10
T = np.zeros((n, n))
np.fill_diagonal(T, 2)  # Main diagonal
np.fill_diagonal(T[1:], -1)  # Lower diagonal
np.fill_diagonal(T[:, 1:], -1)  # Upper diagonal

print(T)

# View the sparsity pattern
plt.spy(T)
plt.show()
```

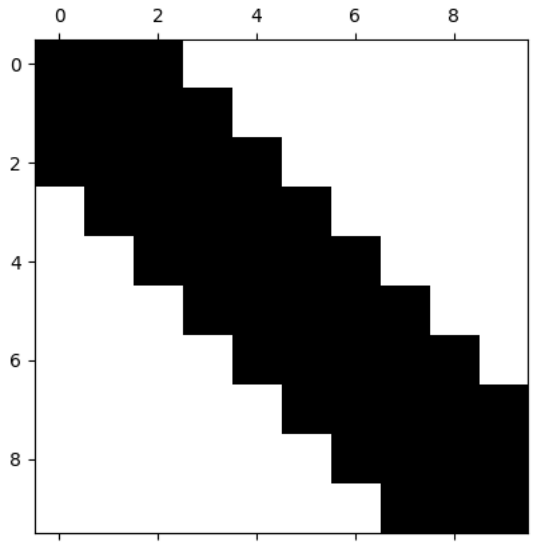
```
[[ 2. -1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  2. -1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. -1.  2. -1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  2. -1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. -1.  2. -1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -1.  2. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0. -1.  2. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1.  2. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. -1.  2. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -1.  2.]]
```




**Pentadiagonal (bandwidth 5)**

```python
# prompt: give me a pentadiagonal matrix of dimension 10 and view its sparsity pattern with spy

import numpy as np
import matplotlib.pyplot as plt

# Create a pentadiagonal matrix of dimension 10
n = 10
A = np.zeros((n, n))
np.fill_diagonal(A, 2)  # Main diagonal
np.fill_diagonal(A[1:], -1)  # Upper diagonal
np.fill_diagonal(A[:, 1:], -1)  # Lower diagonal
np.fill_diagonal(A[2:], -1)  # Second upper diagonal
np.fill_diagonal(A[:, 2:], -1)  # Second lower diagonal

print(A)

# View the sparsity pattern
plt.spy(A)
plt.show()
```

```
[[ 2. -1. -1.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  2. -1. -1.  0.  0.  0.  0.  0.  0.]
 [-1. -1.  2. -1. -1.  0.  0.  0.  0.  0.]
 [ 0. -1. -1.  2. -1. -1.  0.  0.  0.  0.]
 [ 0.  0. -1. -1.  2. -1. -1.  0.  0.  0.]
 [ 0.  0.  0. -1. -1.  2. -1. -1.  0.  0.]
 [ 0.  0.  0.  0. -1. -1.  2. -1. -1.  0.]
 [ 0.  0.  0.  0.  0. -1. -1.  2. -1. -1.]
 [ 0.  0.  0.  0.  0.  0. -1. -1.  2. -1.]
 [ 0.  0.  0.  0.  0.  0.  0. -1. -1.  2.]]
```

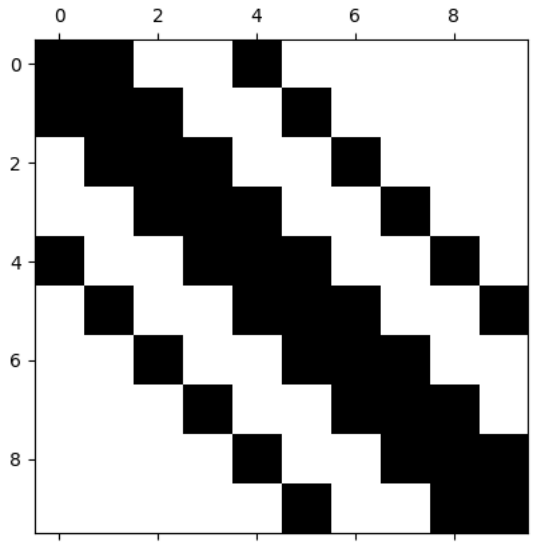



One can imagine the benefit of storing such a matrix

#### General banded matrix

```python
import numpy as np
import matplotlib.pyplot as plt

# Create a pentadiagonal matrix of dimension 10
n = 10
A = np.zeros((n, n))
np.fill_diagonal(A, 2)  # Main diagonal
np.fill_diagonal(A[1:], -1)  # Upper diagonal
np.fill_diagonal(A[:, 1:], -1)  # Lower diagonal
np.fill_diagonal(A[4:], -2)  # Second upper diagonal
np.fill_diagonal(A[:, 4:], -1)  # Second lower diagonal

print(A)

# View the sparsity pattern
plt.spy(A)
plt.show()
```

```
[[ 2. -1.  0.  0. -1.  0.  0.  0.  0.  0.]
 [-1.  2. -1.  0.  0. -1.  0.  0.  0.  0.]
 [ 0. -1.  2. -1.  0.  0. -1.  0.  0.  0.]
 [ 0.  0. -1.  2. -1.  0.  0. -1.  0.  0.]
 [-2.  0.  0. -1.  2. -1.  0.  0. -1.  0.]
 [ 0. -2.  0.  0. -1.  2. -1.  0.  0. -1.]
 [ 0.  0. -2.  0.  0. -1.  2. -1.  0.  0.]
 [ 0.  0.  0. -2.  0.  0. -1.  2. -1.  0.]
 [ 0.  0.  0.  0. -2.  0.  0. -1.  2. -1.]
 [ 0.  0.  0.  0.  0. -2.  0.  0. -1.  2.]]
```




## Block matricies

Block matricies can be partitioned:

$$\begin{split} M= \begin{bmatrix} 2 & -1 & 0 & 1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 1 & 0 \\ 0 & -1 & 2 & 0 & 0 & 1 \\ 1 & 0 & 0 & 2 & -1 & 0 \\ 0 & 1 & 0 & -1 & 2 & -1 \\ 0 & 0 & 1 & 0 & -1 & 2 \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \end{split}$$

with

$$\begin{split} A=D = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \\ \end{bmatrix} \end{split}$$

and

$$\begin{split} B=C = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \end{bmatrix} \end{split}$$

The inverse can be formed:

$$\begin{split} M^{-1} = \begin{bmatrix} A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix} \end{split}$$

This is a substantial savings since the cost of each block scales exponentially!

Sometimes we get lucky and one of the blocks is $0$ or $I$ which is a HUGE benefit!

If the blocks are each $n \times n$, $M$ is $2n \times 2n$ and takes $O((2n)^3)$ to solve. But if we partition into blocks, the inverse can be formed in 2 separate $O(n^3)$ (plus matrix multiplications).

This situation arises commonly in multiphysics applications, with two fields (e.g.: T and c). The blocks are then defined corresponding to the unknown vectors for each field.

Certaint types of physics is solved more efficiently with certain solvers. This decomposition, sometimes called *operator splitting* or *segregated solving* allows each variable (block) to be treated optimally and inverse of the full system assembled.

In contrast, solving the matrix as a whole is called a *monolithic* solution.

#### Block diagonal

Taking $B=C=0$ in the above we can see the inverse of a *block diagonal* matrix is simply the diagonal assembly of the inverse of its blocks:

$$\begin{split} \begin{bmatrix} A & 0 \\ 0 & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} & 0 \\ 0 & D^{-1} \end{bmatrix} \end{split}$$

## Matrix decompositions

We can now summarize some decompositions:

| Matrix equation | Operation count |
|---|---|
| $A = LU$ | $n^3/3$ |
| $S = L D L^T$ | $n^3/6$ |
| $SPD = L L^T$ | $n^3/6$ + n |

The later two are called Choleski decomposition and take advantage of matrix symmetry for efficiency.

Since in general

$A^{-1} = [LU]^{-1} = U^{-1} L^{-1}$

it would be tempting to calculate and store $A^{-1}$ instead of $L$ and $U$, but this is not preferred due to the impact on sparsity, in addition to round-off error $\propto |A|^{-1}$.

##Decompositions of certain matricies

### Inverse of a triangular matrix is triangular

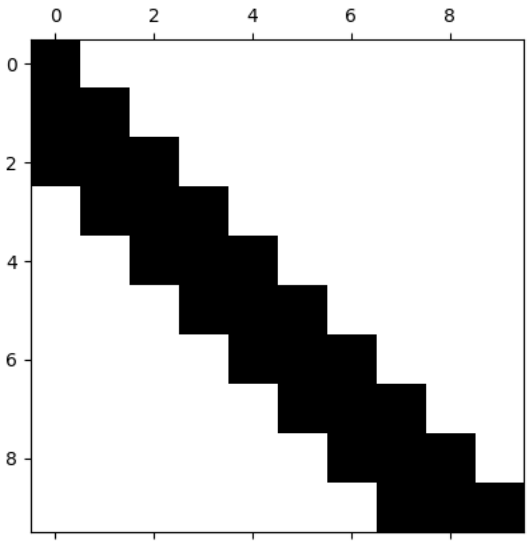
```
# prompt: show the inverse of an upper triangular matrix with numpy

import numpy as np
# Create an upper triangular matrix
A = np.array([[1, 2, 3],
              [0, 4, 5],
              [0, 0, 6]])

# Calculate the inverse
A_inv = np.linalg.inv(A)

print("Upper triangular matrix:\n", A)
print("\nInverse of the upper triangular matrix:\n", A_inv)
```

```
Upper triangular matrix:
 [[1 2 3]
 [0 4 5]
 [0 0 6]]

Inverse of the upper triangular matrix:
 [[ 1.         -0.5        -0.08333333]
 [ 0.          0.25       -0.20833333]
 [ 0.          0.          0.16666667]]
```

### LU decomposition of a banded matrix is banded-triangular

```python
# prompt: Show that the choleski decomposition of a banded matrix is triangular-banded

import numpy as np
from scipy.linalg import cholesky
import matplotlib.pyplot as plt

# Create a pentadiagonal matrix of dimension 10
n = 10
A = np.zeros((n, n))
np.fill_diagonal(A, 2)  # Main diagonal
np.fill_diagonal(A[1:], 1)  # Upper diagonal
np.fill_diagonal(A[:, 1:], 1)  # Lower diagonal
np.fill_diagonal(A[2:], .1)  # Second upper diagonal
np.fill_diagonal(A[:, 2:], .1)  # Second lower diagonal

# Calculate the Cholesky decomposition
L = cholesky(A, lower=True)

print("Original matrix:\n", A)
print("\nCholesky decomposition (L):\n", np.around(L, decimals=2))

# View the sparsity pattern
plt.spy(L)
plt.show()
```

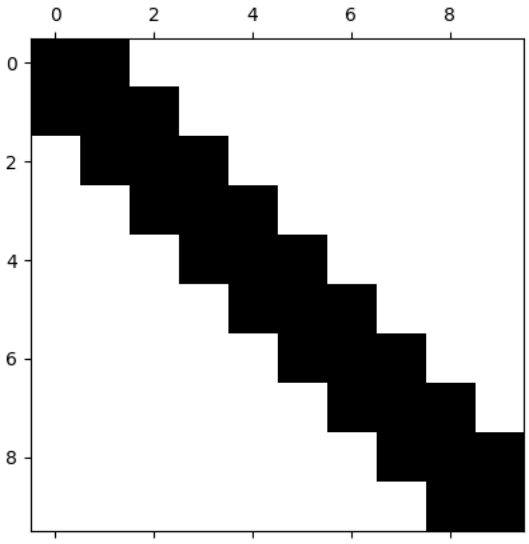
```
Original matrix:
 [[2.  1.  0.1 0.  0.  0.  0.  0.  0.  0. ]
 [1.  2.  1.  0.1 0.  0.  0.  0.  0.  0. ]
 [0.1 1.  2.  1.  0.1 0.  0.  0.  0.  0. ]
 [0.  0.1 1.  2.  1.  0.1 0.  0.  0.  0. ]
 [0.  0.  0.1 1.  2.  1.  0.1 0.  0.  0. ]
 [0.  0.  0.  0.1 1.  2.  1.  0.1 0.  0. ]
 [0.  0.  0.  0.  0.1 1.  2.  1.  0.1 0. ]
 [0.  0.  0.  0.  0.  0.1 1.  2.  1.  0.1]
 [0.  0.  0.  0.  0.  0.  0.1 1.  2.  1. ]
 [0.  0.  0.  0.  0.  0.  0.  0.1 1.  2. ]]

Cholesky decomposition (L):
 [[1.41 0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.71 1.22 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.07 0.78 1.18 0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.08 0.79 1.17 0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.08 0.8  1.16 0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.09 0.8  1.16 0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.09 0.8  1.16 0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.09 0.8  1.16 0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.09 0.8  1.16 0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.09 0.8  1.16]]
```



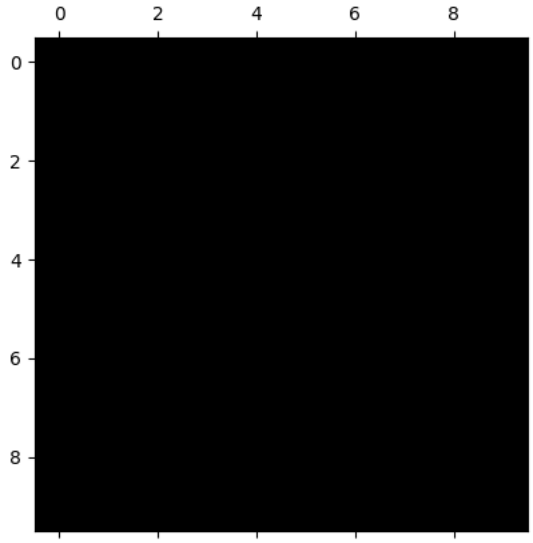

## Inverse of a banded matrix is not banded (except diagonal)

```python
# prompt: show the inverse of a banded matrix in numpy

import numpy as np

# Create a tridiagonal matrix of dimension 10
n = 10
A = np.zeros((n, n))
np.fill_diagonal(A, 2)  # Main diagonal
np.fill_diagonal(A[1:], -1)  # Upper diagonal
np.fill_diagonal(A[:, 1:], -1)  # Lower diagonal

# Calculate the inverse
A_inv = np.linalg.inv(A)

print("Banded matrix:\n")
print(A)
plt.spy(A)
plt.show()
print("\nInverse of the banded matrix:\n")
print(A_inv)
plt.spy(A_inv)
plt.show()
```

```
Banded matrix:

[[ 2. -1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  2. -1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. -1.  2. -1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  2. -1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. -1.  2. -1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -1.  2. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0. -1.  2. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1.  2. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. -1.  2. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -1.  2.]]
```




```
Inverse of the banded matrix:

[[0.90909091 0.81818182 0.72727273 0.63636364 0.54545455 0.45454545
  0.36363636 0.27272727 0.18181818 0.09090909]
 [0.81818182 1.63636364 1.45454545 1.27272727 1.09090909 0.90909091
  0.72727273 0.54545455 0.36363636 0.18181818]
 [0.72727273 1.45454545 2.18181818 1.90909091 1.63636364 1.36363636
  1.09090909 0.81818182 0.54545455 0.27272727]
 [0.63636364 1.27272727 1.90909091 2.54545455 2.18181818 1.81818182
  1.45454545 1.09090909 0.72727273 0.36363636]
 [0.54545455 1.09090909 1.63636364 2.18181818 2.72727273 2.27272727
  1.81818182 1.36363636 0.90909091 0.45454545]
 [0.45454545 0.90909091 1.36363636 1.81818182 2.27272727 2.72727273
  2.18181818 1.63636364 1.09090909 0.54545455]
 [0.36363636 0.72727273 1.09090909 1.45454545 1.81818182 2.18181818
  2.54545455 1.90909091 1.27272727 0.63636364]
 [0.27272727 0.54545455 0.81818182 1.09090909 1.36363636 1.63636364
  1.90909091 2.18181818 1.45454545 0.72727273]
 [0.18181818 0.36363636 0.54545455 0.72727273 0.90909091 1.09090909
  1.27272727 1.45454545 1.63636364 0.81818182]
 [0.09090909 0.18181818 0.27272727 0.36363636 0.45454545 0.54545455
  0.63636364 0.72727273 0.81818182 0.90909091]]
```




Yikes!!! :-(

#Package implementations

Numpy and Scipy linear solvers both do (P)LU decomposition and then solution. In general they can be accessed through:

> numpy.linalg.solve(A,b)

and

> scipy.linalg.solve(A,b)

Examining the options of scipy show that you can choose to:

1. specify the matrix type
2. overwrite the original matricies (which may boost efficiency)

The advent of distributed computing (HPCs) motivated new algorithms that are better suited to large, sparse, systems (You will run in to these names in the future!)

- PARADISO (PARallel Direct SOlver)
- SuperLU (Supernodal LU)
- UMFPACK (Unsymmetric-pattern MultiFrontal method)
- MUMPS (Multifrontal Massively Parallel Sparse Direct Solver)

```
# prompt: use numpy to solve a linear system then repeat with scipy

import numpy as np
from scipy.linalg import solve

# Define the coefficient matrix A
A = np.array([[2, -1, 0],
              [-1, 2, -1],
              [0, -1, 2]])

# Define the right-hand side vector b
b = np.array([1, 2, 3])

# Solve the linear system using numpy.linalg.solve
x_numpy = np.linalg.solve(A, b)
print("Solution using numpy.linalg.solve:\n", x_numpy)

# Solve the linear system using scipy.linalg.solve
x_scipy = solve(A, b)
print("\nSolution using scipy.linalg.solve:\n", x_scipy)
```

```
Solution using numpy.linalg.solve:
 [2.5 4.  3.5]

Solution using scipy.linalg.solve:
 [2.5 4.  3.5]
```

```
# prompt: solve a 200x200 sparse system with sparse solvers then again  with a dense solver

import numpy as np
from scipy.sparse import diags
from scipy.sparse.linalg import spsolve
from scipy.linalg import solve

# Generate a random sparse matrix
n = 200

main_diag = np.full(n, 2)
upper_diag = np.full(n - 1, -1)
lower_diag = np.full(n - 1, -1)
A_sparse = diags([lower_diag, main_diag, upper_diag], offsets=[-1, 0, 1], format='csr')


# row_ind = np.arange(n)
# col_ind = np.arange(n)
# data = np.random.rand(n)
# A_sparse = csr_matrix((data, (row_ind, col_ind)), shape=(n, n))

# Generate a random right-hand side vector
b = np.random.rand(n)

# Solve the sparse system
%timeit x_sparse = spsolve(A_sparse, b)
#print("Solution using sparse solver:\n", x_sparse, "\n")

# Convert the sparse matrix to a dense matrix
A_dense = A_sparse.toarray()

# Solve the dense system

%timeit x_dense = solve(A_dense, b)
#print("Solution using dense solver:\n", x_dense)
```

```
307 µs ± 142 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
1.29 ms ± 101 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

## Iterative methods

Guass Elimination / LU decomposition are *direct* solvers in that they solve systems in a fixed number of operations. If we could calculate in infinite precision these answers would be exact. Direct solvers generally scale ~$O(n^3)$ and are therefore

impractical for large systems.

We will now talk about *iterative / indirect methods* which start with an initial guess and iterate (hopefully) towards a solution. Each iteration is typically faster and more memory efficient than direct solvers, but convergence is only guaranteed under certain limited circumstances. Even then, the *rate of convergence* can still make these techniques impractical.

Iterative techniques have some useful properties:

1. They are self-correcting in that roundoff (or arithmatic) errors are corrected through further cycles.
2. They are more conducive to sparse matrix storage.

Only the solution vector $x$ is altered during these computations. Neither the matrix (nor anything we will do to it) is not altered during any of these steps!

> In fact, there are a class of solvers called *matrix free* solvers that avoid the explicit writing of $A$ entirely… This is useful if the matrix product $Ax$ can be calculated in a more effective manner such that we don't bother writing out and storing A$!

Each iteration will (hopefully) improve the answer asymptotically, so we estimate the error and say when it is stopped getting better. As discussed with error analysis, this involves defining tolerances for computation.

## Iterative approach

Consider: What does it mean for a vector $x$ to satisfy $Ax=b$?

-> All the elements in $x$ must be *consistent*.

What if one element was not consistent?

-> We could solve for it based on the others!

## Big picture algorithm

1. Start with a guess
2. For each iterative step, identify a single element in $x$, make it consistent with all the others.
3. Check how well the guess solves the linear system
4. Repeat (possibly updating a parameter first)

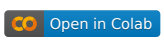
### Quantifying convergence

The system has converged when $Ax=b$. Define the *residual vector*, $R = Ax-b$ such that when $R\rightarrow 0$ (the system is converged).

Let's use the norm (some measure of magnitude) of $R$ and say the system has converged when $||R|| \lt tolerance$.

> Should we use absolute tolerance or relative?

> What other tolerance could we imagine using? (Hint: We are actually interested $x$…)




GOALS:

- Grasp the basic concept of Jacobi, Gauss-Seidel, and Successive Over Relaxation iterative methods

- Understand the pros and cons of each method
- Experience the benefit of iterative solvers

## Jacobi iteration method

Begin by writing out the linear system in index form:

$$\begin{split} \begin{align*} A x &= b \\ \sum_{j} A_{ij} x_j&=b_i \\ \begin{bmatrix} a_{1,1} & a_{1,2} & ... & a_{1,n}\\ a_{2,1} & a_{2,2} & ... & a_{2,n}\\ ... & ... & ... & ... \\ a_{m,1} & a_{m,2} & ... & a_{m,n} \end{bmatrix}\left[\begin{array}{c} x_1 \\x_2 \\ ... \\x_n \end{array}\right] &= \left[\begin{array}{c} b_1 \\b_2 \\ ... \\b_m \end{array}\right] \end{align*}\end{split}$$

Extract the $i$th row, and solve for the diagonal term:

$$\begin{split}\begin{aligned} A_{ii}x_i^{k+1} &+ \sum_{j \ne i} A_{ij} x_j^k = b_i \\ x_i^{k+1} &= \frac{1}{A_{ii}}\left[b_i-\sum_{j \ne i} A_{ij} x_j^k\right] \end{aligned}\end{split}$$

This makes $x_i^{k+1}$ consistent with all the other elements.

We can write this in matrix form by pulling appart $A$ along its diagonal, $D$, into the upper $U$ and lower $L$ matricies:

$A = L + D + U$

NB: This is not a matrix decomposition; we are literally just pulling it apart (blank elements are zeros):

$$\begin{split}\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} & & \\ a_{21} & & \\ a_{22} & a_{23} & 0 \end{bmatrix} + \begin{bmatrix} a_{11} & & \\ & a_{22} & \\ & & a_{33} \end{bmatrix} + \begin{bmatrix} & a_{12} & a_{13} \\ & & a_{23} \\ & & \end{bmatrix} \end{split}$$

The algorithm becomes: $\(\begin{align} D x^{k+1} &= b-[L+U]x^k \\ x^{k+1} &= D^{-1} \big[b-[L+U]x^k\big] \end{align} \)$

Note:

- Each new vector uses the *entire* previous one. We need to store both $x^k$ and $x^{k+1}$.
- More memory but parallelizes better!

```
# prompt: Write a code that take a single Jacobi iteration step
import numpy as np

def jacobi_step(A, b, x):
  """Performs a single Jacobi iteration step.

  Args:
    A: The coefficient matrix.
    b: The right-hand side vector.
    x: The current solution vector.

  Returns:
    The updated solution vector.
  """

  n = len(x)
  x_new = np.zeros_like(x)
  for i in range(n):
    x_new[i] = (b[i] - np.dot(A[i, :i], x[:i]) - np.dot(A[i, i + 1:], x[i + 1:])) / A[i, i]
  return x_new
```

```
# Test it!
A = np.array([[4, 1], [1, 3]])
b = np.array([5, 6])
x0 = np.array([0., 0.])

print('Residual for ', x0, ' is ', np.linalg.norm(A@x0-b))

x_new = jacobi_step(A, b, x0)
print('Residual for ', x_new, ' is ', np.linalg.norm(A@x_new-b))

print('The true answer is ', np.linalg.solve(A, b))
```

```
Residual for  [0. 0.]  is  7.810249675906654
Residual for  [1.25 2.  ]  is  2.358495283014151
The true answer is  [0.81818182 1.72727273]
```

## Gauss-Seidel

Gauss-Seidel uses the same concept except that the next guess for $x_i$ is based on the *current* version of $x$. I.e.: If we are moving downwards through the rows, the $i$th element is updated based on the updated rows above.

$$\begin{aligned} x_i^{k+1} &= \frac{1}{A_{ii}}\left[b_i-\sum_{j=1}^{i-1} A_{ij} x_j^{k+1} -\sum_{j=i+1}^n A_{ij} x_j^k\right] \end{aligned}$$

This can again be written in matrix form: $

$\begin{aligned} A_{ii}x_i^{k+1} + \sum_{j=1}^{i-1} A_{ij} x_j^{k+1} &= b_i-\sum_{j=i+1}^n A_{ij} x_j^k\\ [D + L] x^{k+1} &= b - Ux^{k} \end{aligned}$

$

Gauss-Seidel is conceptually easier to implement since if we modify $x$ in place, each $x_i^{k+1}$ is built from all the other *current elements*! We can drop the iteration superscript and write (note the arrow for assignment):

$$\begin{aligned} x_i \leftarrow \frac{1}{A_{ii}}\left[b_i-\sum_{j \ne i} A_{ij} x_j \right] \end{aligned}$$

Notes:

- Each iteration includes the previously updated elements.
- We can just store 1 vector and update *in-place*.
- Less memory but doesn't parallelize as well.

```python
# prompt: Write a code like above that takes a single Gauss-sidel step

import numpy as np
def gauss_seidel_step(A, b, x, omega = 1):
  """Performs a single Gauss-Seidel iteration step.

  Args:
    A: The coefficient matrix.
    b: The right-hand side vector.
    x: The current solution vector.

  Returns:
    The updated solution vector.
  """

  n = len(x)
  for i in range(n):
    x[i] = (b[i] - np.dot(A[i, :i], x[:i]) - np.dot(A[i, i + 1:], x[i + 1:])) / A[i, i]
  return x
```

```python
# Test it!
A = np.array([[4, 1], [1, 3]])
b = np.array([5, 6])
x0 = np.array([0., 0.])

print('Residual for ', x0, ' is ', np.linalg.norm(A@x0-b))

x_new = gauss_seidel_step(A, b, x0.copy())
print('Residual for ', x_new, ' is ', np.linalg.norm(A@x_new-b))

print('The true answer is ', np.linalg.solve(A, b))
```

```
Residual for  [0. 0.]  is  7.810249675906654
Residual for  [1.25       1.58333333]  is  1.583333333333333
The true answer is  [0.81818182 1.72727273]
```

## Successive Over Relaxation (SOR)

The convergence of Gauss-Sidel can be improved through *relaxation*: Let's take a weighted average of the new and old $x_i^{k+1}$:

$$\begin{split}\begin{aligned} x_i^{k+1} &= \frac{\omega}{A_{ii}}\left[b_i-\sum_{j=1}^{i-1} A_{ij} x_j^{k+1} -\sum_{j=i+1}^n A_{ij} x_j^k\right] + [1-\omega] x^k_i \\ x_i &\leftarrow \frac{\omega}{A_{ii}}\left[b_i-\sum_{j \ne i} A_{ij} x_j \right] + [1-\omega] x_i \end{aligned}\end{split}$$

Let's look at three cases:

- $\omega = 1$ the method is exactly Gauss-Seidel (therefore you will usually only find SOR out in the wild)
- $\omega\lt 1$ the method is *underrelaxed* and will converge / diverge more slowly. Hopefully this will lead to better (albeit slower) convergence…
- $\omega \gt 1$ the method is *over relaxed* and convergence / divergence will be accelarated!

In general, SOR with a well-chosen $\omega$ will outperform Gauss-Seidel, but choosing $\omega$ is not-trivial.

One option: Calculate the difference between successive iterations:

$\Delta x^k = ||x^{k}-x^{k-1})||$

After an integer $p$ more iterations, recalculate:

$\Delta x^{k+p} =||x^{k+p}-x^{k+p-1}||$

then the optimal $\omega$ can be found:

$\omega_{opt} \approx \frac{2}{1+\sqrt{1+\left[ \frac{\Delta x^{k+p}}{\Delta x^{k}} \right]^p }}$

```
# prompt: Write a code that uses the Gauss Seidel function inside a successive over relaxation function t

import numpy as np
def sor_step(A, b, x):
  """Performs a single SOR iteration step.

  Args:
    A: The coefficient matrix.
    b: The right-hand side vector.
    x: The current solution vector.
    omega: The relaxation factor.

  Returns:
    The updated solution vector.
  """
  omega = 1.05
  n = len(x)
  for i in range(n):
    x[i] = (1 - omega) * x[i] + (omega / A[i, i]) * (b[i] - np.dot(A[i, :i], x[:i]) - np.dot(A[i, i + 1:
  return x

#Test it!
A = np.array([[4, 1], [1, 3]])
b = np.array([5, 6])
x0 = np.array([0., 0.])

print('Residual for ', x0, ' is ', np.linalg.norm(A@x0-b))

x_new = gauss_seidel_step(A, b, x0.copy())
print('Residual for ', x_new, ' is ', np.linalg.norm(A@x_new-b))

print('The true answer is ', np.linalg.solve(A, b))
```

```
Residual for  [0. 0.]  is  7.810249675906654
Residual for  [1.25       1.58333333]  is  1.583333333333333
The true answer is  [0.81818182 1.72727273]
```

## Summary

- Jacobi, Gauss-Seidel, and SOR are all suitable for large, sparse matricies.
- Since the matrix, and therefore the L D U partitions don't change, terms can be precomputed outside of the iteration loop for computational efficiency.
- They generally are more memory efficient and faster but convergence is not guaranteed.
- Even with guaranteed convergence, convergence can be slow.

**Jacobi**

- Easily parallelized but slower convergence.
- Guaranteed to converge if $A$ is diagonally dominant

**Gauss-Seidel**

- Faster convergence but doesn't parallelize well.
- Guaranteed to converge if $A$ is diagonally dominant or symmetric ositive definite.

**Successive Over Relaxation**

- Generalization of Gauss-Seidel ($\omega = 1$).
- *May* converge faster.
- Success hinges on choice of $\omega$.
- Shares guaranteed convergence of Gauss-Seidel if $0 \lt \omega \lt 2$.

## Let's code!

Let's code in these methods and compare the results. (The python standard libraries don't have implementaitons).

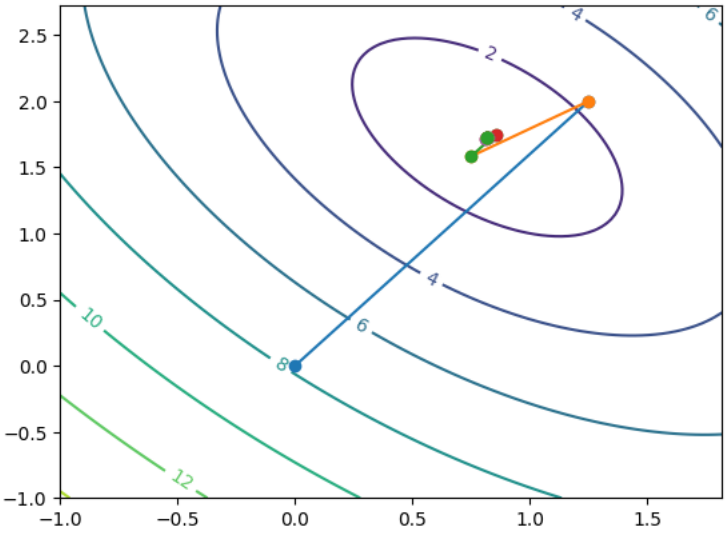
All the schemes:

1. Start with a guess, $A$ matrix and $b$ vector
2. Do *something* to find a new guess (using A and b)
3. Check how well the guess solves the linear system
4. Repeats (possibly updating a parameter first)

Define a wrapper that implements the above framework, passing a function to do each step. Let's plot what the system looks like and trace the estimation.

```
max_iter = 100  # ALWAYS HAVE A FAILSAFE WHEN YOU PLAY WITH INFINITE LOOPS!
tolerance = 1e-6
A = np.array([[4, 1], [1, 3]])
b = np.array([5, 6])
x0 = np.array([0., 0.])

x_true = np.linalg.solve(A, b)
print('The true answer is ', x_true)


# Utility function to autoscale the plot limits to include x0 and x_true
def calculate_box_limits(point1, point2, padding=1.0):
    # Extract coordinates
    x1, y1 = point1
    x2, y2 = point2
    # Calculate min and max coordinates with padding
    min_x = min(x1, x2) - padding
    max_x = max(x1, x2) + padding
    min_y = min(y1, y2) - padding
    max_y = max(y1, y2) + padding
    return (min_x, max_x, min_y, max_y)


xb, xe, yb, ye = calculate_box_limits(x0, x_true)
# Plot the linear system
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
x = np.linspace(xb, xe, 100)
y = np.linspace(yb, ye, 100)
X, Y = np.meshgrid(x, y)
V = np.stack((X,Y),-1)
Z = np.linalg.norm(np.matmul(V,A.T)-b, axis = -1)
fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z)
ax.clabel(CS, inline=True, fontsize=10)


method = jacobi_step
#method = gauss_seidel_step
#method = sor_step


for i in range(max_iter):
  x_new = method(A, b, x0.copy())
  R = np.linalg.norm(A@x_new-b)

  print('Iteration ', i, ' determined ', x_new, 'with residual', R)
  plt.plot([x0[0], x_new[0]], [x0[1], x_new[1]], '-o')
  if np.linalg.norm(R) < tolerance:
    print('Converged after, ', i, ' iterations to: ', x_new)
    break
  x0 = x_new.copy()
```

```
The true answer is  [0.81818182 1.72727273]
Iteration  0  determined  [1.25 2.  ] with residual 2.358495283014151
Iteration  1  determined  [0.75       1.58333333] with residual 0.650854139658888
Iteration  2  determined  [0.85416667 1.75      ] with residual 0.19654127358451298
Iteration  3  determined  [0.8125     1.71527778] with residual 0.05423784497157428
Iteration  4  determined  [0.82118056 1.72916667] with residual 0.016378439465376277
Iteration  5  determined  [0.81770833 1.72627315] with residual 0.004519820414298246
Iteration  6  determined  [0.81843171 1.72743056] with residual 0.0013648699554481016
Iteration  7  determined  [0.81814236 1.72718943] with residual 0.00037665170119081915
Iteration  8  determined  [0.81820264 1.72728588] with residual 0.00011373916295438507
Iteration  9  determined  [0.81817853 1.72726579] with residual 3.1387641766271184e-05
Iteration  10  determined  [0.81818355 1.72727382] with residual 9.47826357947717e-06
Iteration  11  determined  [0.81818154 1.72727215] with residual 2.6156368136569228e-06
Iteration  12  determined  [0.81818196 1.72727282] with residual 7.898552987055788e-07
Converged after,  12  iterations to:  [0.81818196 1.72727282]
```




## Example: Iteratively solve a large matrix

```python
# prompt: Form a large, sparse linear system with a diagonally dominant random matrix

import numpy as np

def create_diagonally_dominant_matrix(n):
  """Creates a diagonally dominant random matrix.

  Args:
    n: The size of the matrix.

  Returns:
    A diagonally dominant random matrix.
  """

  A = np.random.rand(n, n)
  for i in range(n):
    A[i, i] = np.sum(np.abs(A[i, :])) + np.random.rand()  # Ensure diagonal dominance

  return A
```

```
# put this into a function for convenience
def iter_solve(A, b, method):
  x0 = np.zeros(b.shape[0])

  max_iter = 1000
  tolerance = 1e-6

  for i in range(max_iter):
    x_new = method(A, b, x0.copy())
    R = np.linalg.norm(A@x_new-b)

    #print('Iteration ', i, ' determined ', x_new, 'with residual', R)
    if np.linalg.norm(R) < tolerance:
      print('Converged after, ', i, ' iterations.')
      break
    x0 = x_new.copy()
```

**n=4**

```
n = 4  # Size of the matrix
A = create_diagonally_dominant_matrix(n)
b = np.random.rand(n)  # Create a random right-hand side vector

print('LU')
%timeit -n1 -r1 np.linalg.solve(A, b)

print('\nJacobi')
%timeit -n1 -r1 iter_solve(A, b, jacobi_step)
print('\nGauss-Seidel')
%timeit -n1 -r1 iter_solve(A, b, gauss_seidel_step)
print('\nSOR')
%timeit -n1 -r1 iter_solve(A, b, sor_step)
```

```
LU
90.6 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

Jacobi
Converged after,  18  iterations.
6.07 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

Gauss-Seidel
Converged after,  6  iterations.
1.61 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

SOR
Converged after,  6  iterations.
3.14 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

**n = 100**

```
n = 100  # Size of the matrix
A = create_diagonally_dominant_matrix(n)
b = np.random.rand(n)  # Create a random right-hand side vector

print('LU')
%timeit -n1 -r1 np.linalg.solve(A, b)

print('\nJacobi')
%timeit -n1 -r1 iter_solve(A, b, jacobi_step)
print('\nGauss-Seidel')
%timeit -n1 -r1 iter_solve(A, b, gauss_seidel_step)
print('\nSOR')
%timeit -n1 -r1 iter_solve(A, b, sor_step)
```

```
LU
5.24 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

Jacobi
Converged after,  787  iterations.
1.39 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

Gauss-Seidel
Converged after,  10  iterations.
19.4 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

SOR
Converged after,  11  iterations.
12.9 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

**n = 1000**

```
n = 1000  # Size of the matrix
A = create_diagonally_dominant_matrix(n)
b = np.random.rand(n)  # Create a random right-hand side vector

print('LU')
%timeit -n1 -r1 np.linalg.solve(A, b)

#print('\nJacobi')
#%timeit -n1 -r1 iter_solve(A, b, jacobi_step)
print('\nGauss-Seidel')
%timeit -n1 -r1 iter_solve(A, b, gauss_seidel_step)
#print('\nSOR')
#%timeit -n1 -r1 iter_solve(A, b, sor_step)
```

```
LU
50.3 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

Gauss-Seidel
Converged after,  10  iterations.
115 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

**n = 3000**

```
n = 3000  # Size of the matrix
A = create_diagonally_dominant_matrix(n)
b = np.random.rand(n)  # Create a random right-hand side vector

print('LU')
%timeit -n1 -r1 np.linalg.solve(A, b)

#print('\nJacobi')
#%timeit -n1 -r1 iter_solve(A, b, jacobi_step)
print('\nGauss-Seidel')
%timeit -n1 -r1 iter_solve(A, b, gauss_seidel_step)
#print('\nSOR')
#%timeit -n1 -r1 iter_solve(A, b, sor_step)
```

```
LU
692 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

Gauss-Seidel
Converged after,  10  iterations.
411 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

**n = 5000**

HERE WE GO! Remember, this is a terrible implementation of GS! Numpy with loops is going to be slow!

```
n = 5000  # Size of the matrix
A = create_diagonally_dominant_matrix(n)
b = np.random.rand(n)  # Create a random right-hand side vector

print('LU')
%timeit -n1 -r1 np.linalg.solve(A, b)

#print('\nJacobi')
#%timeit -n1 -r1 iter_solve(A, b, jacobi_step)
print('\nGauss-Seidel')
%timeit -n1 -r1 iter_solve(A, b, gauss_seidel_step)
#print('\nSOR')
#%timeit -n1 -r1 iter_solve(A, b, sor_step)
```

```
LU
2.8 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

Gauss-Seidel
Converged after,  11  iterations.
874 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

## Nomenclature

Capital letters are Matricies: $A$

Vectors have arrows above: $\vec{x}$

Scalars are lower case: $\alpha$

Superscripts generally denote iteration number: $\vec{x}^k$.

## Conjugate gradient (CG)

Conjugate gradient method is a standard tool for solving symmetric positive definite systems.

Consider a quadratic surface, defined by:

$f(\vec{x}^k) = \frac{1}{2} \vec{x}^T A \vec{x} -\vec{b}^T \vec{x}$.

This surface has a minimum at:

$$\begin{align} \frac{d f}{ d x} = \vec{0} &= A \vec{x}-\vec{b} \ \vec{0} &= \vec{r} \end{align}$$

Where $\vec{r}=A \vec{x}-\vec{b} \ )$ is the residual. Minimizing $f$ is equivilant to solving $A\vec{x} = \vec{b}$ which is equiviliant to finding $\vec{r}=\vec{0}$.

For a general guess $\vec{x}^k$,

$$\begin{align} A \vec{x}^k-\vec{b} =\vec{r}^k \ne 0 \ \end{align}$$

### The Hessian

Note: The matrix $A$ in this case is the *hessian*, $H$, of $f$, which is the matrix made by the mixed second derivatives:

$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f} {\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$

From the rule of mixed second derivatives, we see that the Hessian is symmetric.

The Hessian is positive definite if $\vec{x}^T H \vec{x} > 0$ for all $\vec{x}$. In this case the quadratic surface $f$ is convex (opening upwards). For a 1D case, this means you have a positive second derivative.

> For comparison, the other common function with second derivatives is the Laplacian, which is a scalar sum of the diagonal elements of H:
> $\Delta f = \nabla^2 f = \frac{\partial^2 f}{\partial x_1^2} + \frac{\partial^2 f}{\partial x_2^2} + \frac{\partial^2 f}{\partial x_3^2} ... \)$

- Since $A$ is symmetric, you could form a surface $f$ which has an *extrenum* at the solution of the system. These problems are generally called *saddle point problem*.
- Since $A$ is symmetric positive definite, the surface is convex quadratic (paraboloid) which has a minimum at the solution of the linear system.

##The algorithm

Starting at $\vec{x}^k$, we will choose step direction $\vec{s}^k$, and step length $\alpha^k$ to reach our next guess $\vec{x}^{k+1}$:

$\vec{x}^{k+1} = \vec{x}^k + \alpha^k \vec{s}^k$

## Choose the step length

An obvious choice for $\alpha^k$ is whatever minimizes $f(\vec{x}^{k+1})$!

$f(\vec{x}+\alpha^k \vec{s}^k) = \frac{1}{2} [\vec{x}+\alpha^k \vec{s}^k]^T A [\vec{x}+\alpha^k \vec{s}^k] -\vec{b}^T [\vec{x}+\alpha^k \vec{s}^k]$
.

The minimum is found when:

$$\begin{aligned} \frac{\partial f}{\partial \alpha^k} = 0 &= {\vec{s}^k}^T A [\vec{x}+\alpha^k \vec{s}^k] -\vec{b}^T \vec{s}^k\ &= {\vec{s}^k}^T A \vec{x}^k+ {\vec{s}^k}^T A \alpha^k \vec{s}^k -{\vec{s}^k}^T \vec{b} \ &= {\vec{s}^k}^T [A \vec{x}^k-\vec{b}] + \alpha^k {\vec{s}^k}^T A \vec{s}^k \ \alpha_k &= \frac{{\vec{s}^k}^T \vec{r}^k}{{\vec{s}^k}^T A \vec{s}^k} \end{aligned}$$

So given a step direction $\vec{s}^k$ we can take an optimal step length $\alpha_k$ which is determined by the current residual $\vec{r}^k$.

## Choose the step direction: Steepest descent

The magic of these methods comes from the choice of step direction.

How do we go about choosing $\vec{s}^k$?

The intuitive answer is the negative gradient of $f(x^k)$, which is simply,

$s^k = -\nabla f = -\frac{d f}{d\vec{x}} = -\vec{r}^k$

The optimal step size is therefore,

$\alpha_k=\frac{{\vec{r}^k}^T \vec{r}^k}{{\vec{r}^k}^T A \vec{r}^k}$

Now we take the step to form $\vec{x}^{k+1}$ and repeat the process until your choice of tolerance on the residual (or $\vec{x}^k$)!) is met.

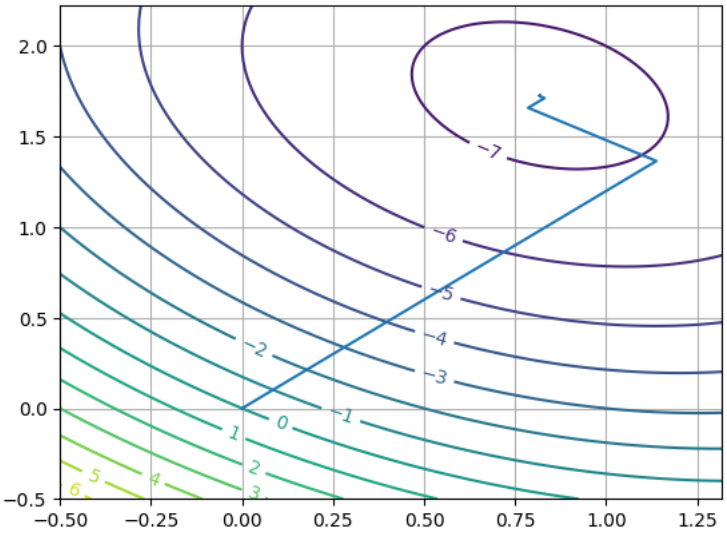
Let’s code it!

```python
# Copy the surface plotter from the last lecture:
import numpy as np
import matplotlib.pyplot as plt

def plot_surface(A, b, x0, x_true):
  # Utility function to autoscale the plot limits to include x0 and x_true
  def calculate_box_limits(point1, point2, padding=.5):
    # Extract coordinates
    x1, y1 = point1
    x2, y2 = point2
    # Calculate min and max coordinates with padding
    min_x = min(x1, x2) - padding
    max_x = max(x1, x2) + padding
    min_y = min(y1, y2) - padding
    max_y = max(y1, y2) + padding
    return (min_x, max_x, min_y, max_y)

  xb, xe, yb, ye = calculate_box_limits(x0, x_true)
  # Plot the linear system
  x = np.linspace(xb, xe, 100)
  y = np.linspace(yb, ye, 100)
  X, Y = np.meshgrid(x, y)
  V = np.stack((X,Y),-1)
  Z = 0.5 * (A[0, 0] * X**2 + A[1, 1] * Y**2 + 2 * A[0, 1] * X * Y) - (b[0] * X + b[1] * Y)
  fig, ax = plt.subplots()

  CS = ax.contour(X, Y, Z, levels = 15)
  ax.set_xlim([xb, xe])  # Set x-axis limits
  ax.set_ylim([yb, ye])  # Set y-axis limits
  ax.clabel(CS, inline=True, fontsize=10)

  return fig
```

```python
# prompt: write a function to solve a linear system using the method of steepest descent

import numpy as np
def steepest_descent(A, b, x0, tol=1e-6, max_iter=100):
  """
  Solves a linear system Ax = b using the method of steepest descent.


  Args:
    A: The coefficient matrix (must be symmetric positive definite).
    b: The right-hand side vector.
    x0: The initial guess for the solution.
    tol: The tolerance for the residual.
    max_iter: The maximum number of iterations.

  Returns:
    x: The approximate solution.
    residuals: A list of the residuals at each iteration.
  """
  x = x0
  iterations = [x]
  for i in range(max_iter):
    r = A @ x - b
    if np.linalg.norm(r) < tol:
      break
    alpha = np.dot(r, r) / np.dot(r, A @ r)
    x = x - alpha * r
    print('Iteration ', i, ' : ',x)
    iterations.append(np.copy(x))
  return x, iterations
```

```
A = np.array([[4, 1], [1, 3]])
b = np.array([5, 6])
x0 = np.array([0., 0.])

x_true = np.linalg.solve(A, b)
print('The true answer is ', x_true)

fig = plot_surface(A, b, x0, x_true)

_, iterations = steepest_descent(A, b, x0)

iterations = np.array(iterations)

plt.plot(iterations[:, 0], iterations[:, 1], marker='', linestyle='-')
plt.grid(True)
plt.show()
```

```
The true answer is  [0.81818182 1.72727273]
Iteration  0  :  [1.1380597  1.36567164]
Iteration  1  :  [0.78590538 1.65913358]
Iteration  2  :  [0.83080067 1.71300793]
Iteration  3  :  [0.81690855 1.72458471]
Iteration  4  :  [0.81867962 1.72671   ]
Iteration  5  :  [0.81813159 1.72716669]
Iteration  6  :  [0.81820146 1.72725053]
Iteration  7  :  [0.81817984 1.72726854]
Iteration  8  :  [0.81818259 1.72727185]
Iteration  9  :  [0.81818174 1.72727256]
```




The approach results in the final \(\vec{x}^{approx}\) (eg: after 9 iterations) being formed as:

\(\vec{x}^{9} = \vec{x}^0 + \alpha^1 \vec{s}^1 + \alpha^2 \vec{s}^2 + \alpha^3 \vec{s}^3 ...\)

Let's consider this for a minute. We are assembling the vector that moves from \(\vec{x}^0\) to \(~\vec{x}^9\), as a series of 9 steps in directions \(\vec{s}^k\) with magnitude \(\alpha^k\).

-> What does this expression remind you of?

This path is inefficient because you end up with zig-zagging where one step *undoes* some of the others which is inefficient.

-> How do we ensure the next step doesn't undo the previous step?

## Choose the step direction: Conjugate gradient

The problem is that certain steps undo the effect of the previous step(s). We can instead choose successive step directions to *not undo each other*, by requiring the next step be *conjugate* to each other.

This means we must base the $\vec{s}^{k+1}$ step on the $\vec{s}^{k}$ step and the gradient descent (which we just saw was $\vec{r}^k$)).

$\vec{s}^{k+1} = \vec{r}^{k+1}+\beta^k \vec{s}^k$

and require that $\vec{s}^{k+1}$ and $\vec{s}^k$ are *conjugate to each other in $A$)*:

${\vec{s}^{k+1}}^T A \vec{s}^k = 0$

This does not mean that $\vec{s}^{k+1}$ and $\vec{s}^k$ are orthogonal; rather vectors $\vec{s}^{k+1}$ and $A \vec{s}^k$ (or equivilantly $A \vec{s}^{k+1}$ and $\vec{s}^k$) are orthogonal.

We can then solve,

$$\begin{aligned} [\vec{r}^{k+1}+\beta^k \vec{s}^k]^T A \vec{s}^k &= 0\\ \beta^k &= -\frac{\vec{r}^{k+1} A \vec{s}^k}{{\vec{s}^k}^T A \vec{s}^k} \end{aligned}$$

and so we find the next step direction,

$s^{k+1} = r^{k+1} -\frac{r^{k+1} A s^k}{{s^k}^T A s^k} s^k$.

The step length $\alpha_k$ is determined exactly as before.

Actually, we are going to make the new step direction conjugate to *all* the previous directions, so we have to say:

$\vec{s}^{k+1} = \vec{r}^{k+1} - \sum_{i<k}\frac{\vec{r}^{k+1} A \vec{s}^k}{{\vec{s}^k}^T A \vec{s}^k} \vec{s}^k$.

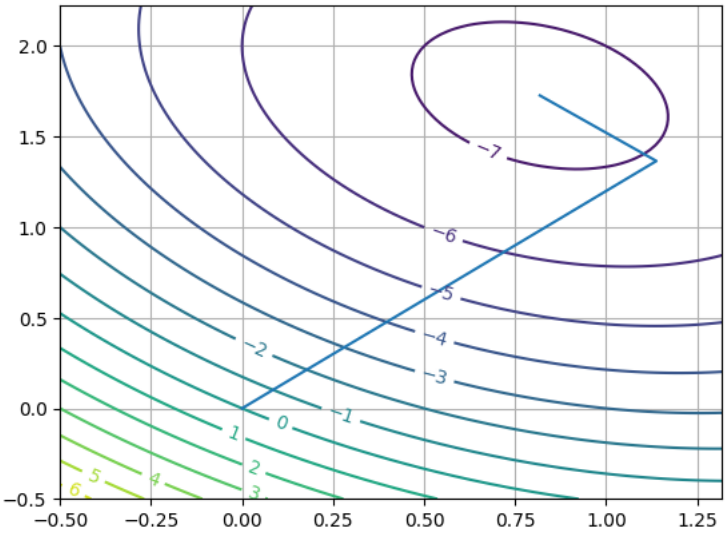
```
A = np.array([[4, 1], [1, 3]])
b = np.array([5, 6])
x0 = np.array([0., 0.])

x_true = np.linalg.solve(A, b)
print('The true answer is ', x_true)
```

```
The true answer is  [0.81818182 1.72727273]
```

```
# prompt: call a cg solve on this system, but build a list of the iterations
from scipy.sparse.linalg import cg
import numpy as np

iterations = [x0]
print(x0)
def print_iteration_and_save(xk):
  iterations.append(np.copy(xk))
  print(xk)

solution, info = cg(A, b, x0=x0, callback=print_iteration_and_save)
```

```
[0. 0.]
[1.1380597  1.36567164]
[0.81818182 1.72727273]
```

```
iterations = np.array(iterations)

fig = plot_surface(A, b, x0, x_true)

# Plot the iterations on a 2D surface
plt.plot(iterations[:, 0], iterations[:, 1], marker='', linestyle='-')
plt.grid(True)
plt.show()
```




Let's examine the steps, which we can find as the difference between successive iterations:

```
s1 = iterations[1,:]-iterations[0,:]
s2 = iterations[2,:]-iterations[1,:]
print(s1,s2)
print(np.dot(s1,s2))  #not zero
print(np.dot(s1,A@s2))  #Zero
print(np.dot(s2,A@s1))  #Zero
```

```
[1.1380597  1.36567164] [-0.31987788  0.36160109]
0.1297882196885316
6.661338147750939e-16
6.661338147750939e-16
```

Successive steps are conjugate in \(A\) as expected.

Another interesting property is that the residual at each step is orthogonal:

```
r0 = A@iterations[0,:]-b
r1 = A@iterations[1,:]-b
r2 = A@iterations[2,:]-b

print(r0.dot(r1))
print(r1.dot(r2))
```

```
0.0
0.0
```

which helps with computational efficiency.

## Analysis

Beginning at \(\vec{x}^0\), we are tasked with finding \(\vec{x}^{approx}\) through an expansion:

\(\vec{x}^{approx} - \vec{x}^0 = \alpha^1 \vec{s}^1 + \alpha^2 \vec{s}^2 + ...\).

but this is simply vector addition using a set of directions. Our algorithms has generated the directions \(\vec{s}^k\) which form a set of *vector bases* for:

\(\vec{x}^{approx} - \vec{x}^0\)

and, at the same time, the corresonding *components*, the \(\alpha^k\).

- The method of steepest descent doesn't enforce all vector bases be orthogonal (or conjugate), resulting in frenetic zig-zagging and more bases than necessary.
- The conjugate gradient method enforces conjugacy in the vector bases such that they don't overlap. Vectors are linearly independent from the matrix-vector products of the others. This algorithm relies on \(A\) being SPD.

> Note that these techniques only ever use \(A\) in the context for a matrix-vector product - \(A \vec{x}\). Matrix-vector products are \(O(n^2)\) (for dense matricies, better for sparse) but there is also the storage requirement for \(A\). If we can find a better way to get the vector \(A\vec{x}\) that doesn't involve (assembling), storing, and multiplying \(A\), we can see improvements. This is the basis of **Matrix-free** algorithms.

Note the *space* of the solution vector \(x\) is just its dimension \(n\), and if CG determins linearly independent vectors, it can only find, at most \(n\)! Therefore, CG will find the exact solution in \(n\) iterations (baring roundoff error) which some may take to imply it is a *direct method*. However, the situation is even better, which qualifies it as an iterative technique! Let's see what happens with a larger system:

```
# prompt: Solve a 10x10 random spd linear system with CG

import numpy as np
from scipy.sparse.linalg import cg

n = 8
A_r = np.random.rand(n, n)
A_rand = A_r @ A_r.T  # Ensure A is symmetric positive definite
b_rand = np.random.rand(n)
x0_rand = np.zeros(n)


class it_counter(object):
    def __init__(self, disp=True):
        self._disp = disp
        self.niter = 0
    def __call__(self, rk=None):
        self.niter += 1
        if self._disp:
            print('iter %3i\trk = %s' % (self.niter, str(rk)))

np.set_printoptions(precision=4)

solution, info = cg(A_rand, b_rand, x0=x0_rand, atol = 1e-6, rtol = 1e-10, callback = it_counter())
print("Solution:", solution)
print("Info:", info)
```

```
iter   1        rk = [0.0209 0.0075 0.032  0.0783 0.0439 0.0073 0.0369 0.0801]
iter   2        rk = [-0.8925 -1.2871 -0.2747  1.7474  1.0466 -0.8757  0.0896  1.3888]
iter   3        rk = [-1.1486 -1.8634 -0.5373  1.7162  1.341  -0.7321  0.404   1.9777]
iter   4        rk = [-1.2676 -2.5568 -0.1426  1.5324  2.2171 -0.4695  0.4377  1.9871]
iter   5        rk = [-2.2603 -2.864   0.6716  1.1551  3.1509 -0.1319  0.5006  2.1877]
iter   6        rk = [-2.9958 -2.8345  0.9681  0.9131  4.1145 -0.3175  1.4379  1.8352]
iter   7        rk = [-2.9739 -2.8203  1.0523  0.7471  4.2074 -0.5366  1.4883  1.9233]
iter   8        rk = [-2.9605 -2.7277  0.9849  0.6628  4.3529 -0.6061  1.4368  1.9791]
iter   9        rk = [-2.9605 -2.7277  0.9849  0.6628  4.3529 -0.6061  1.4368  1.9791]
Solution: [-2.9605 -2.7277  0.9849  0.6628  4.3529 -0.6061  1.4368  1.9791]
Info: 0
```

Note that successive iterations are getting closer and closer. Usually, the estimates converge before \(n\) iterations (which is especially useful for large sparse matricies!)

###Restart

Sometimes, the set of \(s^k\) gets too large and round-off error can accumulate. Therefore, some implementation have an integer 'restart' parameter which will trigger a restart of the bases after a set number have been generated.

## Generalized Minimal Residual Method (GMRES)

Let's finally turn our attention towards the general case of $A$ being a general non-symmetric matrix. Since it is non-symmetric, it can not be a Hessian and there is no quadratic surface associated with it.

We can adapt the same basic method of CG:

- Take the residual of the system, $\vec{r} = A\vec{x}-\vec{b}$ and minimize its Euclidian norm, $||\vec{r}||$.
- Beginning with $\vec{x}^0$, construct a basis for $\vec{x}^{approx}-\vec{x}^0$ in $\vec{s}^k$.
- Enforce orthogonality with successive $\vec{s}^{k}$
- Since each $\vec{s}^k$ is linearly independent, the method will converge in at most $n$ iterations, but typically converges much sooner.
- A restart may still be necessary to control memory consumption for large systems.

The major difference is that the conjugacy of $\vec{s}^k$ is not possible since $A$ is not SPD. Rather, $\vec{s}^k$ are chosen to be orthogonal which is a more general, but less powerful condition.

```
from scipy.sparse.linalg import gmres
A = np.array([[4, 1], [1, 3]])
b = np.array([5, 6])
x0 = np.array([0., 0.])


iterations = [x0]
print(x0)
def print_iteration_and_save(xk):
  iterations.append(np.copy(xk))
  print(xk)

solution, info = gmres(A, b, callback=it_counter(), callback_type = 'legacy')  #x
np.linalg.solve(A,b)
print(iterations)
```

```
[0. 0.]
iter   1        rk = 0.15122563463888616
iter   2        rk = 7.674967607276873e-17
[array([0., 0.])]
```

```
np.set_printoptions(precision=4)

solution, info = gmres(A_rand, b_rand, x0=x0_rand, atol = 1e-6, rtol = 1e-10, callback = it_counter(), ca
print("Solution:", solution)
print("Info:", info)
```

```
iter   1        rk = 0.6009604017495943
iter   2        rk = 0.3918500855823959
iter   3        rk = 0.20243617177538392
iter   4        rk = 0.15688283627620023
iter   5        rk = 0.10599045198232475
iter   6        rk = 0.0398441273250505
iter   7        rk = 0.022361032564208502
iter   8        rk = 2.2495773485670963e-15
Solution: [-2.9605 -2.7277  0.9849  0.6628  4.3529 -0.6061  1.4368  1.9791]
Info: 0
```

## Preconditioners

Examples for this lecture are drawn from here

The basic strategy of preconditioning is to modify the original linear system to one that *behaves better*:

- Direct solvers typically don't need *user specified* preconditioning since they are already robust and/or the preconditioner can be integrated. E.g.: Row swapping via the permutation matrix before conducting Gaussian Elimination / LU decomposition to achieve diagonal dominance which is cheap and easy to implement.
- Iterative solvers benefit substantially from preconditioners to achieve efficient convergence and control roundoff errors. Many (if not all) sparse matrix iterative solvers expect preconditioners and are applied *efficiently* during execution. Unfortuantely, **there is no general preconditioning technique that always works well** :-(

Certain preconditioners are effective on a particular type of physics, but even then the nature of the boundaries can torpedo convergence.

Preconditioners are an active field of research and key to solving large linear systems ($n=10^{6+}$). They are also critical to using distributed computing (HPC).

There are a variety of preconditioners, the mathematics of which we won't explore here. Rather, lets explore what to look for in a preconditioner, and how to use them.

Consider the linear system, $Ax=b$ we want to multiply the equation by a matrix $P^{-1}$ to make the system easier to solve. As long as $P^{-1}$ isn't singular, the answer won't be changed.

## Left preconditioning

The most traditional and common preconditioner is:

$P^{-1} A x = P^{-1} b$

## Right preconditioning

A varient that is better for some cases

$A P^{-1} y = b$

with,

$P x = y$

## Two sided preconditioning

The combination of left and right:

$Q A P^{-1} y = Q b$

again with,

$P x = y$

this method may be needed to preserve symmetry of $A$ (which we have seen is very useful!). The problem is that in general the product of two symmetric matricies is not symmetric! If we say that $Q^T=P^{-1}$, then $QAP^{-1}$ is symmetric.

Recall that the inverse of a sparse matrix is typically dense, and therefore we don't usually want to store $P^{-1}$ explicitly. It is better to store $P$ and the method by which we would apply $P^{-1}$ to a vector. This is similar to solving a system by LU decomposition, $A=LU$. We don't calculate $A^{-1}$ explicitly, we keep $L$ and $U$ and solve systems sequentially.

## Qualities

The best preconditioner is $P = A$, which makes the system trivial. Failing this, we aim for $P^{-1} \approx A^{-1}$ *in some sense*. This can be quantified by the *condition number* of the preconditioned matrix, which we aim to make $=1$.

Preconditioner quality is based on:

- speed of computation and application of $P^{-1}$
- memory requirements of storing $P^{-1}$ (if it is stored)
- degree of *fill-in* (sparsity compared to $A$)
- parallelizability

Since calculation, storage, and application of $P^{-1}$ is an additional computational expense, the benefit must be *worth it*. Often this translates to the total speed of solution (preconditioning and iteration) but it may also extend to the stability of the computation.

E.g.: In time-dependent nonlinear PDEs, we will be solving linear systems thousands of times. A solver that takes twice as long every time but always converges may be better than one that solves fast but sometimes doesn't converge (which may spoil any progress you've made!).

## Examples

Let's discuss some common preconditioners and see their effect **on a particular system**.

### Jacobi

The Jacobi preconditioner (otherwise called diagonal scaling) is simply the main diagonal of $A$. The inverse is trivially the inverse of the diagonal elements, which means we can store $P^{-1}$ efficiently.

Example: Solve the discrete Poisson equation on the unit square,

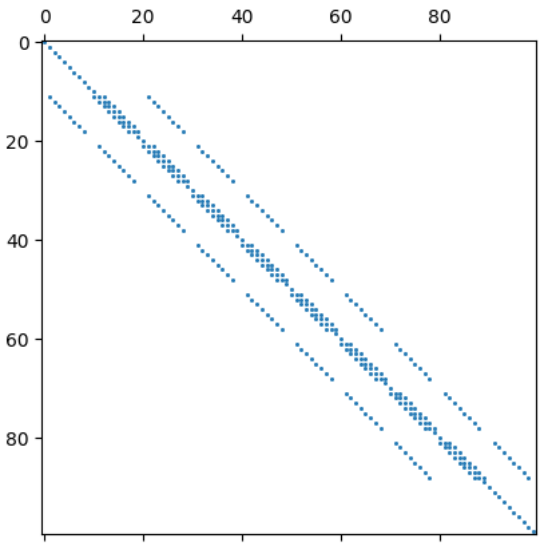
$-\Delta u = 1$ on $\Omega=[0,1]^2$

with

$u=0$ on $\partial \Omega$

```python
import numpy as np
from scipy.sparse import coo_matrix

def discretise_poisson(N, Q):
    """Generate the matrix and rhs associated with the discrete Poisson operator."""

    nelements = 5 * N**2 - 16 * N + 16

    row_ind = np.empty(nelements, dtype=np.float64)
    col_ind = np.empty(nelements, dtype=np.float64)
    data = np.empty(nelements, dtype=np.float64)

    f = np.empty(N * N, dtype=np.float64)

    count = 0
    for j in range(N):
        for i in range(N):
            if i == 0 or i == N - 1 or j == 0 or j == N - 1:
                row_ind[count] = col_ind[count] = j * N + i
                data[count] =  1
                f[j * N + i] = 0
                count += 1

            else:
                row_ind[count : count + 5] = j * N + i
                col_ind[count] = j * N + i
                col_ind[count + 1] = j * N + i + 1
                col_ind[count + 2] = j * N + i - 1
                col_ind[count + 3] = (j + 1) * N + i
                col_ind[count + 4] = (j - 1) * N + i

                data[count] = 4 * (N - 1)**2
                data[count + 1 : count + 5] = - (N - 1)**2
                f[j * N + i] = Q

                count += 5

    return coo_matrix((data, (row_ind, col_ind)), shape=(N**2, N**2)).tocsr(), f
```

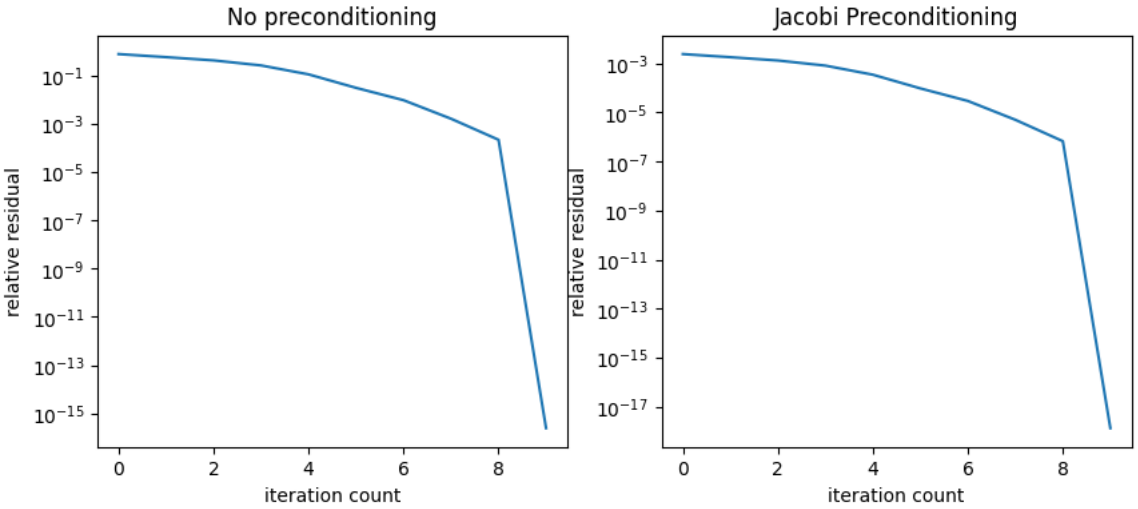
```python
N = 10  # Mesh size
A, f = discretise_poisson(N, 1)

import matplotlib.pyplot as plt

plt.spy(A, markersize=1)
plt.show()
```




Implement a Jacobi preconditioner

```python
from scipy.sparse import diags#, LinearOperator
from scipy.sparse.linalg import gmres
from numpy.linalg import cond



Pinv_jacobi = diags(1/A.diagonal(), offsets = 0,format='csr')

print('Unconditioned condition number :', np.linalg.cond(A.todense()))
print('Preconditioned condition number :', np.linalg.cond(Pinv_jacobi@A.todense()))


residuals1 = []
callback1 = lambda res: residuals1.append(res)
residuals2 = []
callback2 = lambda res: residuals2.append(res)

#Solve the system using GMRES with the Jacobi preconditioner
x_uncond, _ = gmres(A, f, callback=callback1, callback_type='pr_norm')


x_cond, _ = gmres(A, f, M = Pinv_jacobi, callback=callback2, callback_type='pr_norm')

print(x_cond-x_uncond)


fig = plt.figure(figsize=(10, 4))
ax1 = fig.add_subplot(121)
ax1.semilogy(residuals1)
ax1.set_title('No preconditioning')
ax1.set_xlabel('iteration count')
ax1.set_ylabel('relative residual')

ax2 = fig.add_subplot(122)
ax2.semilogy(residuals2)
ax2.set_title('Jacobi Preconditioning')
ax2.set_xlabel('iteration count')
ax2.set_ylabel('relative residual');
```

```
Unconditioned condition number : 1115.4433890393989
Preconditioned condition number : 32.30864938962053
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00 -3.46944695e-18
 -1.38777878e-17 -1.38777878e-17 -2.77555756e-17 -2.77555756e-17
 -1.04083409e-17 -1.38777878e-17 -1.73472348e-18  0.00000000e+00
  0.00000000e+00 -1.73472348e-17 -3.46944695e-17 -3.46944695e-17
 -4.16333634e-17 -4.16333634e-17 -2.77555756e-17 -3.46944695e-17
 -1.04083409e-17  0.00000000e+00  0.00000000e+00 -1.38777878e-17
 -3.46944695e-17 -5.55111512e-17 -5.55111512e-17 -5.55111512e-17
 -4.85722573e-17 -3.46944695e-17 -1.04083409e-17  0.00000000e+00
  0.00000000e+00 -2.77555756e-17 -4.16333634e-17 -5.55111512e-17
 -5.55111512e-17 -5.55111512e-17 -5.55111512e-17 -4.16333634e-17
 -2.77555756e-17  0.00000000e+00  0.00000000e+00 -2.77555756e-17
 -4.16333634e-17 -5.55111512e-17 -5.55111512e-17 -4.16333634e-17
 -5.55111512e-17 -4.16333634e-17 -2.77555756e-17  0.00000000e+00
  0.00000000e+00 -1.04083409e-17 -3.46944695e-17 -4.85722573e-17
 -5.55111512e-17 -5.55111512e-17 -4.85722573e-17 -2.77555756e-17
 -1.04083409e-17  0.00000000e+00  0.00000000e+00 -1.04083409e-17
 -2.77555756e-17 -3.46944695e-17 -3.46944695e-17 -3.46944695e-17
 -3.46944695e-17 -2.77555756e-17 -1.04083409e-17  0.00000000e+00
  0.00000000e+00 -1.73472348e-18 -1.04083409e-17 -1.04083409e-17
 -2.08166817e-17 -2.08166817e-17 -1.04083409e-17 -1.04083409e-17
 -1.73472348e-18  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
```




meh.

##Incomplete LU (ILU)

For a sparse matrix \(A\), the \(L\) and \(U\) factores are more filled in than than \(A\) which increases memory usage.

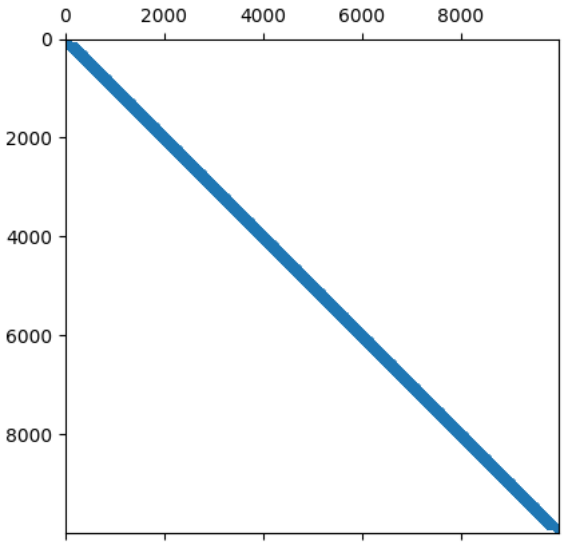
Instead of \(A=LU\), we can look for factors \(\tilde{L}\), \(\tilde{U}\) such that \(P=\tilde{L}\tilde{U} \approx A\).

The different approximations characterized by the degree of fill-in compared to the original matrix \(A\). ILU(0) preserves the sparsity pattern. A common approach is to match the sparsity of \(A^k\) (which becomes more and more dense), forming the \(ILU(k)\) approximations.

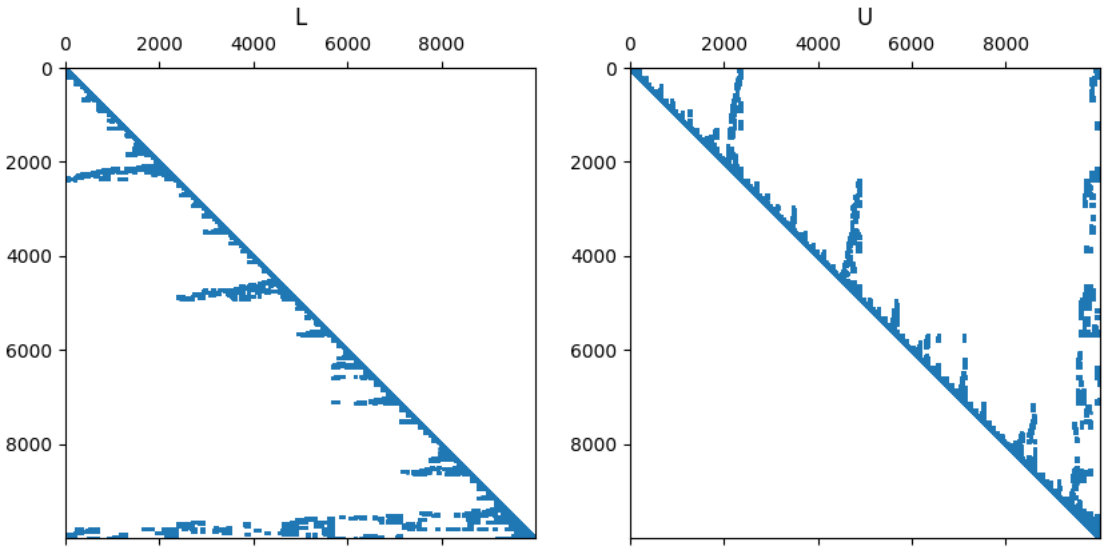
Obviously solving \(\tilde{L} \tilde{U} x = b\) won't give us the correct answer, so instead we use \(P\) as a preconditioner for an iterative algorithm.

Let's see the LU decomposotion

```
# prompt: Find the LU factors of A, and make spy plots of both side by side

import matplotlib.pyplot as plt
from scipy.sparse.linalg import splu,  spilu, gmres

N = 100 #Mesh size
A, f = discretise_poisson(N, 1)

import matplotlib.pyplot as plt

plt.spy(A, markersize=1)
plt.show()


lu = splu(A)
L = lu.L
U = lu.U

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
ax1.spy(L, markersize=1)
ax1.set_title('L')
ax2.spy(U, markersize=1)
ax2.set_title('U')
plt.show()
```




```
<ipython-input-21-2ebbdb197289>:15: SparseEfficiencyWarning: splu converted its input to CSC format
  lu = splu(A)
```

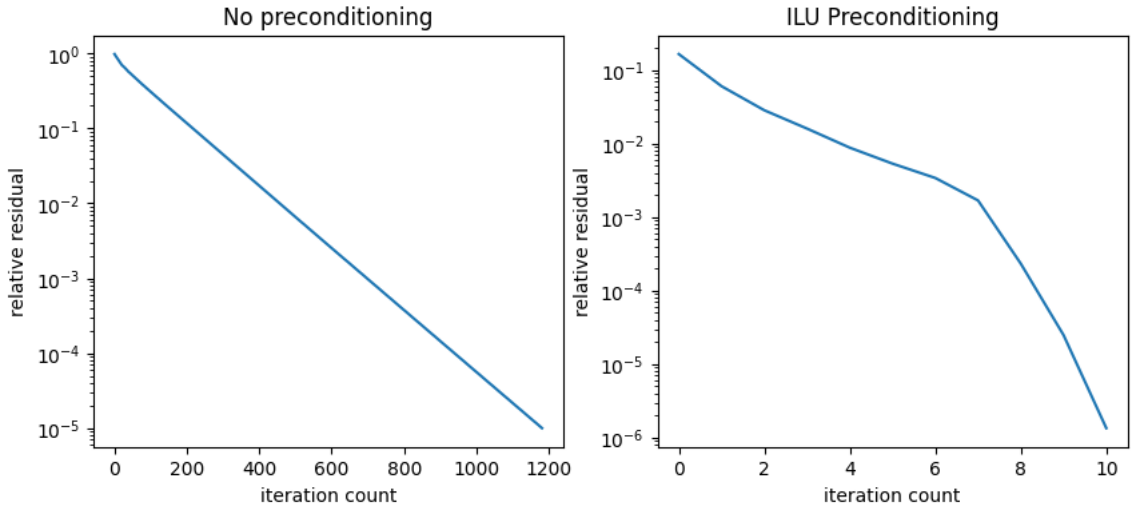



Perform sparse ilu and apply to the system.

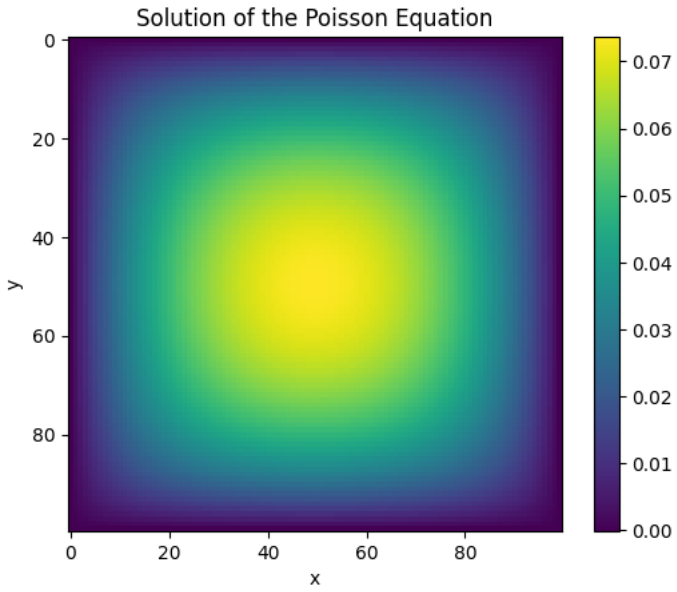
**NOTE** The precoditioner is passed as an operator. Rather than passing \(P^{-1}\) we are passing a function that solve \(P x = y\). This way we avoid storing a dense matrix!

```
import matplotlib.pyplot as plt
from scipy.sparse.linalg import spilu, LinearOperator
ilu = spilu(A, fill_factor=20, drop_rule='dynamic')

M = LinearOperator(matvec=ilu.solve, shape=(N**2, N**2), dtype=np.float64)

residuals1 = []
callback1 = lambda res: residuals1.append(res)
x, _ = gmres(A, f, callback=callback1, callback_type='pr_norm')


residuals2 = []
callback2 = lambda res: residuals2.append(res)
x, _ = gmres(A, f, M=M, callback=callback2, callback_type='pr_norm')

fig = plt.figure(figsize=(10, 4))
ax1 = fig.add_subplot(121)
ax1.semilogy(residuals1)
ax1.set_title('No preconditioning')
ax1.set_xlabel('iteration count')
ax1.set_ylabel('relative residual')

ax2 = fig.add_subplot(122)
ax2.semilogy(residuals2)
ax2.set_title('ILU Preconditioning')
ax2.set_xlabel('iteration count')
ax2.set_ylabel('relative residual');
```

```
<ipython-input-13-c4528330daae>:3: SparseEfficiencyWarning: spilu converted its input to CSC format
  ilu = spilu(A, fill_factor=20, drop_rule='dynamic')
```




```
import matplotlib.pyplot as plt

# Assuming x_cond contains the solution obtained with preconditioning
solution = x.reshape(N, N)

plt.imshow(solution, cmap='viridis')
plt.colorbar()
plt.title('Solution of the Poisson Equation')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```




##Sparse Approximate Inverse

Sparse approximate inverse is another tool that attempts to find \(P^{-1}\) so as to minimize the Frobenius norm of

\(||I-AP^{-1}||\)

It generates successive iterations of the preconditioner with increasing density.

```
def spai(A, m):
    """Perform m step of the SPAI iteration."""
    from scipy.sparse import identity
    from scipy.sparse import diags
    from scipy.sparse.linalg import onenormest

    n = A.shape[0]

    ident = identity(n, format='csr')
    alpha = 2 / onenormest(A @ A.T)
    M = alpha * A

    for index in range(m):
        C = A @ M
        G = ident - C
        AG = A @ G
        trace = (G.T @ AG).diagonal().sum()
        alpha = trace / np.linalg.norm(AG.data)**2
        M = M + alpha * G

    return M
```

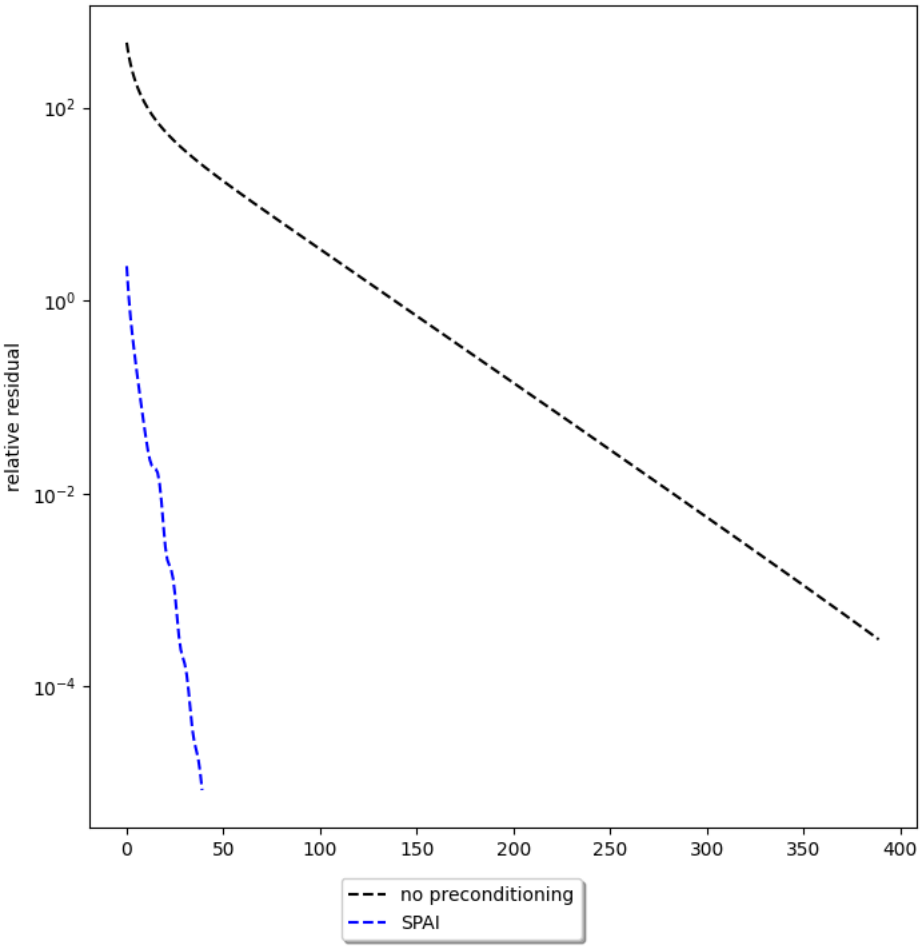
Note the demonstrative example is for a different matrix.

```
import numpy as np
from scipy.sparse import diags

n = 1000

data = [2.001 * np.ones(n),
        -1. * np.ones(n - 1),
        -1. * np.ones(n - 1)]

offsets = [0, 1, -1]

A = diags(data, offsets=offsets, shape=(n, n), format='csr')

M = spai(A, 50)

print('Unconditioned condition number :', np.linalg.cond(A.todense()))
print('Preconditioned condition number :', np.linalg.cond(A.todense() @ M.todense()))
```

```
Unconditioned condition number : 3961.9652414685806
Preconditioned condition number : 40.18659718436124
```

```
from scipy.sparse.linalg import cg
n = A.shape[0]
b = np.ones(n)


residuals = []
callback = lambda x: residuals.append(np.linalg.norm(A @ x - b))
x, _ = cg(A, b, callback=callback)

residuals_preconditioned = []
callback = lambda x: residuals_preconditioned.append(np.linalg.norm(A @ x - b) / np.linalg.norm(b))
x, _ = cg(A, b, M=M, callback=callback)

fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111)
ax.semilogy(residuals, 'k--')
ax.semilogy(residuals_preconditioned, 'b--')
ax.set_ylabel('relative residual')
fig.legend(['no preconditioning', 'SPAI'], loc='lower center', fancybox=True, shadow=True)
```

```
<matplotlib.legend.Legend at 0x7a8dd30bf100>
```




## Preconditioners in High Performance Computing

High Performance Computing (HPC) describes systems that have a number of distributed nodes (thousands to millions of cores) connected by a high speed network. Each node has its own memory bank and possibly multiple cores which share that memory.

Effective usage of HPC requires a balance between distributed computing and the expense of inter-node communication.

- Parallelized direct solver methods do exists (most modern implementations are) but they scale poorly with the number of nodes.
- Iterative solvers parallelize better but require preconditioning for effective convergence. Since the preconditioner doesn't have to be a perfect inverse, there are strategies to distribute preconditioner computation with limited inter-node communication, which is then iterated to find the actual solution.

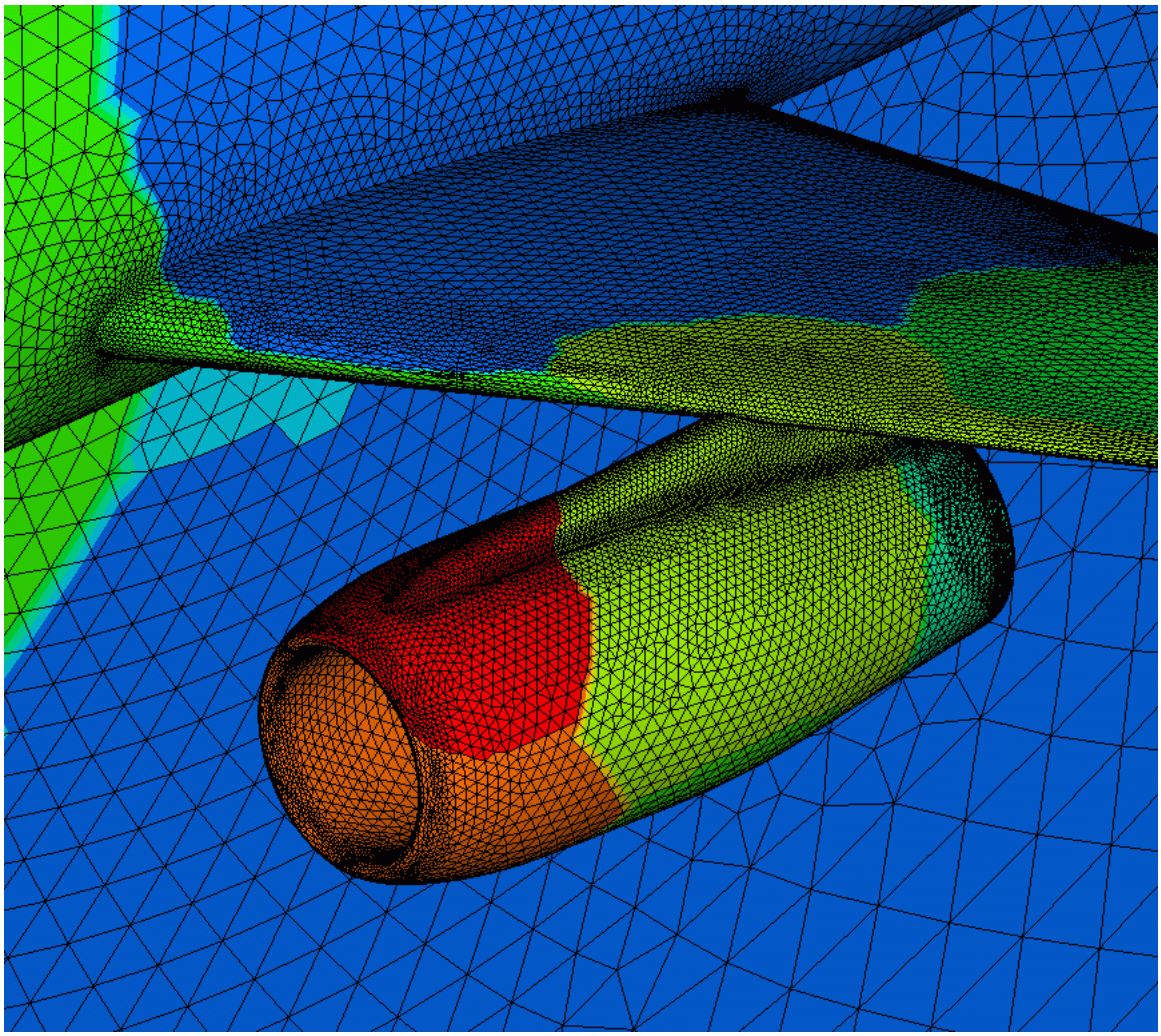
Let us now focus on solutions of differential equations over a given spatial domain, which is the most common area where these techniques are needed and encountered.

Note - the following are geometric methods, which implies passing of information of the geometry to the solver.

##Domain decomposition

Domain decomposition methods are the most intuitive approach since they literally partition the full domain (geometry) into smaller chunks. Each chunk is assigned to a node (or subgroup) and solved locally.

For example, a finite element representation of a plane wing, with colours denoting different *chunks*:



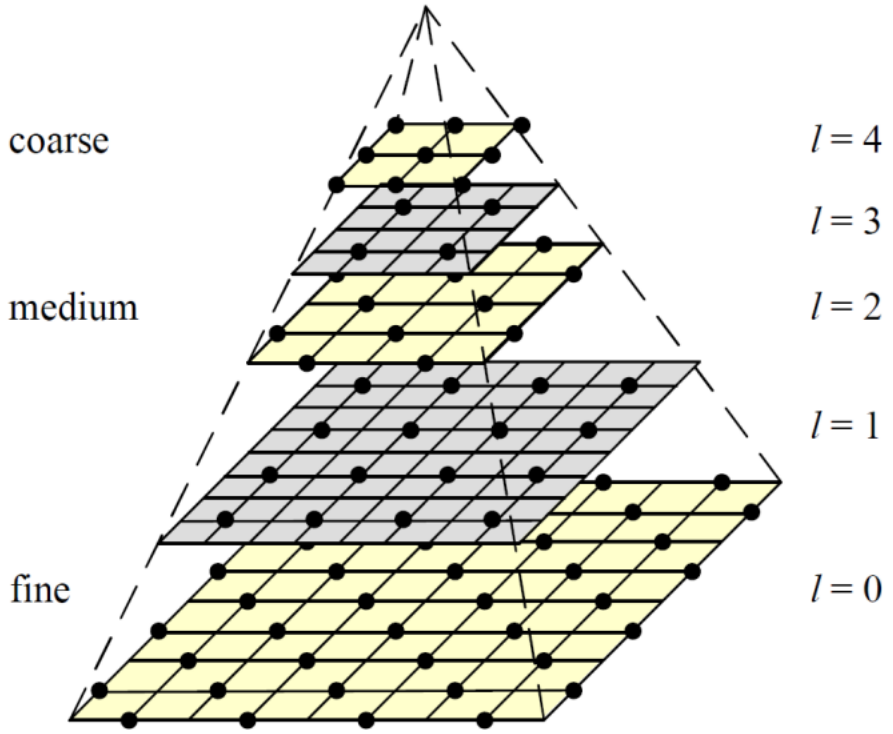
A common approach is the *Addative Schwartz method* which defines *levels of overlap* between adjacent subdomains. The overlap enables the recombination of subdomains by passing information between subdomains.

The nice thing about this is that we can do something like a direct solve on the subdomains which is great for black-box robustness! Since direct solvers scale \(~O(n^3)\), this can be very effective although the cost of the global iterative scheme and the internode communication must be considered.

These methods suffer with physics that propogates quickly throughout the domain. E.g.: Elasticity which is quasitstatic - a small displacement on one side must cascade through all the chunks.

### Multigrid methods

Multigrid methods are a modern and tremendously powerful approach that can even achieve \(O(n)\) scaling! The idea is to solve the differential equations on a *hierarchy of discretizations*.




The fine domain is the resolution that you actually want, but the course domain can be small enough to treat with a direct solver (for robustness). The equations are treated to a few iterations (Commmonly SOR) on each lower level before being passed along.

The method naturally parallelizes; fine domains can be domain decomposed without overlap, because the communication is handled by the level above it.

The full mathematical analysis actually shows this method is best considered in terms of Fourier space as each level reduces certain wavelengths of the residual. Elliptic problems (e.g.: elasticity) are especially well treated since information can be passed throughout the domain on the coarse level.

- Geometric multigrid methods need to *know* about the geometry, not just the matrix.
- Algebraic multigrid methods can work completely on the matrix itself.

#Multiphysics

Typically, a certain types of physics (e.g.: parabolic / diffusive equations like heat and mass transport) will respond well to some, and other physics (e.g.elliptic / quasi-static equations like elasticity) will respond better to others. The question then is *how to combine them for a multiphysics equation?*

Generally, each type of physics will have an associated field and vector of unknowns. E.g.: the temperature and concentration fields represented by \(x_T\) and \(x_c\) respectively. Linear system then adopts a block structure,

\(\begin{bmatrix} A_{TT} & A_{Tc} \\ A_{cT} & A_{cc} \end{bmatrix} \begin{bmatrix} x_T \\ x_c \end{bmatrix} = \begin{bmatrix} b_T \\ b_c \end{bmatrix}\)

where the diagonal of the \(A\) matrix reflect the type of physics (heat / mass transport) and the cross-terms reflect couping (temperature dependent diffusivity / concentration dependent conductivity). These are sometimes called *operator split* methods, *physics-based preconditioners*, or *segregated solvers*.

Typically we have some ideas how to solve the diagonals, but how do we effectively precondition the block \(A\) matrix? One option is using a *block Jacobi* method:

\begin{align} P &= \begin{bmatrix} A_{TT} & 0 \ 0 & A_{cc} \end{bmatrix} \ P^{-1} &= \begin{bmatrix} A^{-1}*{TT} & 0 \ 0 & A^{-1}* {cc} \end{bmatrix} \end{align}

For a \(2x2\) block system, we may use the *block Schur complement* method:

\[\begin{split} \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix} \end{split}\]
but this has a problematic \(A^{-1}\) buried inside.

#Dr. Mike's tips

- Parallelized LU is still the go-to for most problems and should be your first approach.
- GMRES iteration is your next tool, but it needs preconditioning for effective use.
- If you are studying a well-known model, you can generally find good preconditioners in the literature. Use them!
- If not, most PDE solvers will have the ability to test different preconditioners and combine them hierarchically (COMSOL Multiphysics, PETSc*, etc).
- Addative Schwartz with LU on the subdomains is the most robust preconditioner I've found (aim for about n=10,000 per node).
- Geometric multigrid is finiky, but if you can get it working it is remarkable (Use it for elasticity!).

*Portable, Extensible Toolkit for Scientific Computation https://petsc.org/release/

# Interpolation and Curve-fitting

Interpolation and curve-fitting both deal with fitting lists curves to a list of distrete points but there are some key differences in terminology:

**Interpolation** seeks a curve that

- Goes through all the points in the inputs.
- Assumes there is no measurement error in data points
- No ambiguity in mapping x and y (no duplicate y's for a given x)
- Often used to capture the *local* behaviour

**Curve fitting** seeks a curve that

- is the *best fit* for all datapoints (in some sense)
- doesn't necessarily traverse all the datapoints
- permits ambiguity in x-y pairs
- Is more of a *global* encapsulation of the data.
- generally recovers interpolation as a 'perfect fit' under the interpolation criteria.

## Interpolation

Interpolation is a more fundamental concept since it was historically *easier* to do, either direclty as a local process and/or adding new information as it was obtained.

We need to consider a few things for interpolation:

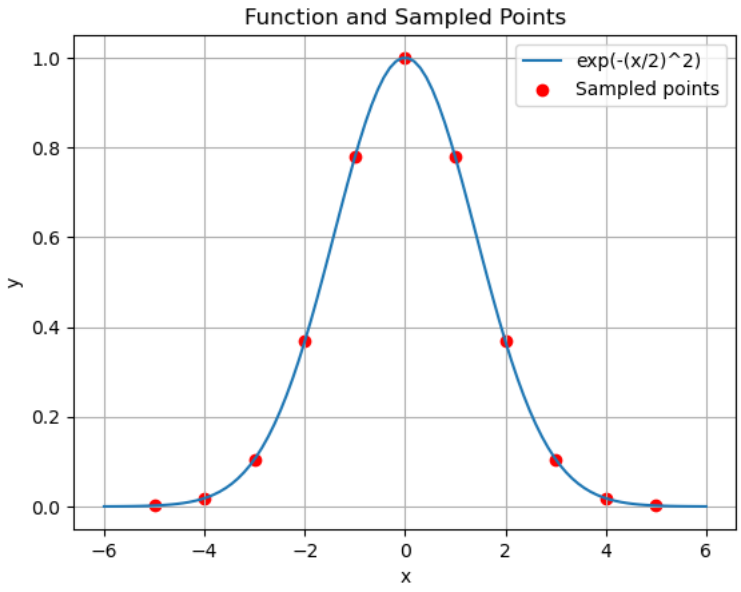
- Speed of building the model
- Speed of adding new data to the model
- Speed of execution for interpolated values
- Generalizability to N-D

The methods discussed here rely on a fundamental property of linear algebra: **It is always possible to construct a *unique* polynomial of degree \(n\) that passes through \(n + 1\) distinct data points!**

## Example: Interpolating a Gaussian curve

For illustrative purposes, let's design a toy problem for exploration:

```
# prompt: Plot the function exp(-(x/2)^2) from -5 to 5. Then sample 11 times at 1 intervals,  marking the

import numpy as np
import matplotlib.pyplot as plt

# Define the function
def f(x):
  return np.exp(-(x/2)**2)

# Create x values for plotting
x_toy = np.linspace(-6, 6, 100)
y_toy = f(x_toy)

# Sample 11 times at 1-interval intervals
x_d = np.arange(-5, 6, 1)
y_d = f(x_d)

# Plot the function and sampled points
plt.plot(x_toy, y_toy, label='exp(-(x/2)^2)')
plt.scatter(x_d, y_d, color='red', label='Sampled points')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('Function and Sampled Points')
plt.grid(True)
plt.show()

print("x_d:", x_d)
print("y_d:", y_d)
```




```
x_d: [-5 -4 -3 -2 -1  0  1  2  3  4  5]
y_d: [0.00193045 0.01831564 0.10539922 0.36787944 0.77880078 1.
 0.77880078 0.36787944 0.10539922 0.01831564 0.00193045]
```

Our goal is to use the sampled data (the red points) and recover the 'true' function (in blue) as faithfully as possible.

# Polynomial interpolation

Polynomial interpolation is a straightforward approach to interpolation.

Three methods to obtain polynomials are established here. For a given set of data, they all *must* result in the same polynomial. The difference is the means by which they are achieved, which translates to the ways that they are used.

## Lagrange Polynomial Interpolation

Legendre polynomial interpolation constructs the Legendre polynomial as, $\( y(x) = \sum_{i = 1}^n y_i P_i(x) \)$

which is a weighted sum of the Lagrange basis polynomials, \(P_i(x)\),

$$ P_i(x) = \prod_{j = 1, j\ne i}^n\frac{x - x_j}{x_i - x_j}. $$

N.B.: \(\prod\) means *the product of*, like \(\sum\) means *the sum of*.

### Lagrange basis polynomials

By construction,

- \(P_i(x_j) = 1\) when \(i = j\)
- \(P_i(x_j) = 0\) when \(i \ne j\).

### Example: Find and plot the Lagrange basis polynomials

Use the data: *x = [0, .5, 2] y = [1, 3, 2]*

$$\begin{eqnarray} P_1(x) &=& \frac{(x - x_2)(x - x_3)}{(x_1-x_2)(x_1-x_3)} = \frac{(x - 1)(x - 2)}{(0-1)(0-2)} = \frac{1}{2}(x^2 - 3x + 2),\ P_2(x) &=& \frac{(x - x_1)(x - x_3)}{(x_2-x_1)(x_2-x_3)} = \frac{(x - 0)(x - 2)}{(1-0)(1-2)} = -x^2 + 2x,\ P_3(x) &=& \frac{(x - x_1)(x - x_2)}{(x_3-x_1)(x_3-x_2)} = \frac{(x - 0)(x - 1)}{(2-0)(2-1)} = \frac{1}{2}(x^2 - x). \end{eqnarray}$$

Plot each polynomial and verify the property that \(P_i(x_j) = 1\) when \(i = j\) and \(P_i(x_j) = 0\) when \(i \ne j\).

```
# prompt: show me the legendre basis polynomials for the data aove using the numpy.polynomial.legendre L

import numpy as np
import matplotlib.pyplot as plt
from numpy.polynomial import legendre

# Data points
x = [0, .5, 2]
y = [1, 3, 2]

# Calculate the Lagrange basis polynomials
n = len(x)
P = []
for i in range(n):
  numerator = 1
  denominator = 1
  for j in range(n):
    if i != j:
      numerator = np.polymul(numerator, np.poly1d([1, -x[j]]))
      denominator = denominator * (x[i] - x[j])
  P.append(np.poly1d(np.polydiv(numerator, denominator)[0]))


# Plot the Lagrange basis polynomials
x_plot = np.linspace(-1, 3, 100)

for i in range(n):
    y_plot = P[i](x_plot)
    plt.plot(x_plot, y_plot, label=f'P_{i+1}(x)')

plt.scatter(x, [1] * len(x), color='black')
plt.scatter(x, [0] * len(x), color='red')
plt.scatter(x, [0] * len(x), color='red')

plt.xlabel('x')
plt.ylabel('P_i(x)')
plt.title('Lagrange Basis Polynomials')
plt.legend()
plt.grid(True)
plt.show()
```

#### Assembling the polynomial

Since \(P_{i\ne j}=0\), and \(P_{i = j}=1\), it is trivial to see that for \( y(x) = \sum_{i = 1}^n \omega_i P_i(x) \), the coefficients are simply:

\[ y(x) = \sum_{i = 1}^n y_i P_i(x) \]

```
# prompt: Plot the legendre polynomial from the basis above

import matplotlib.pyplot as plt
import numpy as np
# Construct the Lagrange polynomial
L = np.poly1d(0)
for i in range(n):
  L = L + y[i] * P[i]

# Plot the Lagrange polynomial
y_plot = L(x_plot)
plt.plot(x_plot, y_plot, label='L(x)')
plt.scatter(x, y, color='red', label='Data points')
plt.xlabel('x')
plt.ylabel('L(x)')
plt.title('Lagrange Polynomial Interpolation')
plt.legend()
plt.grid(True)
plt.show()
```

#### Analysis

We can observe some notes:

- For \(n\) data points we necessarily produce a unique polynominal that crosses each one.

- If we have two measurements at the same input, $x_i = x_j$, $P_i =\sim \frac{1}{0}$ which is undefined *unless* $x_i=x_j$ and $y_i=y_j$ in which case the data pair is redundant and can be removed.
- Each evalulation of $P(x)$ involves $n-1$ products, and $L(x)$ is the sum of $n$ bases, therefore evaluation is $O(n^2)$
- Adding new data means restarting the compuation.

```
# prompt: Use sympy to fit a lagrange polynomial to the data above (with some extra points)

import sympy as sp

x = [0, 1, 2, 3, 4]
y = [1, 3, 2, 5, 7]

n = len(x)
x_sym = sp.Symbol('x')

L = 0
for i in range(n):
    term = y[i]
    for j in range(n):
        if i != j:
            term *= (x_sym - x[j]) / (x[i] - x[j])
    L += term


print(L)

print('which is an ugly way of writing out:')
print(L.simplify())
```

**Error**

It can be shown that the error in the interpolation is,

$$ y^{true}(x)-y(x) = \frac{[x-x_1][x-x_2][x-x_3]...[x-x_n]}{(n+1)!} f^{(n+1)}(\xi) $$

where $\xi$ is in the interval $(x_0, x_n)$.

Since for $n$ datapoints there is a unique polynomial of degree $n-1$, which can be expressed as a Lagrange polynomial, **this analysis is universal to all polynomial interpolations!**. The main takeaway is that:

*The further a data point is from $x$, the more it contributes to the error.*

##Barycentric Lagrange Interpolation

Let's try to improve the performance of Lagrange Interpolation. Let:

$$ \Omega(x) = \prod_{j = 1}^n [x - x_j] $$

and the *barycentric weights*, $w_i$:

$$ w_i = \prod_{j = 1, j\ne i}^n\frac{1}{x_i - x_j}. $$

and write:

$$ P_i(x) = \Omega(x) \frac{w_i}{x - x_j}. $$

and factor the $\Omega$ out of the sum:

$$ y(x) = \Omega(x) \sum_{j = 1}^n \frac{w_i}{x - x_j} y_i. $$

which is $O(n)$ for evaluation. Calculation of $w_i$ can be formulated recursively, such that each $w_i$ takes $O(n)$ and the full takes $O(n^2)$ with updates n.

NB: The weights depend only on $x_i$, not $y_i$ - this means if we are measuring multiple functions on the same spacing, we can reuse the weights, leading to substantial computaitonal savings.

The benefit being that the calucation of the $\omega_i$, $O(n^2)$ is precomputed.

**Barycentric formula**

We can write one more form which is commonly implemented. Let's add one more piece of data:

$$ 1 = \sum_{j=0}^n P_j = \Omega(x) \sum_{j=0}^n \frac{w_j}{x-x_j}$$

then we divide the previous function and write:

$$ y(x) = \frac{\sum_{j = 0}^n \frac{w_i}{x - x_j} y_i}{\sum_{j = 0}^n \frac{w_i}{x - x_j}} $$

where we have cancelled $\Omega$! Besides elegance, his avoids an issue when evaluating $x\rightarrow x_i$ where roundoff can cause subtractive cancellation. Since the term appears in the numerator and denominator this cancels out!

## Newton's divided difference method

Newton's polynomial interpolation has the form:

$$ y(x) = a_0 + a_1[x-x_0] + a_2 [x-x_0][x-x_1] + \dots + a_n[x-x_0][x-x_1]\dots[x-x_n]$$

which has the advantage of $O(n)$ evaluations due to recursion and nested multiplication. E.g. for 4 terms,

$$ y(x) = a_0 + [x-x_0] \bigg[a_1 + [x-x_1] \big[a_2 + [x-x_2] a_3 \big] \bigg] $$

Newton's method is also known as the **divided differences**

> This was the algorithm was used to calculate function tables like logarithms and trignometry functions. It was then the basis for the *difference engine*, an early mechanical calculator.

Let's pick a data point to start at. Say $y(x_0) = a_0 = y_0$, $$a_0 = y_0$$

Add the next data point: $y(x_1) = a_0 + a_1(x_1-x_0) = y_1$, or:

$$a_1 = \frac{y_1 - y_0}{x_1 - x_0}$$

Now, insert data point $(x_2, y_2)$,

$$a_2 = \frac{\frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0}$$

and similarly,

$$a_3 = \frac{\frac{\frac{y_3-y_2}{x_3-x_2} - \frac{y_2 - y_1}{x_2-x_1}}{x_3 - x_1} - \frac{\frac{y_2-y_1}{x_2-x_1}-\frac{y_1 - y_0}{x_1 - x_0}}{x_2-x_0}}{x_3 - x_0}$$

Notice the recurrsion and the division of the differences.

Let's generalize this. Define the two-argument function:

$$ y[x_1, x_0] = \frac{y_1 - y_0}{x_1 - x_0}$$

and the ternary recursively:

$$ y[x_2, x_1, x_0] = \frac{\frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0} = \frac{y[x_2,x_1] - y[x_1,x_0]}{x_2-x_1}$$

The $n-nary$ function is:

$$ y[x_k, x_{k-1}, \dots, x_{1}, x_0] = \frac{y[x_k, x_{k-1}, \dots, x_{2}, x_2] - y[x_{k-1}, x_{k-2}, \dots, x_{1}, x_0]}{x_k-x_0}$$

We can visualize this is in a *tableau*: $

$$ \begin{array}{cccccc} x_0 & y_0 \\ & & y[x_1,x_0] \\ x_1 & y_1 & & y[x_2, x_1,x_0]\\ & & y[x_2,x_1] & & y[x_3, x_2, x_1,x_0]\\ x_2 & y_2 & & y[x_3, x_2,x_1] & & y[x_4, x_3, x_2, x_1,x_0]\\ & & y[x_3,x_2] & & y[x_4, x_3, x_2, x_1]\\ x_3 & y_3 & & y[x_4, x_3,x_2]\\ & & y[x_4,x_3] \\ x_4 & y_4 \end{array} $$

$

where element is the difference of the two to the left. Alternately, it is sometimes written in the form,

\[\begin{split} \begin{array}{c||cccccc} x_0 & y_0 & 0 & 0 & 0 & 0\\ x_1 & y_1 & y[x_1,x_0] & 0 & 0 & 0\\ x_2 & y_2 & y[x_2,x_1] & y[x_2, x_1,x_0] & 0 & 0 \\ x_3 & y_3 & y[x_3,x_2] & y[x_3, x_2,x_1] & y[x_3, x_2, x_1,x_0] & 0 \\ x_4 & y_4 & y[x_4,x_3] & y[x_4, x_3,x_2] & y[x_4, x_3, x_2, x_1] & y[x_4, x_3, x_2, x_1,x_0] \end{array} \end{split}\]
Note that the diagonal is the coefficients that we need, i.e. \(a_0, a_1, a_2, a_3, a_4\) for the polynomial.

## Direct solution

Lastly, there is a direct solution method that only became really poractical with the advent of modern computing since it focusses on linear systes:

Consider fitting a function

\[ y(x) = a_n x^n + a_{n-1} x^{n-1} \dots a_2 x^2 + a_1 x +a_0\]

since

\(y(x_i) = a_n x_i^n + a_{n-1} x_i^{n-1} \dots a_2 x_i^2 + a_1 x_i +a_0 = y_i\)

we can write out in matrix form,

\[\begin{split} \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^m \\ 1 & x_2 & x_2^2 & \cdots & x_2^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} \end{split}\]
where the matrix of coefficients is called a Vandermonde matrix. This system can be solved for \(a_i\) with a dense linear solver. The issue with this method is that the system is notoriously ill-conditioned and roundoff error accumulates rapidly for large \(n\).

#Example: Interpolate our toy problem

Let us know examine our toy problem. Since all the polynomial interpolation functions generate the same unique polynomial, any will suffice:

```
# prompt: Interpolate the data in x_d and y_d using numpy Legendre, and plot along with the original cur

import matplotlib.pyplot as plt
import numpy as np
# Interpolate using numpy Legendre
coefficients = legendre.legfit(x_d, y_d, len(x_d) - 1)
legendre_polynomial = legendre.Legendre(coefficients)

# Create x values for plotting the interpolated polynomial
x_interp = np.linspace(-5.5, 5.5, 200)
y_interp = legendre_polynomial(x_interp)


# Plot the original curve, sampled points, and interpolated polynomial
plt.plot(x_toy, y_toy, label='exp(-(x/2)^2)')
plt.scatter(x_d, y_d, color='red', label='Sampled points')
plt.plot(x_interp, y_interp, label='Legendre Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('Function, Sampled Points, and Legendre Interpolation')
plt.grid(True)
plt.show()
```

YIKES!

This is an example of *Runge's phenomenon*: That even for a seeminlgly ideal case of equalspaced samples, higher order polynomials can show huge oscillations between samples!

- The order that the datapoints are added is arbitrary but will result in a different tableau (with the same diagonal).
- We can build this matrix / tableau diagonal-by-diagonal which means adding new data points doesn't require recalculation of the others.

- Each new diagonal (datapoint) takes $O(n)$ so assembly of the tableau takes $~O(n^2)$.
- Evaluation of f(x) takes $O(n)$
- These coefficients are independant of $x$

### Summary

Let's recap and generalize:

- For any $n$ points there is a polynomial that fits it, but because of Runge's phenomenon you don't want to use that!
- Piecewise polynomials are *stiffer* and avoids Runge's phenomenon, but smoothness causes issues for N-D So what do we do? Standard pacakges offer simplistic but pragmatic interpolators (optimized for either rectangular or irregular grids) :
- Nearest ND interpolator: Find the nearest data point and use that.
- Linear ND interpolators: For each input, a triangulation finds the nearest data points and a linear barycentric Lagrange interpolation is performed.

Neither of these are completely satisfactory, so we will have to respost to more advanced methods.

## Cubic splines

Splines were formulated to relieve these oscillations by piecing together a series of lower-order polynomials and requiring *smoothness*. Consider a polynomial over the interval between $x_i$ and $x_{i+1}$, and assert:

- $y(x_i) = y_i$
- $y(x_{i+1}) = y_{i+1}$
- $y'(x_i)$ be continuous
- $y''(x_i)$ be continuous

with these 4 constraints, it is clear we are looking for cubic functions, and therefore these splines are *piecewise cubic curves*.

We will be describing the splines in terms of the *knots*, $k_i$ which parameterize the curves. For these splines, these knots are the second derivatives at a point $x_i$.

To find the coefficients of the cubic splines, consider that the second derivative is linear and represent it with a 2-point Lagrange interpolation: \begin{align} y''*{i, i+1} &= k_i P_i(x) + k*{i+1} P_{i+1}(x) \ &= \frac{k_i [x-x_{i+1}] + k_{i+1} [x-x_i]}{x_i-x_{i+1}} \end{align}

Using the constraints above we end up with:

$$k_{i-1}[x_{i-1}-x_i] + 2 k_i [x_{i-1} - x_{i+1}] + k_{i+1}[x_i-x_{i+1}] = 6\left[ \frac{y_{i-1}-y_i}{x_{i-1}-x_i} - \frac{y_{i}-y_{i+1}}{x_{i}-x_{i+1}} \right]$$
which is a tridiagonal matrix!

$$\begin{split} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 4 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 4 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 4 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 4 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \end{bmatrix} \begin{bmatrix} k_1 \\ k_2\\ k_3\\ k_4\\ k_5\\ k_6\\ k_7\\ k_8\\ k_9\\ k_{10}\\ k_{11} \end{bmatrix} = \begin{bmatrix} 0\\ -0.424 \\ -1.052 \\ -0.891 \\ 1.138 \\ 2.654 \\ 1.138 \\ -0.891 \\ -1.052 \\ -0.424 \\ 0 \end{bmatrix} \end{split}$$

```python
# prompt: Do a cubic spline of x_d and y_d and plot against the original function from -5.5 to 5.5

import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import CubicSpline

# Create a cubic spline interpolation
cs = CubicSpline(x_d, y_d)

# Create x values for plotting the interpolated spline
x_interp = np.linspace(-6, 6, 200)
y_interp = cs(x_interp)

# Plot the original curve, sampled points, and interpolated spline
plt.plot(x_toy, y_toy, label='exp(-(x/2)^2)')
plt.scatter(x_d, y_d, color='red', label='Sampled points')
plt.plot(x_interp, y_interp, label='Cubic Spline Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('Function, Sampled Points, and Cubic Spline Interpolation')
plt.grid(True)
plt.show()
```

### Analysis of cubic splines

We note:

- Cubic splines are *stiffer* in that they don't have high-frequency oscillations (thus avoiding Runge's phenomenon).
- The concept of *smoothness* is easy in 1D, but what does it mean for 2D+? How would you ensure continuity along an edge?
- Specifying *smoothness* as part of the goals going in suggest this is more of a global scheme. This requires simultaneous linears systems to be solved.

## Radial Basis Functions

Radial basis functions are an n-dimensional interpolation technique that doesn't rely on polynomials. Rather, we define a radial basis function, called a *kernel*, applied to each data point:

$$\varphi_i(||x-x_i||)$$

Commonly, we say $\varphi_i(x=x_i)\equiv 1$.

The kernel only depends on the Euclidian distance between the associated data point, $x_i$ and the evaluation point $x$ (and are therefore *radial*).

The interpolation function $y(x)$ is the weighted sum of the $N$ kernels:

$$y(x) = \sum_i^N \omega_i \varphi_i(||x-x_i||)$$

To determine the weights $w_i$, we use the data points we have. Consider the $i$'th datapoints,

$$y(x_i) = \sum_j^N \omega_j \varphi_j(||x-x_j||)=y_i$$

and applied to all N data points generates a linear system:

$$\begin{split} \begin{bmatrix} \phi(\|x_1 - x_1\|) & \phi(\|x_1 - x_2\|) & \cdots & \phi(\|x_1 - x_n\|) \\ \phi(\|x_2 - x_1\|) & \phi(\|x_2 - x_2\|) & \cdots & \phi(\|x_2 - x_n\|) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(\|x_n - x_1\|) & \phi(\|x_n - x_2\|) & \cdots & \phi(\|x_n - x_n\|) \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \end{split}$$

which we know how to solve!

Kernels are defined with $r = \| x-x_i\|$ and a tuning parameter $\epsilon$. Some common simple kernels are:

| Kernel | Formula |
|---|---|
| Gaussian | $e^{-\epsilon^2 r^2}$ |
| Inverse quadratic | $\frac{1}{1+[\epsilon r ]^2}$ |
| Inverse multiquadric | $\frac{1}{\sqrt{1+[\epsilon r ]^2}}$ |

Determination of optimal $\epsilon$ is a nuanced question, but a good rule of thumb is to use the average distance between samples.

$\epsilon = avg |x_i-x_j|$

Let's see the kernels:

```
# prompt: Plot the above radial basis functions for epsilon = 1

import numpy as np
import matplotlib.pyplot as plt

# Define the radial basis functions
def gaussian(r, epsilon):
  return np.exp(-(epsilon * r)**2)

def inverse_quadratic(r, epsilon):
  return 1 / (1 + (epsilon * r)**2)

def inverse_multiquadric(r, epsilon):
  return 1 / np.sqrt(1 + (epsilon * r)**2)

epsilon = 1

# Create a range of r values
r_values = np.linspace(0, 10, 100)

# Calculate the function values for each kernel
gaussian_values = gaussian(r_values, epsilon)
inverse_quadratic_values = inverse_quadratic(r_values, epsilon)
inverse_multiquadric_values = inverse_multiquadric(r_values, epsilon)

# Plot the radial basis functions
plt.plot(r_values, gaussian_values, label='Gaussian')
plt.plot(r_values, inverse_quadratic_values, label='Inverse Quadratic')
plt.plot(r_values, inverse_multiquadric_values, label='Inverse Multiquadric')

plt.xlabel('r')
plt.ylabel('φ(r)')
plt.title('Radial Basis Functions (epsilon = 1)')
plt.legend()
plt.grid(True)
plt.show()
```

> In general, $\varphi_i(r=0)$ is not necessarily $1$, and $\varphi(r \rightarrow \infty) \ne 0$, but this requires one more key factor to implement robustly.

### Example - Our Toy problem from last lecture (Gaussian sampled at 10 points, equally spaced)

```
#Sampled gaussian

import numpy as np
import matplotlib.pyplot as plt

# Define the function
def f(x):
  return np.exp(-(x/2)**2)

def gaussian(r, epsilon):
  return np.exp(-(epsilon * r)**2)

# Create x values for plotting
x_toy = np.linspace(-6, 6, 100)
y_toy = f(x_toy)

# Sample 11 times at 1-interval intervals
x_d = np.arange(-5, 6, 1)
y_d = f(x_d)
```

```
# prompt: Construct gaussian radial basis functions and fit to y_d and x_d

import matplotlib.pyplot as plt
import numpy as np
# Create a matrix of the radial basis functions
phi_matrix = np.zeros((len(x_d), len(x_d)))

epsilon = 1

for i in range(len(x_d)):
  for j in range(len(x_d)):
    phi_matrix[i, j] = gaussian(np.abs(x_d[i] - x_d[j]), epsilon)

#~~ How do we solve for w_i?
# Take a look at the matrix!



# #~~ Answer
# np.set_printoptions(precision=2, suppress=True)
# print(phi_matrix)
# weights = np.linalg.solve(phi_matrix, y_d)
# #~~

# Define the interpolation function
def interpolation_function(x, weights, x_d, epsilon):
  y = 0
  for i in range(len(x_d)):
    y += weights[i] * gaussian(np.abs(x - x_d[i]), epsilon)
  return y

# Interpolate y_fit
y_fit = [interpolation_function(x, weights, x_d, epsilon) for x in x_toy]

# Plot the results
plt.plot(x_toy, y_toy, label='Original Function')
plt.scatter(x_d, y_d, color='red', label='Data Points')
plt.plot(x_toy, y_fit, label='Interpolation', linestyle='--')

plt.xlabel('x')
plt.ylabel('y')
plt.title('Radial Basis Function Interpolation (Gaussian Kernel)')
plt.legend()
plt.grid(True)
plt.show()
```

Note this is a great result, but it works because the true function tends to zero outside of the data samples.

### Example - 2D gaussian

```python
# prompt: Generate a function exp(-x^2-7*y^2)*sin(x)*cos(8y), sample 100 times and fit using gaussian ra

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

# Define the function
def f(x, y):
  return np.exp(-x**2 - 7*y**2) * np.sin(x) * np.cos(8*y)

# Create a grid of x and y values
x = np.linspace(-3, 3, 100)
y = np.linspace(-3, 3, 100)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)

# Sample 100 times
num_samples = 100
x_samples = np.random.uniform(-3, 3, num_samples)
y_samples = np.random.uniform(-3, 3, num_samples)
z_samples = f(x_samples, y_samples)

# Plot the original function and data samples
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, alpha=0.5)
ax.scatter(x_samples, y_samples, z_samples, color='red', marker='o', s=20)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Original Function and Data Samples')
plt.show()



# Define the radial basis function (Gaussian)
def gaussian_2d(x1, y1, x2, y2, epsilon):
    r = np.sqrt((x1 - x2)**2 + (y1 - y2)**2)
    return np.exp(-(epsilon * r)**2)

def phi_matrix_2d(x_samples, y_samples, epsilon):
    num_samples = len(x_samples)
    phi_matrix = np.zeros((num_samples, num_samples))
    for i in range(num_samples):
        for j in range(num_samples):
            phi_matrix[i, j] = gaussian_2d(x_samples[i], y_samples[i], x_samples[j], y_samples[j], epsil
    return phi_matrix

phi_matrix = phi_matrix_2d(x_samples, y_samples, epsilon = 1)

# #~~ Examine the condition number of the matrix before inverting it.
# print('The matrix condition number is, ', np.linalg.cond(phi_matrix))
# distances = []
# for i in range(num_samples):
#   for j in range(i + 1, num_samples):
#     distance = np.sqrt((x_samples[i] - x_samples[j])**2 + (y_samples[i] - y_samples[j])**2)
#     distances.append(distance)
# average_distance = np.mean(distances)
# eps = average_distance
# phi_matrix = phi_matrix_2d(x_samples, y_samples, epsilon= eps)
# print('The matrix condition number is, ', np.linalg.cond(phi_matrix))
# #~~~

# Calculate the weights
weights = np.linalg.solve(phi_matrix, z_samples)



# Define the interpolation function
def interpolation_function_2d(x, y, weights, x_samples, y_samples, epsilon):
    z = 0
    for i in range(num_samples):
        z += weights[i] * gaussian_2d(x, y, x_samples[i], y_samples[i], epsilon)
    return z

# Interpolate Z_fit
Z_fit = np.zeros((100, 100))
for i in range(100):
    for j in range(100):
        Z_fit[i, j] = interpolation_function_2d(x[i], y[j], weights, x_samples, y_samples, epsilon=3)
```

```
# Plot the fitted function
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z_fit, cmap=cm.coolwarm, alpha=0.5)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title('Fitted Function (RBF)')
plt.show()
```

# Curve fitting

Generally, we will have some datapoint pairs \(x_i, y_i\). We will have some function \(y\) with parameters \(a\) that will be evaluated at a point \(x\): \(y(a, x)\).

Our goal is to find the set of parameters \(a\) that gives us the *best fit* of the data. Commonly, this implies minimizing the squared error between the prediction and the data,

\[y(a,x_i)-y_i = r_i \]

where \(r\) is the residual vector. The *least squares fit* is formulated as finding \(a\) so as to minimize \(\| r\|\).

In general, this is an optimization problem (much more complicated than you would expect!) since \(y(a,x)\) can be complex.

## Linear least squares regression

Let's look at the *simpler* case first; in particular where \(y\) is the sum of basis functions weighted by \(a\) (e.g.: polynomial interpolation, radial basis functions… can you think of another one?).

In this case, \(y=Ax\), and the data is inserted into \(b\).

> CAUTION! We have swapped notation to follow suit with standard practice! The matrix A is the function of the 'position', \(x\), and the parameters are in the vector!

Given a matrix system, $\( A x = b\)\( where \)A\( is an \)m\times n\( matrix, \)x\( is \)n\(, and \)b\( is \)
\(. We cannot solve this for an exact \)x\( with our normal techniques since \)A$ is rectangular, not square.

Recalling the residual is \(Ax-b\), let's broaden our concept to a 'solution' to say we want to minimize the (norm of the) residual.

\[Min_x \ of \ \| Ax-b\|=\frac{1}{2} [Ax-b]^T [Ax-b]\]

Setting \(\frac{d}{dx} = 0\), we get:

\begin{align} 2 A^T [Ax-b] &= 0 \ A^T A x &= A^T b \ x &= [A^T A]^{-1} A^T b \ x &= A^† b \end{align}

where \(A^†=[A^T A]^{-1} A^T\) is called the *(Moore-Penrose) pseudoinverse* of \(A\). The pseudoinverse is defined for any rectangular matrix. Note \(A^T A\) is necessarily square, and is generally invertible.

- The pseudoinverse is defined for any rectangular matrix
- When used to solve \(Ax=b\) it results in the *best fit* (in the least squares sense)
- Since the ultimate *minimum* is \(0\), the pseudoinverse is the true inverse for an exactly solvable system.

### Conditioning of a rectangular matrix

The determinant of a rectangual matrix is undefined, but we can resort to the definition of the condition number: $
\(cond(A) = \|A\| \|A^\dagger\|\)$

Terminology

- A **consistent** system of equations has a solution that satisfies *all* the equations.
- An **inconsistent** system has no solution that satisfies all equations simultaneously.
- - **Overdetermined** systems have more equations that unknowns which is typical of curve fitting. These systems are inconsistent in that there is *no simultaneous solution*, but a solution does exists that *simultaneously minimizes the error*.
- - **Underdetermined** systems have fewer equations than unknowns and are also inconsistent but with an *infinite* number of solutions. E.g.: Parallel lines / 2 equations with 3 variables.

**Example: An overdetermined, consistent linear system (our headscratcher from the early lectures!)**

$20 c + 50 t = 700$

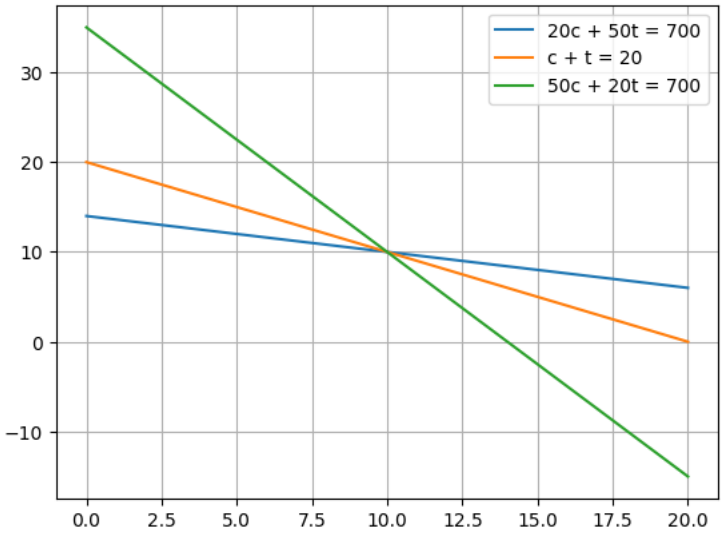
$c+t = 20$

$50 c + 20 t = 700$

which gives the linear system:

$$\begin{split}\begin{bmatrix} 20 & 50 \\ 1 & 1 \\ 50 & 20 \end{bmatrix} \begin{bmatrix} c \\ t \end{bmatrix} = \begin{bmatrix} 700 \\ 20 \\ 700 \end{bmatrix} \end{split}$$

```
#Plot it!

import matplotlib.pyplot as plt
import numpy as np

# Define the x values
x = np.linspace(0, 20, 100)

# Calculate the y values for the first equation (20c + 50t = 700)
y1 = (700 - 20 * x) / 50

# Calculate the y values for the second equation (c + t = 20)
y2 = 20 - x

# Calculate the y values for the third equation (50c + 20t = 700)
y3 = (700 - 50 * x) / 20

# Plot the lines
plt.plot(x, y1, label='20c + 50t = 700')
plt.plot(x, y2, label='c + t = 20')
plt.plot(x, y3, label='50c + 20t = 700')

# Add a grid
plt.grid(True)
plt.legend()
plt.show()
```




```
#The arrays are:
# ~~ Question - what is the linear system and how do we solve it?

A = np.array([[20, 50], [1, 1], [50, 20]])
b = np.array([700, 20, 700])

#x = np.linalg.solve(A, b)

M = np.linalg.inv(A.T @ A)@A.T
print(np.linalg.pinv(A))
print(M-np.linalg.pinv(A))
print(M@b)




# ~~ Answer
# A = np.array([[20, 50], [1, 1], [50, 20]])
# b = np.array([700, 20, 700])

# #x = np.linalg.solve(A, b)

# M = np.linalg.inv(A.T @ A)@A.T
# print(M)
# print(np.linalg.pinv(A))
# print(M-np.linalg.pinv(A))
# print(M@b)
```

```
[[-0.00952672  0.000204    0.02380661]
 [ 0.02380661  0.000204   -0.00952672]]
[[-1.73472348e-18  2.71050543e-19  6.93889390e-18]
 [ 1.04083409e-17 -3.25260652e-19 -1.73472348e-18]]
[10. 10.]
```

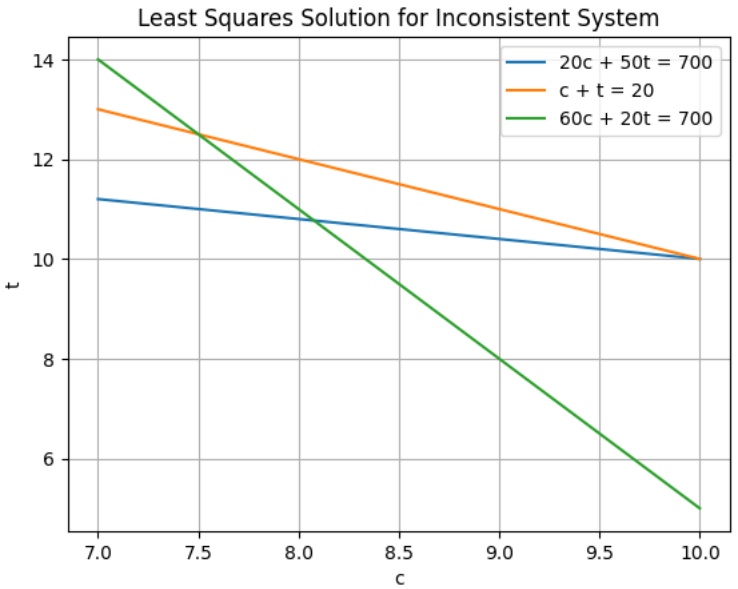
**Example: An overdetermined, inconsistent linear system**

$20 c + 50 t = 700$

$ c+t = 20$

$ 60 c + 20 t = 700$

```
import matplotlib.pyplot as plt
import numpy as np

# Define the x values
x = np.linspace(7, 10, 100)

# Calculate the y values for the first equation (20c + 50t = 700)
y1 = (700 - 20 * x) / 50

# Calculate the y values for the second equation (c + t = 20)
y2 = 20 - x

# Calculate the y values for the third equation (60c + 20t = 700)
y3 = (700 - 60 * x) / 20

# Plot the lines
plt.plot(x, y1, label='20c + 50t = 700')
plt.plot(x, y2, label='c + t = 20')
plt.plot(x, y3, label='60c + 20t = 700')

plt.xlabel('c')
plt.ylabel('t')
plt.title('Least Squares Solution for Inconsistent System')
plt.legend()
plt.grid(True)
plt.show()
```

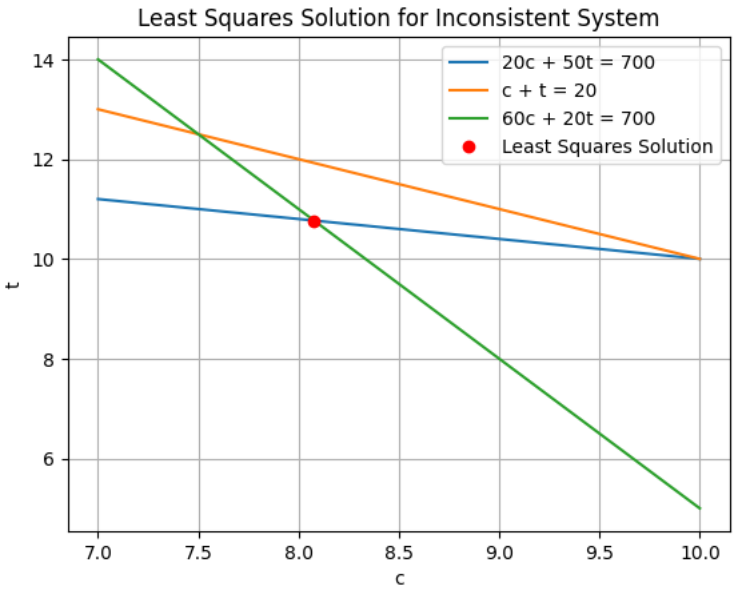



Where do you think he solution is going to be?

```
A = np.array([[20, 50], [1, 1], [60, 20]])
b = np.array([700, 20, 700])

#np.linalg.(A, b)
x_lsq = np.linalg.pinv(A)@b
print(x_lsq)
x_lsq,_,_,_ = np.linalg.lstsq(A,b)
print(x_lsq)

# Plot the lines
plt.plot(x, y1, label='20c + 50t = 700')
plt.plot(x, y2, label='c + t = 20')
plt.plot(x, y3, label='60c + 20t = 700')


plt.plot(x_lsq[0], x_lsq[1], 'ro', label='Least Squares Solution')
plt.xlabel('c')
plt.ylabel('t')
plt.title('Least Squares Solution for Inconsistent System')
plt.legend()
plt.grid(True)
plt.show()
```

```
[ 8.07704251 10.76953789]
[ 8.07704251 10.76953789]
```

```
<ipython-input-7-52bb65e377be>:7: FutureWarning: `rcond` parameter will change to the default of machine
To use the future default and silence this warning we advise to pass `rcond=None`, to keep using the old,
  x_lsq,_,_,_ = np.linalg.lstsq(A,b)
```




Was this what you were expecting?

## Weights

As with most approximate methods, the moment we start to move away from an *exact solution* subtle effects start to show up!

Notice we are minimizing the *residuals* but there is a subtle problem with the problem definition above:

$$\begin{split}\begin{align} 20 c + 50 t &= 700 \\ c+t &= 20 \\ 60 c + 20 t &= 700 \end{align} \end{split}$$

The coefficients of the second equation is about an order of magnitude lower than the others. Of course this system is equivilant to:

$$\begin{split}\begin{align} 20 c + 50 t &= 700 \\ 10c+10t &= 200 \\ 60 c + 20 t &= 700 \end{align} \end{split}$$

or even

$$\begin{split}\begin{align} 0.2 c + 0.5 t &= 7 \\ c+t &= 20 \\ 0.6 c + 0.2 t &= 7 \end{align} \end{split}$$

What does this remind you of?

Jacobi (diagonal) preconditioning!

$$\begin{split}P^{-1} = \begin{bmatrix} W_{1} & 0 & 0 \\ 0 & W_{2} & 0 \\ 0 & 0 & W_{3} \end{bmatrix} \end{split}$$

If we define $r_1, r_2, r_3$:

$$\begin{split}\begin{align} 20 c + 50 t -700 &= r_1 \\ c+t -20 &= r_2 \\ 60 c + 20 t -700 &= r_3 \end{align} \end{split}$$

We say the residuals are / can be *weighted*, i.e.: the least squares problem becomes,

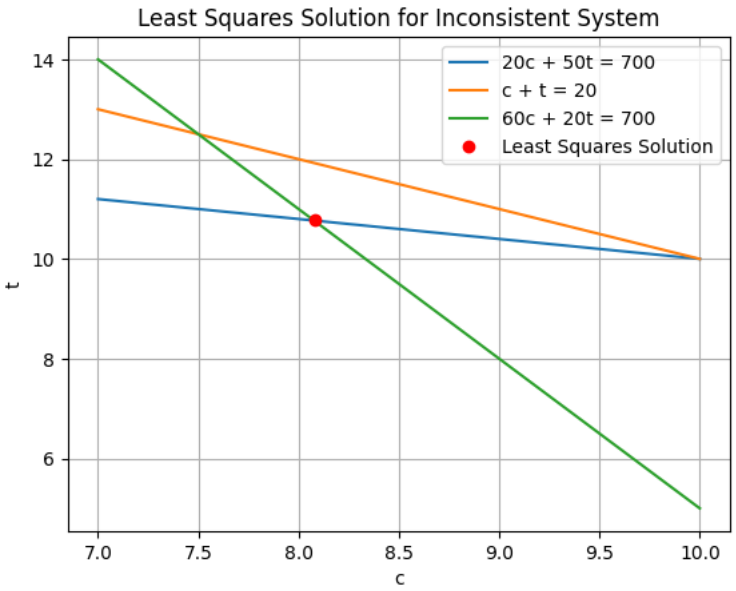
$$Min_x \ of \ \sum W_i^2 r_i$$

Let's code it!

```python
A = np.array([[20, 50], [1, 1], [60, 20]])
b = np.array([700, 20, 700])

#~~ Question: What's the preconditioner? How do we apply it?
Pi = np.diag([1,7,1])
print(Pi)
A = Pi@A
b = Pi@b
###

#~~ Answer
# Pi = np.diag([1,70,1])
# print(Pi)
# A = Pi@A
# b = Pi@b
###



x_lsq,_,_,_ = np.linalg.lstsq(A,b)

print(x_lsq)

# Plot the lines
plt.plot(x, y1, label='20c + 50t = 700')
plt.plot(x, y2, label='c + t = 20')
plt.plot(x, y3, label='60c + 20t = 700')


plt.plot(x_lsq[0], x_lsq[1], 'ro', label='Least Squares Solution')
plt.xlabel('c')
plt.ylabel('t')
plt.title('Least Squares Solution for Inconsistent System')
plt.legend()
plt.grid(True)
plt.show()
```

```
[[1 0 0]
 [0 7 0]
 [0 0 1]]
[ 8.08267345 10.78401744]
```

```
<ipython-input-13-2aa1c6412f60>:20: FutureWarning: `rcond` parameter will change to the default of machir
To use the future default and silence this warning we advise to pass `rcond=None`, to keep using the old,
  x_lsq,_,_,_ = np.linalg.lstsq(A,b)
```




Weights are an excellent way to introduce measurement uncertainty into your fit!

## Polyfit

Let's return to our polynomial fitting armed with our new tool, and use it to separate the order of the polynomial from the number of data points.

An \(n\) th degree polynomial,

$$ y(x) = a_n x^n + a_{n-1} x^{n-1} \dots a_2 x^2 + a_1 x +a_0 $$

can be applied to \(m\) data points,

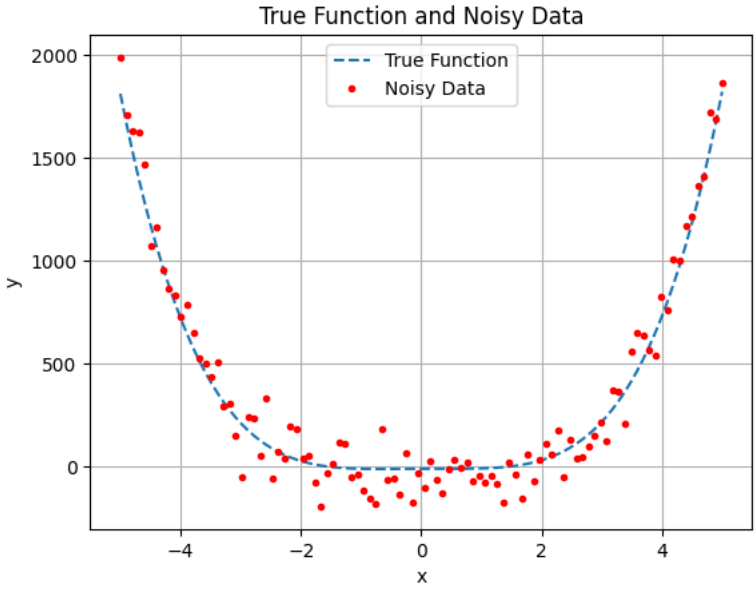
$y(x_i) = a_n x_i^n + a_{n-1} x_i^{n-1} \dots a_2 x_i^2 + a_1 x_i +a_0 = y_i$

to generate an $m \times n$ matrix, multiplied by an $n$ vector of polynomial coefficients to equal an $m$ vector of data:

$$\begin{split} \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} \end{split}$$

### Example: Determine the coefficients of a cubic polynomial

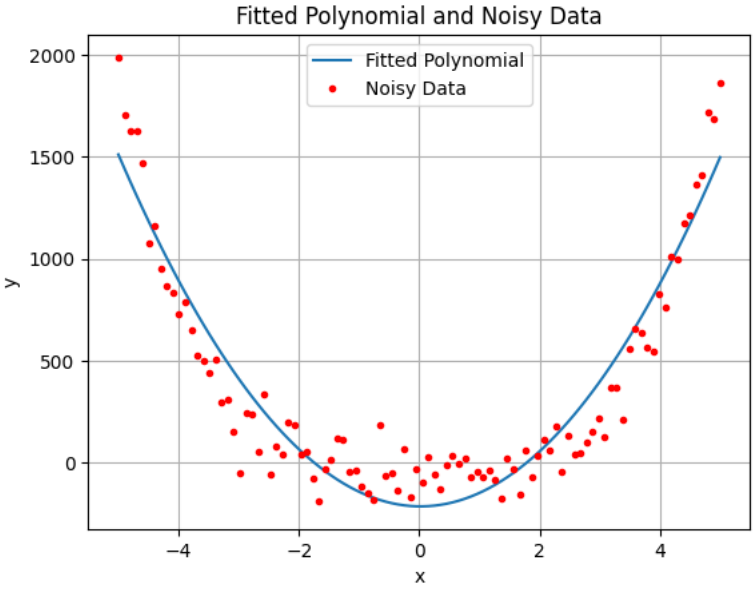
```
# prompt: generate 100 samples of the function 3x^4-2x^2+x-9 with +-100 noise. Plot the true curve with (

import matplotlib.pyplot as plt
import numpy as np

# Generate x values
x = np.linspace(-5, 5, 100)

# Define the true function
def true_function(x):
  return 3 * x**4 - 2 * x**2 + x - 9

# Calculate the true y values
y_true = true_function(x)

# Generate noisy data
np.random.seed(0)  # For reproducibility
noise = np.random.normal(0, 100, 100)
x_data = np.linspace(-5, 5, 100)
y_data = true_function(x_data) + noise

# Plot the true curve and the data
plt.plot(x, y_true, '--', label='True Function')
plt.plot(x_data, y_data, 'ro', markersize=3, label='Noisy Data')

plt.xlabel('x')
plt.ylabel('y')
plt.title('True Function and Noisy Data')
plt.legend()
plt.grid(True)
plt.show()
```




```
# prompt: Generate the vermonde matrix for a cubic polynomial, invert it using pinv and the y data to fi

import matplotlib.pyplot as plt
import numpy as np
# Generate the Vandermonde matrix for a cubic polynomial
n = 3  # Degree of the polynomial
X = np.vander(x_data, n + 1)

# Calculate the coefficients using the pseudoinverse
coefficients = np.linalg.pinv(X) @ y_data

coeffs = np.polyfit(x_data, y_data, 3)
print('Coefficients calculated manually', coefficients, '\n')
print('Coefficients calculated with polyfit', coeffs)

# Generate y values for the fitted polynomial
fitted_polynomial = np.poly1d(coeffs)
y_fitted = fitted_polynomial(x)


# Plot the fitted polynomial along with the data
plt.plot(x, y_fitted, label='Fitted Polynomial')
plt.plot(x_data, y_data, 'ro', markersize=3, label='Noisy Data')

plt.xlabel('x')
plt.ylabel('y')
plt.title('Fitted Polynomial and Noisy Data')
plt.legend()
plt.grid(True)
plt.show()
```

```
[[-1.25000000e+02  2.50000000e+01 -5.00000000e+00  1.00000000e+00]
 [-1.17576257e+02  2.40001020e+01 -4.89898990e+00  1.00000000e+00]
 [-1.10452422e+02  2.30206101e+01 -4.79797980e+00  1.00000000e+00]
 [-1.03622311e+02  2.20615243e+01 -4.69696970e+00  1.00000000e+00]
 [-9.70797404e+01  2.11228446e+01 -4.59595960e+00  1.00000000e+00]
 [-9.08185260e+01  2.02045710e+01 -4.49494949e+00  1.00000000e+00]
 [-8.48324846e+01  1.93067034e+01 -4.39393939e+00  1.00000000e+00]
 [-7.91154325e+01  1.84292419e+01 -4.29292929e+00  1.00000000e+00]
 [-7.36611859e+01  1.75721865e+01 -4.19191919e+00  1.00000000e+00]
 [-6.84635612e+01  1.67355372e+01 -4.09090909e+00  1.00000000e+00]
 [-6.35163748e+01  1.59192939e+01 -3.98989899e+00  1.00000000e+00]
 [-5.88134431e+01  1.51234568e+01 -3.88888889e+00  1.00000000e+00]
 [-5.43485822e+01  1.43480257e+01 -3.78787879e+00  1.00000000e+00]
 [-5.01156087e+01  1.35930007e+01 -3.68686869e+00  1.00000000e+00]
 [-4.61083388e+01  1.28583818e+01 -3.58585859e+00  1.00000000e+00]
 [-4.23205888e+01  1.21441690e+01 -3.48484848e+00  1.00000000e+00]
 [-3.87461751e+01  1.14503622e+01 -3.38383838e+00  1.00000000e+00]
 [-3.53789141e+01  1.07769615e+01 -3.28282828e+00  1.00000000e+00]
 [-3.22126221e+01  1.01239669e+01 -3.18181818e+00  1.00000000e+00]
 [-2.92411154e+01  9.49137843e+00 -3.08080808e+00  1.00000000e+00]
 [-2.64582103e+01  8.87919600e+00 -2.97979798e+00  1.00000000e+00]
 [-2.38577232e+01  8.28741965e+00 -2.87878788e+00  1.00000000e+00]
 [-2.14334705e+01  7.71604938e+00 -2.77777778e+00  1.00000000e+00]
 [-1.91792685e+01  7.16508520e+00 -2.67676768e+00  1.00000000e+00]
 [-1.70889334e+01  6.63452709e+00 -2.57575758e+00  1.00000000e+00]
 [-1.51562817e+01  6.12437506e+00 -2.47474747e+00  1.00000000e+00]
 [-1.33751297e+01  5.63462912e+00 -2.37373737e+00  1.00000000e+00]
 [-1.17392938e+01  5.16528926e+00 -2.27272727e+00  1.00000000e+00]
 [-1.02425902e+01  4.71635547e+00 -2.17171717e+00  1.00000000e+00]
 [-8.87883529e+00  4.28782777e+00 -2.07070707e+00  1.00000000e+00]
 [-7.64184545e+00  3.87970615e+00 -1.96969697e+00  1.00000000e+00]
 [-6.52543700e+00  3.49199061e+00 -1.86868687e+00  1.00000000e+00]
 [-5.52342628e+00  3.12468115e+00 -1.76767677e+00  1.00000000e+00]
 [-4.62962963e+00  2.77777778e+00 -1.66666667e+00  1.00000000e+00]
 [-3.83786338e+00  2.45128048e+00 -1.56565657e+00  1.00000000e+00]
 [-3.14194388e+00  2.14518927e+00 -1.46464646e+00  1.00000000e+00]
 [-2.53568745e+00  1.85950413e+00 -1.36363636e+00  1.00000000e+00]
 [-2.01291045e+00  1.59422508e+00 -1.26262626e+00  1.00000000e+00]
 [-1.56742922e+00  1.34935211e+00 -1.16161616e+00  1.00000000e+00]
 [-1.19306008e+00  1.12488522e+00 -1.06060606e+00  1.00000000e+00]
 [-8.83619379e-01  9.20824406e-01 -9.59595960e-01  1.00000000e+00]
 [-6.32923460e-01  7.37169677e-01 -8.58585859e-01  1.00000000e+00]
 [-4.34788658e-01  5.73921028e-01 -7.57575758e-01  1.00000000e+00]
 [-2.83031313e-01  4.31078461e-01 -6.56565657e-01  1.00000000e+00]
 [-1.71467764e-01  3.08641975e-01 -5.55555556e-01  1.00000000e+00]
 [-9.39143501e-02  2.06611570e-01 -4.54545455e-01  1.00000000e+00]
 [-4.41874103e-02  1.24987246e-01 -3.53535354e-01  1.00000000e+00]
 [-1.61032836e-02  6.37690032e-02 -2.52525253e-01  1.00000000e+00]
 [-3.47830926e-03  2.29568411e-02 -1.51515152e-01  1.00000000e+00]
 [-1.28826269e-04  2.55076013e-03 -5.05050505e-02  1.00000000e+00]
 [ 1.28826269e-04  2.55076013e-03  5.05050505e-02  1.00000000e+00]
 [ 3.47830926e-03  2.29568411e-02  1.51515152e-01  1.00000000e+00]
 [ 1.61032836e-02  6.37690032e-02  2.52525253e-01  1.00000000e+00]
 [ 4.41874103e-02  1.24987246e-01  3.53535354e-01  1.00000000e+00]
 [ 9.39143501e-02  2.06611570e-01  4.54545455e-01  1.00000000e+00]
 [ 1.71467764e-01  3.08641975e-01  5.55555556e-01  1.00000000e+00]
 [ 2.83031313e-01  4.31078461e-01  6.56565657e-01  1.00000000e+00]
 [ 4.34788658e-01  5.73921028e-01  7.57575758e-01  1.00000000e+00]
 [ 6.32923460e-01  7.37169677e-01  8.58585859e-01  1.00000000e+00]
 [ 8.83619379e-01  9.20824406e-01  9.59595960e-01  1.00000000e+00]
 [ 1.19306008e+00  1.12488522e+00  1.06060606e+00  1.00000000e+00]
 [ 1.56742922e+00  1.34935211e+00  1.16161616e+00  1.00000000e+00]
 [ 2.01291045e+00  1.59422508e+00  1.26262626e+00  1.00000000e+00]
 [ 2.53568745e+00  1.85950413e+00  1.36363636e+00  1.00000000e+00]
 [ 3.14194388e+00  2.14518927e+00  1.46464646e+00  1.00000000e+00]
 [ 3.83786338e+00  2.45128048e+00  1.56565657e+00  1.00000000e+00]
 [ 4.62962963e+00  2.77777778e+00  1.66666667e+00  1.00000000e+00]
 [ 5.52342628e+00  3.12468115e+00  1.76767677e+00  1.00000000e+00]
 [ 6.52543700e+00  3.49199061e+00  1.86868687e+00  1.00000000e+00]
 [ 7.64184545e+00  3.87970615e+00  1.96969697e+00  1.00000000e+00]
 [ 8.87883529e+00  4.28782777e+00  2.07070707e+00  1.00000000e+00]
 [ 1.02425902e+01  4.71635547e+00  2.17171717e+00  1.00000000e+00]
 [ 1.17392938e+01  5.16528926e+00  2.27272727e+00  1.00000000e+00]
 [ 1.33751297e+01  5.63462912e+00  2.37373737e+00  1.00000000e+00]
 [ 1.51562817e+01  6.12437506e+00  2.47474747e+00  1.00000000e+00]
 [ 1.70889334e+01  6.63452709e+00  2.57575758e+00  1.00000000e+00]
 [ 1.91792685e+01  7.16508520e+00  2.67676768e+00  1.00000000e+00]
 [ 2.14334705e+01  7.71604938e+00  2.77777778e+00  1.00000000e+00]
 [ 2.38577232e+01  8.28741965e+00  2.87878788e+00  1.00000000e+00]
 [ 2.64582103e+01  8.87919600e+00  2.97979798e+00  1.00000000e+00]
 [ 2.92411154e+01  9.49137843e+00  3.08080808e+00  1.00000000e+00]
```

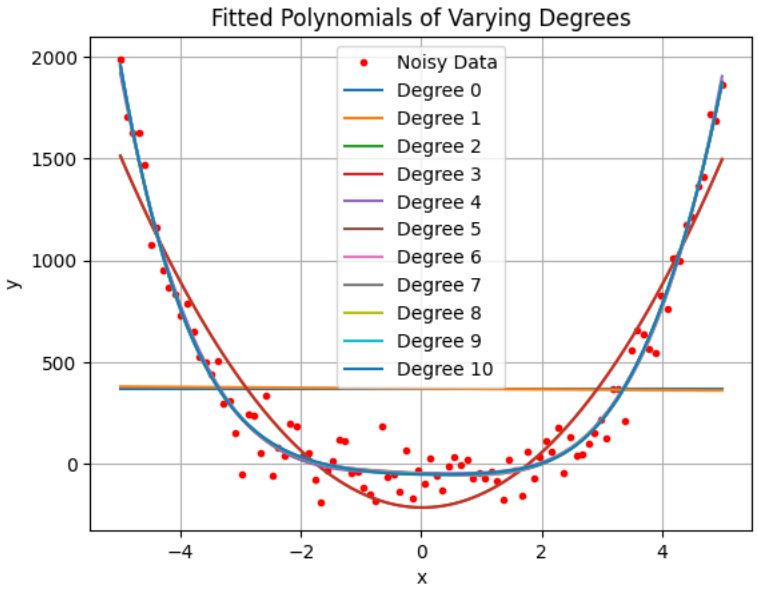
```
 [ 3.22126221e+01  1.01239669e+01  3.18181818e+00  1.00000000e+00]
 [ 3.53789141e+01  1.07769615e+01  3.28282828e+00  1.00000000e+00]
 [ 3.87461751e+01  1.14503622e+01  3.38383838e+00  1.00000000e+00]
 [ 4.23205888e+01  1.21441690e+01  3.48484848e+00  1.00000000e+00]
 [ 4.61083388e+01  1.28583818e+01  3.58585859e+00  1.00000000e+00]
 [ 5.01156087e+01  1.35930007e+01  3.68686869e+00  1.00000000e+00]
 [ 5.43485822e+01  1.43480257e+01  3.78787879e+00  1.00000000e+00]
 [ 5.88134431e+01  1.51234568e+01  3.88888889e+00  1.00000000e+00]
 [ 6.35163748e+01  1.59192939e+01  3.98989899e+00  1.00000000e+00]
 [ 6.84635612e+01  1.67355372e+01  4.09090909e+00  1.00000000e+00]
 [ 7.36611859e+01  1.75721865e+01  4.19191919e+00  1.00000000e+00]
 [ 7.91154325e+01  1.84292419e+01  4.29292929e+00  1.00000000e+00]
 [ 8.48324846e+01  1.93067034e+01  4.39393939e+00  1.00000000e+00]
 [ 9.08185260e+01  2.02045710e+01  4.49494949e+00  1.00000000e+00]
 [ 9.70797404e+01  2.11228446e+01  4.59595960e+00  1.00000000e+00]
 [ 1.03622311e+02  2.20615243e+01  4.69696970e+00  1.00000000e+00]
 [ 1.10452422e+02  2.30206101e+01  4.79797980e+00  1.00000000e+00]
 [ 1.17576257e+02  2.40001020e+01  4.89898990e+00  1.00000000e+00]
 [ 1.25000000e+02  2.50000000e+01  5.00000000e+00  1.00000000e+00]]
Coefficients calculated manually [ 6.28230705e-02  6.87425501e+01 -2.93452380e+00 -2.14197455e+02]

Coefficients calculated with polyfit [ 6.28230705e-02  6.87425501e+01 -2.93452380e+00 -2.14197455e+02]
```




**Example 2: Find a series of ‘best fit polynomials’**

```python
# prompt: parameterize polyfit for degrees 0 to 10, plotting each on the same plot along with the sample

import matplotlib.pyplot as plt
import numpy as np

# Generate x values
x = np.linspace(-5, 5, 100)

# Define the true function
def true_function(x):
  return 3 * x**4 - 2 * x**2 + x - 9

# Generate noisy data
np.random.seed(0)  # For reproducibility
noise = np.random.normal(0, 100, 100)
x_data = np.linspace(-5, 5, 100)
y_data = true_function(x_data) + noise

# Plot the true curve and the data
plt.plot(x_data, y_data, 'ro', markersize=3, label='Noisy Data')

for degree in range(11):
    coeffs = np.polyfit(x_data, y_data, degree)
    fitted_polynomial = np.poly1d(coeffs)
    y_fitted = fitted_polynomial(x)
    plt.plot(x, y_fitted, label=f'Degree {degree}')
    residuals = y_data - fitted_polynomial(x_data)
    residual_norm = np.linalg.norm(residuals)
    print(f"Degree {degree}: Residual Norm = {residual_norm:.2f}, Coefficients = {[round(c, 2) for c in (

plt.xlabel('x')
plt.ylabel('y')
plt.title('Fitted Polynomials of Varying Degrees')
plt.legend()
plt.grid(True)
plt.show()
```

```
Degree 0: Residual Norm = 5515.03, Coefficients = [370.23]
Degree 1: Residual Norm = 5514.73, Coefficients = [-1.97, 370.23]
Degree 2: Residual Norm = 1759.53, Coefficients = [68.74, -1.97, -214.2]
Degree 3: Residual Norm = 1759.49, Coefficients = [0.06, 68.74, -2.93, -214.2]
Degree 4: Residual Norm = 923.52, Coefficients = [3.03, 0.06, 2.62, -2.93, -45.63]
Degree 5: Residual Norm = 917.81, Coefficients = [-0.08, 3.03, 2.38, 2.62, -15.58, -45.63]
Degree 6: Residual Norm = 917.57, Coefficients = [0.01, -0.08, 2.8, 2.38, 4.56, -15.58, -47.98]
Degree 7: Residual Norm = 916.86, Coefficients = [-0.0, 0.01, 0.1, 2.8, 0.23, 4.56, -9.51, -47.98]
Degree 8: Residual Norm = 916.67, Coefficients = [-0.0, -0.0, 0.05, 0.1, 2.15, 0.23, 7.55, -9.51, -50.1]
Degree 9: Residual Norm = 916.49, Coefficients = [0.0, -0.0, -0.02, 0.05, 0.45, 2.15, -2.0, 7.55, -5.65,
Degree 10: Residual Norm = 916.28, Coefficients = [0.0, 0.0, -0.01, -0.02, 0.24, 0.45, 0.53, -2.0, 12.3,
```




Interesting points: The curve actually is 4th order but because additional terms will always reduce the error, it is not trivial to tell which is the *best*, best fit!

### Example 3: Consider the condition number of the Vermonde matrix for increasing n

```
# prompt: print the condition number for an increasing series of vermonde matricies on 10 data points

import numpy as np
# Generate x values
x_data = np.linspace(-5, 5, 10)

for degree in range(1, 11):
    # Generate the Vandermonde matrix for a given degree
    X = np.vander(x_data, degree + 1)

    # Calculate the condition number
    condition_number = np.linalg.cond(X)

    print(f"Degree {degree}: Condition Number = {condition_number:.2f}")
```

```
Degree 1: Condition Number = 3.19
Degree 2: Condition Number = 20.60
Degree 3: Condition Number = 94.89
Degree 4: Condition Number = 583.63
Degree 5: Condition Number = 2841.25
Degree 6: Condition Number = 17743.56
Degree 7: Condition Number = 95457.72
Degree 8: Condition Number = 648343.57
Degree 9: Condition Number = 5082996.99
Degree 10: Condition Number = 25360712.21
```

Recall that as the condition number strays from 1, numerical algorithms deteriorate. This is why low-order polynomials are more numerically robust to fit than high order!

## Radial Basis Functions revisted!

The modern implementation of RBFs accoutns for the *global* trend of the data through a polynomial least squares fit alongside normal RBFs for local features.

$$ \begin{align} y(x) &= \sum_i^N \omega_i \varphi_i(\|x-x_i\|) + \sum_i^N P_i(x_i) b_i \end{align} $$

Where $P_i$ is an order $n\lt m$ polynomial. The Numpy RBFInterpolator object fits this equation to:

$$\begin{split} \begin{align} [\Phi(x_i, x_j) -\lambda I]\omega +P(x_i) b &= y_i \\ P(x_i)^T \omega &=0 \end{align} \end{split}$$

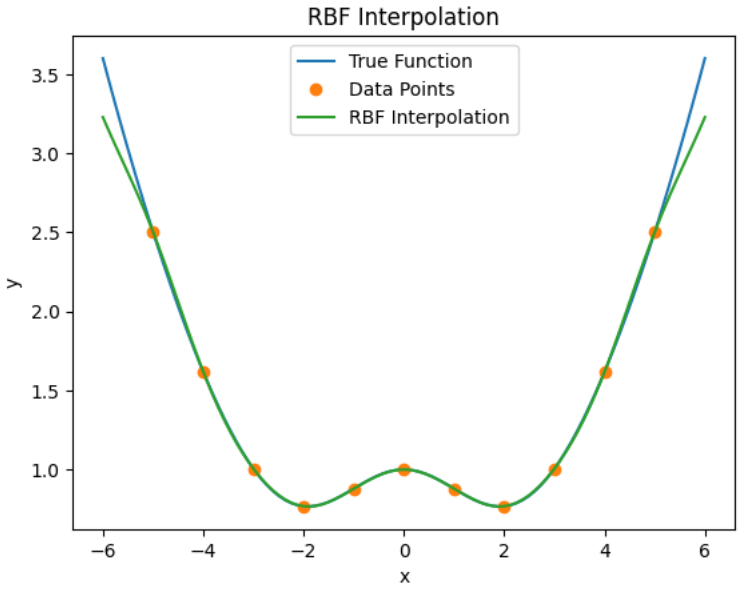
where $\lambda = 0$ recovers an exact fit and $\lambda \gt 0$ effecitvely shifts the fitting of the $x_i=x_j$ terms to the bestfit polynomial.

### Example: Toy gaussian over a quadratic

```
#Sampled gaussian

import numpy as np
import matplotlib.pyplot as plt

# Define the function
def f(x):
  return np.exp(-(x/2)**2)+.1*x**2

# Create x values for plotting
x_toy = np.linspace(-6, 6, 100)
y_toy = f(x_toy)

# Sample 11 times at 1-interval intervals
x_d = np.arange(-5, 6, 1)
y_d = f(x_d)
```

```
# prompt: Use a numpy scipy.interpolate.RBFInterpolator over x_d and y_d and plot the result against the

import matplotlib.pyplot as plt
from scipy.interpolate import RBFInterpolator

# Create an RBFInterpolator object
print(len(np.array([y_d]).T))
rbf = RBFInterpolator(np.array([x_d]).T, y_d.T, kernel='gaussian', epsilon=1, degree=2)

# Interpolate at the x_toy values
y_rbf = rbf(np.array([x_toy]).T)

# Plot the results
plt.plot(x_toy, y_toy, label='True Function')
plt.plot(x_d, y_d, 'o', label='Data Points')
plt.plot(x_toy, y_rbf, label='RBF Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('RBF Interpolation')
plt.show()
```

```
11
```




Open in Colab

Goals:

- Understand the nature of the root finding problem.
- Use standard tools for rootfinding of polynomials
- Understand bracketting root finding methods
- Understand open root finding methods

# Root finding

The roots (*aka zeros*) of a function are values of function arguments forwhich the function is zero:

Find $x$ such that:

$$ f(x) = 0 $$

It can become complicated when we consider vector $\vec{x}$ and even $\vec{f}$, which may seem complicated at first, but consider a special case of finding the roots of $\vec{f}(\vec{x})$ is our familiar linear system, $A \vec{x} -\vec{b} = \vec{0}$. This topic is mearly the generalization to nonlinear functions.

# Roots of some nonlinear functions

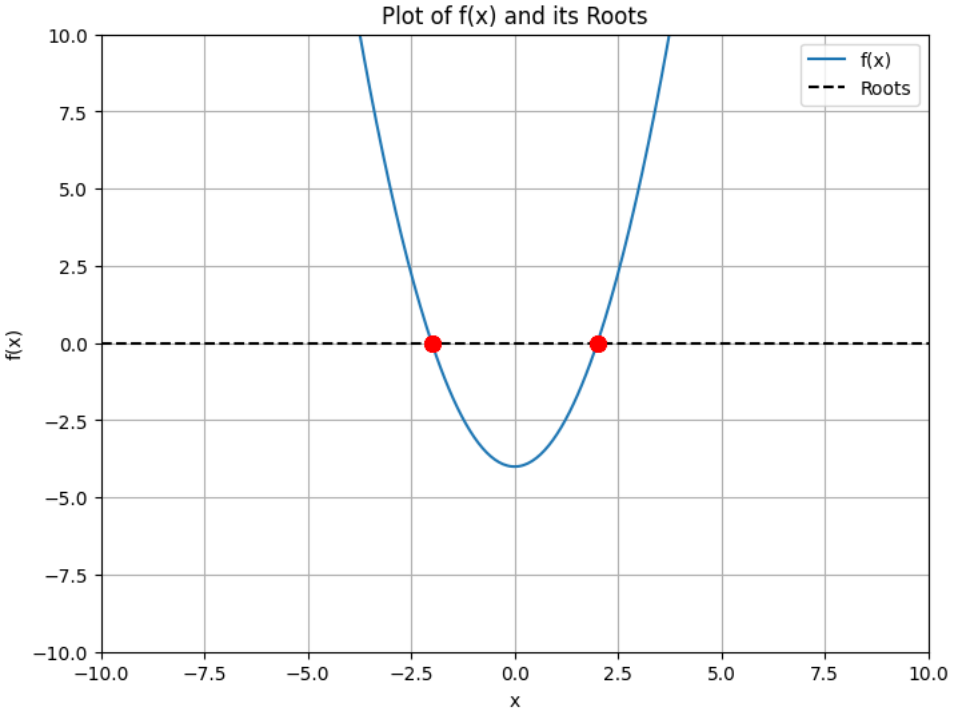
Let's build some intuition by exploring some type of roots in 1D functions using the *graphical method*: Plot the function and examine where it crosses the x-axis.

NB: Note the structure of the code below - Since we don't know *a priori* where the roots will be, we have to take a series of initial guesses and cross our finger…. and even then we may fail to find them all!

```
# prompt: Define a function that takes a function, plots it with xrange -10 to 10 and y range -10 to 10 w
# NB: Modified from original output

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import root

def plot_and_find_roots(func):
  """Plots a function and finds its roots using fsolve.

  Args:
    func: The function to plot and find roots for.
  """

  x = np.linspace(-10, 10, 400)
  y = func(x)

  plt.figure(figsize=(8, 6))
  plt.plot(x, y, label='f(x)')
  plt.axhline(y=0, color='black', linestyle='--')  # Plot the x-axis
  plt.xlabel('x')
  plt.ylabel('f(x)')
  plt.title('Plot of f(x) and its Roots')
  plt.xlim([-10, 10])
  plt.ylim([-10, 10])

  x0s = np.arange(-10,10,1)
  for x0 in x0s:
    r = root(func, x0=x0)
    if r.success:
      plt.plot(r.x, r.fun, 'ro', markersize=8)  # Plot root with a red dot
    plt.legend(['f(x)', 'Roots'])

  plt.grid(True)
  plt.show()
```

## Example 1: Real roots - \(x^2-4\)

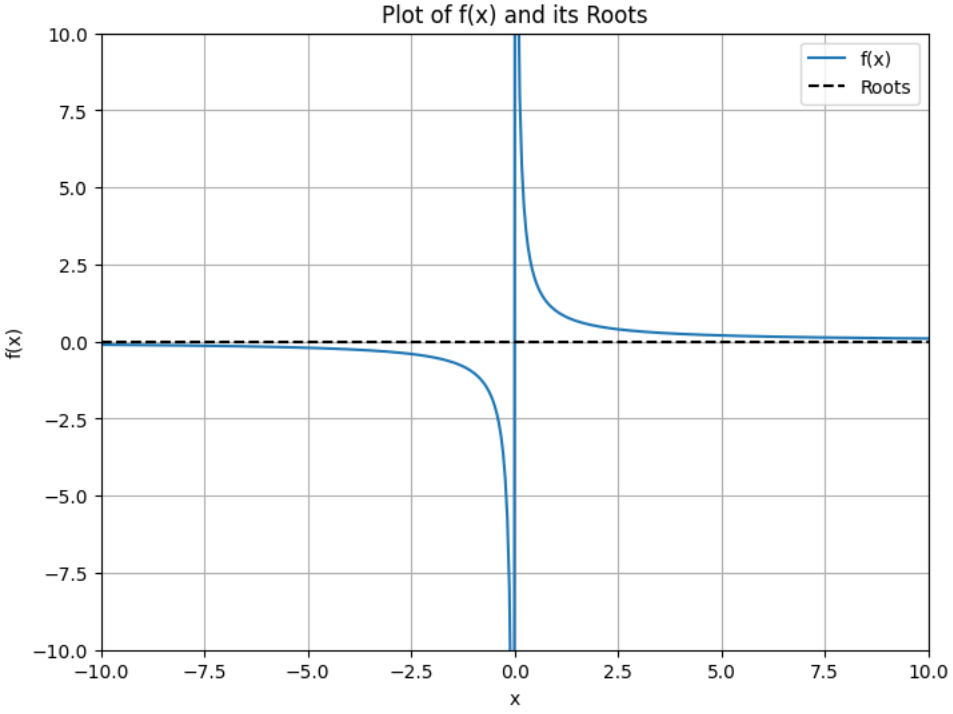
```
plot_and_find_roots(lambda x: x**2-4)
```




## Example 2: No roots - \(1/x\)

```
plot_and_find_roots(lambda x: 1/x)
```

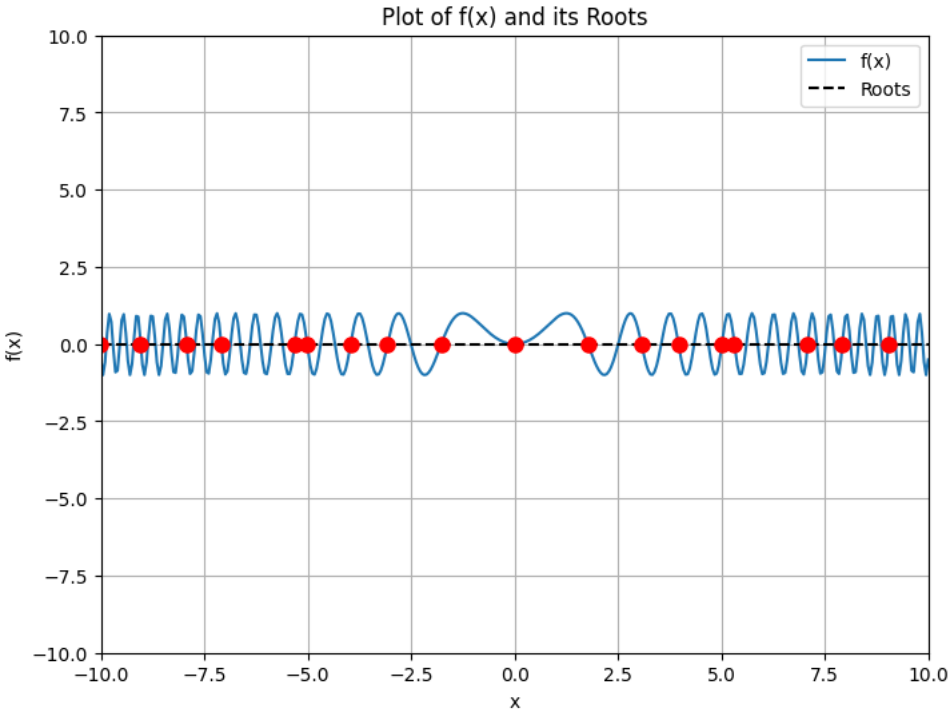
```
<ipython-input-4-f7fd98d7a758>:1: RuntimeWarning: divide by zero encountered in divide
  plot_and_find_roots(lambda x: 1/x)
```




Noting that the vertical line is a plotting artifact.

## Example 3: Infinite roots \(sin(x^2)\)

```
plot_and_find_roots(lambda x: np.sin(x**2))
```

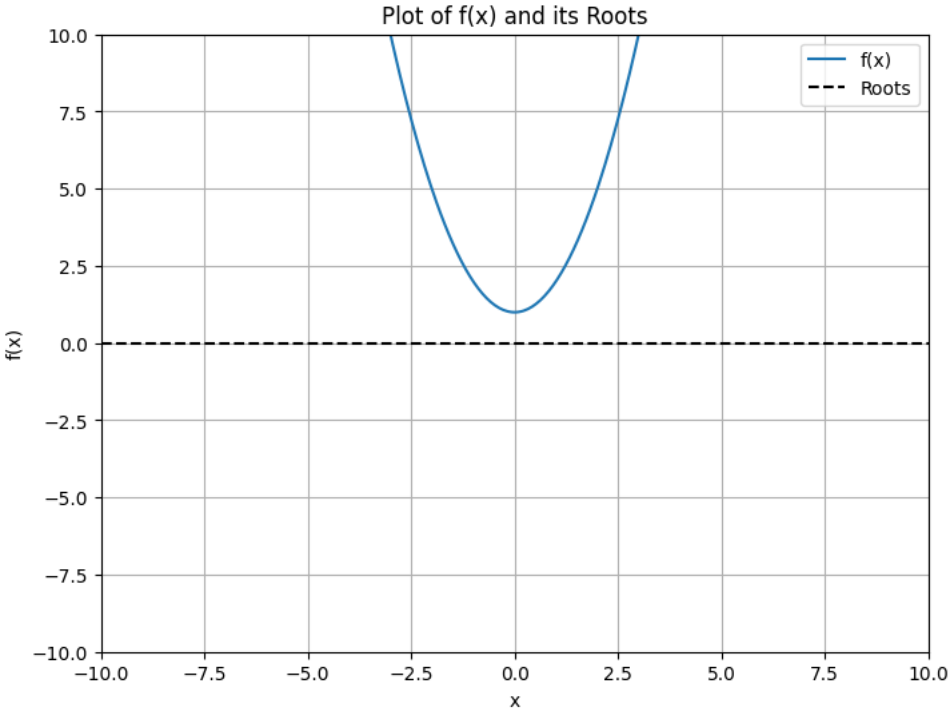



Only the roots closest to the initial guesses are found!

# Complex roots - \(x^2+1\)

Even the graphical method is not completely reliable due to the existence of *complex roots*

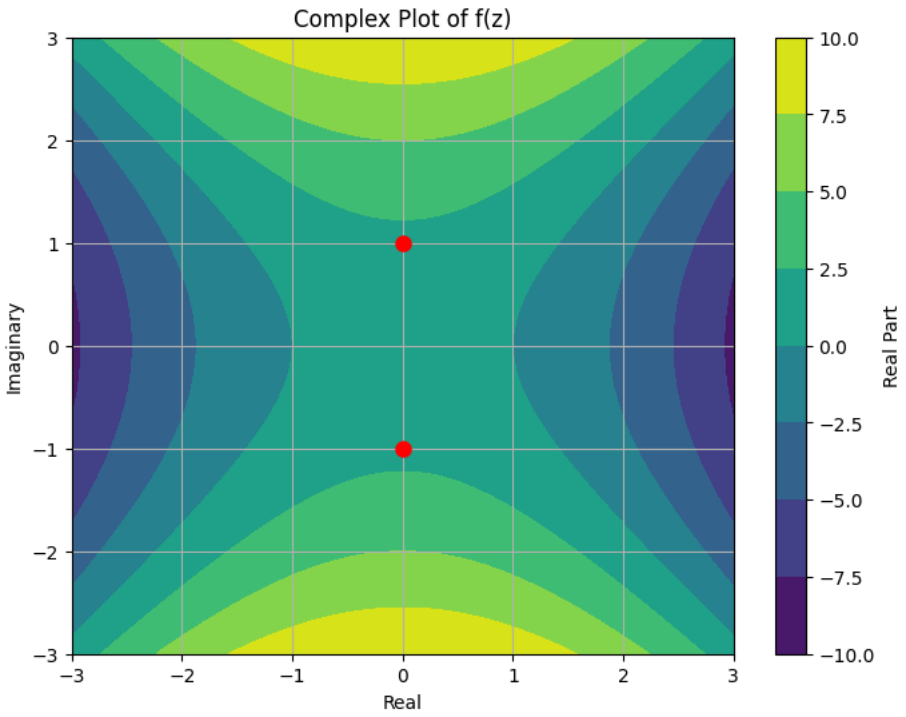
```
def fun(x):
  return x**2 + 1

# Wrong!
plot_and_find_roots(fun)
```




**But this is wrong!** The quadratic has 2 roots but we need to use a different method:

```
root(lambda x: x**2+1, x0 = [1+1j, 1-1j], method = "krylov")
```

```python
# prompt: Do a complex plot of x**2+1 and add points at the roots

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import root

def complex_plot(func):
  """Plots a complex function and finds its roots using root.

  Args:
    func: The function to plot and find roots for.
  """

  real_range = np.linspace(-3, 3, 100)
  imag_range = np.linspace(-3, 3, 100)

  real_part = np.empty((len(real_range), len(imag_range)))
  imag_part = np.empty((len(real_range), len(imag_range)))

  for i, real in enumerate(real_range):
    for j, imag in enumerate(imag_range):
      z = complex(real, imag)
      result = func(z)
      real_part[i, j] = result.real
      imag_part[i, j] = result.imag

  plt.figure(figsize=(8, 6))
  plt.contourf(real_range, imag_range, real_part, cmap='viridis')
  plt.colorbar(label='Real Part')
  plt.xlabel('Real')
  plt.ylabel('Imaginary')
  plt.title('Complex Plot of f(z)')
  plt.grid(True)

  # Find roots and plot them
  r = root(lambda x: x**2 + 1, x0=[1 + 1j, 1 - 1j], method="krylov")
  if r.success:
    for root_val in r.x:
      plt.plot(root_val.real, root_val.imag, 'ro', markersize=8)  # Plot root with a red dot

  plt.show()


# Use the function to plot x**2 + 1
complex_plot(lambda z: z**2 + 1)
```




DON'T WORRY - We won't be dealing with complex numbers in general in this course :-)

# Polynomial roots

Polynomial roots are *nice* because we can solve for them directly. Polynomials of degree $n$ have exactly $n$ roots, including multiplicity (coincident roots). The roots may be complex, even in polynomials with real coefficients. If the root is complex ($a+bi$), then its *complex conjugate* ($a-bi$) is also a root.

The roots of polynomials of order $\le 4$ can be solved analytically.

E.g.:

$a x^2 +b x + c = 0$

has roots:

$x = \frac{-b \pm \sqrt{b^2-4ac}} {2a}$

per the quadratic formula.

Certaint polynomials of order $\gt 4$ may have analytically solvable roots but there is no such general formula per the Abel-Ruffini theorem.

All the roots of polynomials may be found numerically by solving for the eigenvalues of the polynomial *companion matrix*. This is the basis for modern numerical methods which we will cover when we get to matrix eigenvalue calculations.

## Example usage of numerical root finding tools.

```
# prompt: Give me a 10th order polynomial with integer coefficients,, print the polynomial, its companio

import numpy as np
from scipy.linalg import companion

# Define the coefficients of the polynomial
coefficients = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

# Print the polynomial
print("Polynomial:")
print(np.poly1d(coefficients))

# Calculate the companion matrix
companion_matrix = companion(coefficients)

# Print the companion matrix
print("\nCompanion Matrix:")
print(np.round(companion_matrix, 2))

# Find the eigenvalues (roots) of the companion matrix
roots = np.linalg.eigvals(companion_matrix)

# Print the roots
print("\nRoots from eigenvalues of companion matrix:")
print(np.round(roots, 2))

print("\nRoots from 'roots' function:")
print(np.round(np.roots(coefficients),2))
```

```
Polynomial:
   10     9     8     7     6     5     4     3     2
1 x  + 2 x + 3 x + 4 x + 5 x + 6 x + 7 x + 8 x + 9 x + 10 x + 11

Companion Matrix:
[[ -2.  -3.  -4.  -5.  -6.  -7.  -8.  -9. -10. -11.]
 [  1.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   1.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   1.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   1.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   1.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   1.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   1.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   1.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   1.   0.]]

Roots from eigenvalues of companion matrix:
[-1.26+0.36j -1.26-0.36j -0.88+0.96j -0.88-0.96j -0.25+1.26j -0.25-1.26j
  0.44+1.17j  0.44-1.17j  0.95+0.73j  0.95-0.73j]

Roots from 'roots' function:
[-1.26+0.36j -1.26-0.36j -0.88+0.96j -0.88-0.96j -0.25+1.26j -0.25-1.26j
  0.44+1.17j  0.44-1.17j  0.95+0.73j  0.95-0.73j]
```

Note: in many software environments, $j$ is used as the imaginary number instead of $i$.

# Closed root-finding methods

The roots of nonlinear functions are more complicated than linear functions. We can seldom solve for the roots directly (notable exception being polynomial functions), and instead will need to *search* for them iteratively.

Iterative search methods are characterized by their *order of convergence*, which measure how successive guesses approach the true root. Given the true root $x$, we can check how successive guesses approach it by calculating the error:

$$ x^{i+1}-x \propto [x^{i}-x]^k $$

where $k$ is the order.

## Bracketting methods

Bracketing methods exploits the fact that functions change sign across the roots of a 1-D function (a simple applicaiton of the intermediate value theorem). This does not work for certain cases (eg: $1/x$).

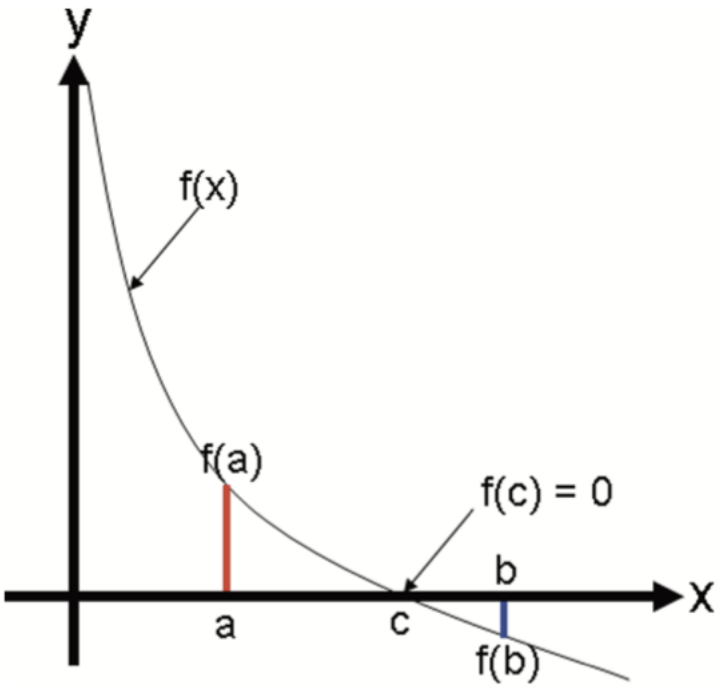
### Bisection methods

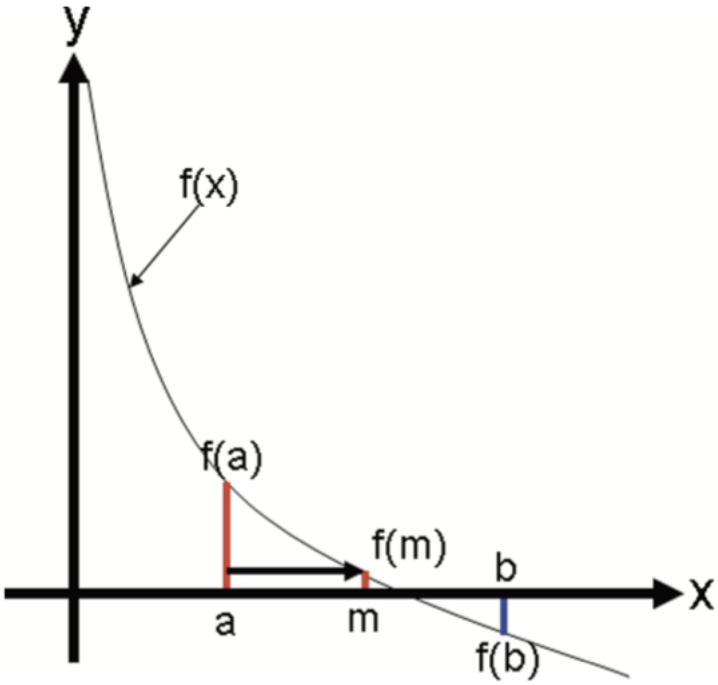
Bisection methods are essentially a binary search for the root. If $f(x)$ is continuous between bounds $a$ and $b>a$ and $f(a)$ and $f(b)$ are opposite signs, there must be a point $c$ where $f(c)=0$.

The algorithm is:

> Given brackets $a$ and $b$
>
> Calculate the midpoint $c = (b-a)/2$.
>
> If $f(c) \approx 0$ or $a \approx b$: exit.
>
> If $f(c)>0$: set $a = c$
>
> If $f(c)<0$: set $b = c$
>
> repeat

Graphically this is:




```
# prompt: Find the root of x^2-2 using bisect

from scipy.optimize import bisect

def f(x):
  return x**2 - 2

print('Setting x tolerance: \n')
xtols = [1e-1, 1e-2, 1e-3, 1e-4, 1e-5]
for xtol in xtols:
  print(bisect(f, 1, 2, maxiter = 100, xtol = xtol))


print('\nSetting relative tolerance \n', bisect(f, 1, 2, maxiter = 100, rtol = 1e-12))
```

```
Setting x tolerance:

1.4375
1.4140625
1.4150390625
1.41424560546875
1.4142074584960938

Setting relative tolerance
 1.4142135623715149
```

The error of the Bisection Method generaly follows:

$$ x^{i+1}-x = \frac{1}{2} [x^{i}-x] $$

and therefore has a linear order of convergence ($k=1$).

### Method of False Position (Regula falsi)

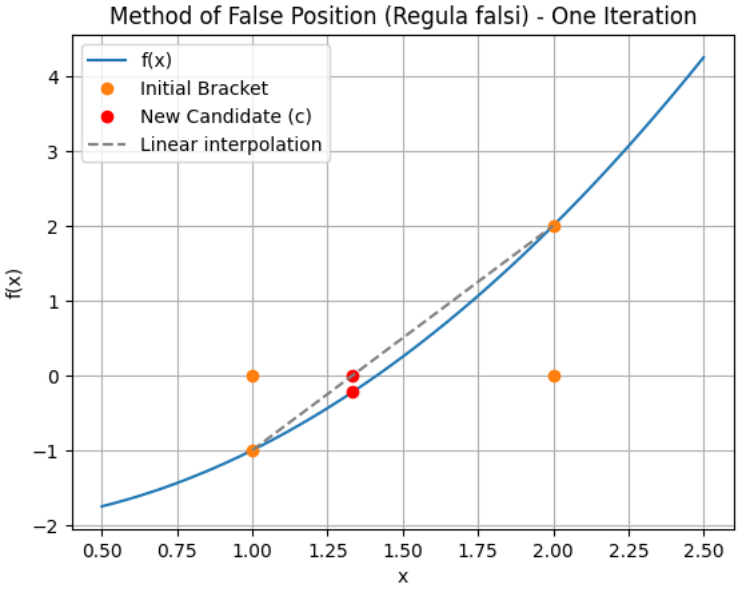
Let's use more information! In bisection we are only interested in $f(a,b)$ switching signs, but it stands to reason the larger $f(b)$ is compared to $f(a)$, the further the root is from $b$!

The method of false position uses this information to determine the next candidate solution:

$$ c= b - f(b) \frac{b-a}{f(b)-f(a)} $$

The same algorithm as for bisection is then applied to replace either $a$ or $b$ with $c$ so that the root remains bracketted.

```python
# prompt: Give me a plot that illustrates the method of false position using one iteration, showing a li

import numpy as np
import matplotlib.pyplot as plt


def f(x):
  return x**2 - 2

# Example values for a and b
a = 1
b = 2

# Calculate f(a) and f(b)
fa = f(a)
fb = f(b)

# Calculate the next candidate solution c using the method of false position
c = b - fb * (b - a) / (fb - fa)
fc = f(c)

# Generate x values for the plot
x = np.linspace(a - 0.5, b + 0.5, 100)

# Plot the function
plt.plot(x, f(x), label='f(x)')

# Plot the initial bracket
plt.plot([a, b, a, b], [fa, fb, 0, 0], 'o', label='Initial Bracket')

# Plot the new candidate solution c
plt.plot([c, c], [fc, 0], 'ro', label='New Candidate (c)')


# Plot the line connecting (a, f(a)) and (b, f(b))
plt.plot([a, b], [fa, fb], '--', color='gray', label='Linear interpolation')

# Add labels and legend
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Method of False Position (Regula falsi) - One Iteration')
plt.legend()
plt.grid(True)
plt.show()
```




Note that the line joining the brackets approximates the tangent of the curve.

The method of False Position has an order of convergence of 1.618 (the Golden Ratio!):

$$ x^{i+1}-x \propto [x^i-x]^{1.618} $$

This is considered *superlinear* and a good thing!

### Summary of bracketting methods

Bracketting methods are usually robust but slow to converge and generlization to N-D is not trivial.

Through incorporating the additional information of linear interpolation, the method of Flase Position achieves superlinear convergence.

But bracketting is complicated in N-D, so let's try to remove it.

## Open methods

Another class of root finding algorithms do not require bracketting, and are therefore deemed *open*. This alleviates the issue of N-D dimensionalization at the expense of robustness. They are usually faster than bracketting methods since we are not constantly updating the brackets, but our method is now succeptible to divergence.

### Secant method

Let's reconsider the method of False Position but now we always disregard bracket updating and simply take $c$ as our new guess. Our algorithm is now:

> Take 2 initial guesses: $x^i$
>
> Calculate the next guess:

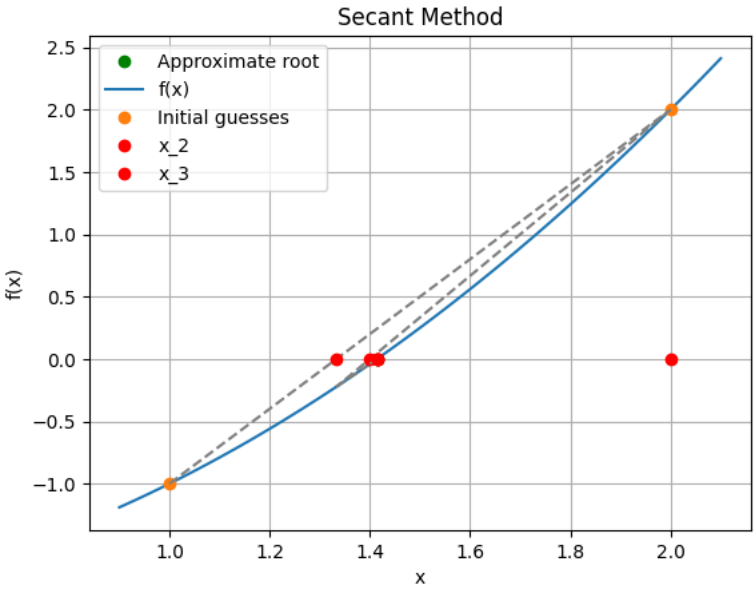
$$x^{i+1} = {x^i} - f(x^{i}) \frac{x^i-x^{i-1}}{f(x^i)-f(x^{i-1})}$$

> Check if tolerance is met

(Which tolerance?)

```python
# prompt: Solve x^2-2 using the secant method, plotting each step with lines and each guess with points (

import numpy as np
import matplotlib.pyplot as plt

def f(x):
  return x**2 - 2

def secant_method(f, x0, x1, tolerance=1e-6, max_iterations=100):
  """
  Finds the root of a function using the secant method.

  Args:
    f: The function to find the root of.
    x0: The initial guess.
    x1: The second initial guess.
    tolerance: The desired tolerance for the root.
    max_iterations: The maximum number of iterations.

  Returns:
    The approximate root of the function.
  """
  print(f"Iteration 0: x = {x1}")
  x_values = [x0, x1]
  for i in range(max_iterations):
    x_new = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
    x_values.append(x_new)
    print(f"Iteration {i+1}: x = {x_new}")
    if abs(f(x_new)) < tolerance:
      return x_new, x_values
    x0 = x1
    x1 = x_new
  return None, x_values

# Initial guesses
x0 = 1
x1 = 2

# Find the root using the secant method
root, x_values = secant_method(f, x0, x1)


if root:
    print("Approximate root:", root)
    plt.plot(root, 0, 'go', label='Approximate root')
else:
    print("Secant method did not converge within the maximum number of iterations.")

error = abs(np.array(x_values)-root)
print("\nThe sequence of errors is:")
for i,e in enumerate(error):
  print('Iteration, ', i, ' error in x is ', e)


# Plot the function and the secant method iterations
x = np.linspace(.9, 2.1, 100)
plt.plot(x, f(x), label='f(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Secant Method')
plt.grid(True)

# Plot the initial guesses
plt.plot([x0, x1], [f(x0), f(x1)], 'o', label='Initial guesses')

# Plot each iteration of the secant method
for i in range(len(x_values) - 1):
    plt.plot([x_values[i], x_values[i+1]], [f(x_values[i]), f(x_values[i+1])], '--', color='gray')
    plt.plot(x_values[i+1], 0, 'ro', label=f'x_{i+2}' if i < 2 else None)


plt.legend()
plt.show()
```

```
Iteration 0: x = 2
Iteration 1: x = 1.3333333333333335
Iteration 2: x = 1.4000000000000001
Iteration 3: x = 1.4146341463414633
Iteration 4: x = 1.41421143847487
Iteration 5: x = 1.4142135620573204
Approximate root: 1.4142135620573204

The sequence of errors is:
Iteration,  0  error in x is  0.4142135620573204
Iteration,  1  error in x is  0.5857864379426796
Iteration,  2  error in x is  0.08088022872398692
Iteration,  3  error in x is  0.014213562057320273
Iteration,  4  error in x is  0.00042058428414293303
Iteration,  5  error in x is  2.12358245033073e-06
Iteration,  6  error in x is  0.0
```




The Secant method maintains superlinear convergence … but we can do better with a little more information…

## The Newton-Raphson method

Take another look at the fraction in the Secant method update equation:

$$x^{i+1} = {x^i} - f(x^{i}) \frac{x^i-x^{i-1}}{f(x^i)-f(x^{i-1})}$$

This is an (inverse) approximation of $\frac{\partial f}{\partial x}$, the derivative of $f$! Typically, we are able to find this quantity, and the algorithm becomes:

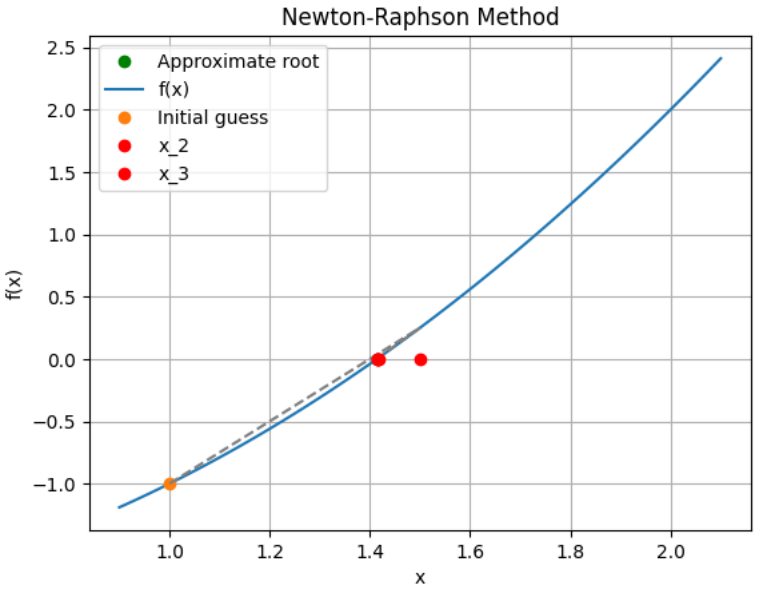
$$ x^{i+1} = {x^i} - \frac{f(x^{i})}{f'(x^i)} $$

or, solving for the increment $\Delta x = x^{i+1}-x^i$ and dropping the indicies,

$$\begin{split} \begin{align} \Delta x &= - \frac{f(x)}{f'(x)} \\ f'(x) \Delta x &= - f(x) \end{align} \end{split}$$

```python
# prompt: Repeat the root finding using Newton's method and scipy tools

import numpy as np
import matplotlib.pyplot as plt

def f(x):
  return x**2 - 2

def df(x):
  return 2*x

def newton_raphson(f, df, x0, tolerance=1e-6, max_iterations=100):
  """
  Finds the root of a function using the Newton-Raphson method.

  Args:
    f: The function to find the root of.
    df: The derivative of the function.
    x0: The initial guess.
    tolerance: The desired tolerance for the root.
    max_iterations: The maximum number of iterations.

  Returns:
    The approximate root of the function.
  """

  x_values = [x0]
  print(f"Iteration 0: x = {x0}")
  for i in range(max_iterations):
    x_new = x0 - f(x0) / df(x0)
    x_values.append(x_new)
    print(f"Iteration {i+1}: x = {x_new}")
    if abs(f(x_new)) < tolerance:
      return x_new, x_values
    x0 = x_new
  return None, x_values

# Initial guess
x0 = 1

# Find the root using the Newton-Raphson method
root, x_values = newton_raphson(f, df, x0)

if root:
    print("Approximate root:", root)
    plt.plot(root, 0, 'go', label='Approximate root')
else:
    print("Newton-Raphson method did not converge within the maximum number of iterations.")

error = abs(np.array(x_values)-root)
print("\nThe sequence of errors is:")
for i,e in enumerate(error):
  print('Iteration, ', i, ' error in x is ', e)


# Plot the function and the Newton-Raphson method iterations
x = np.linspace(.9, 2.1, 100)
plt.plot(x, f(x), label='f(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Newton-Raphson Method')
plt.grid(True)

# Plot the initial guess
plt.plot([x0], [f(x0)], 'o', label='Initial guess')

# Plot each iteration of the Newton-Raphson method
for i in range(len(x_values) - 1):
    plt.plot([x_values[i], x_values[i+1]], [f(x_values[i]), f(x_values[i+1])], '--', color='gray')
    plt.plot(x_values[i+1], 0, 'ro', label=f'x_{i+2}' if i < 2 else None)


plt.legend()
plt.show()
```

```
Iteration 0: x = 1
Iteration 1: x = 1.5
Iteration 2: x = 1.4166666666666667
Iteration 3: x = 1.4142156862745099
Iteration 4: x = 1.4142135623746899
Approximate root: 1.4142135623746899

The sequence of errors is:
Iteration,  0  error in x is  0.41421356237468987
Iteration,  1  error in x is  0.08578643762531013
Iteration,  2  error in x is  0.002453104291976871
Iteration,  3  error in x is  2.123899820016817e-06
Iteration,  4  error in x is  0.0
```

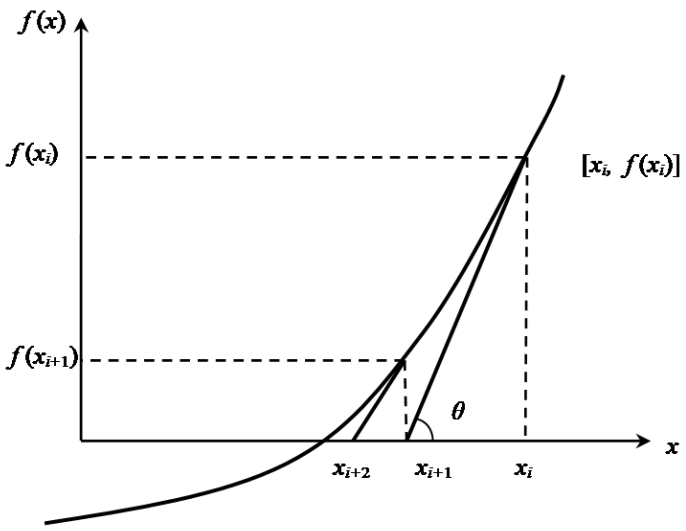



The Newton-Raphson method has quadratic convergence (\(k=2\)) *near the root* which is a great result! It does so, however at the cost of calculating the Jacobian and solving a linear system.

As we saw in the previous lecture, Newton's method amounts to usign the current position and the (true) tangent to estimate the next guess:

\[ f'(x) \Delta x = - f(x)\]

Graphically:



Near the root, it converges quadratically, but there are some not-uncommon scenarios where it can fail:

The Newton-Raphson method only finds *local* roots, not all of them. Efficient and robust root finding **requires a good initial guess**.

Fortunatley, in Engineering, this is commonly the case!

## Example of an initial guess

If we need to solve for temperature $T(x,y,z,t)$ as a nonlinear, time dependent, partial differential equation, we will be given an initial value for $T(x,y,z, t=0)$.

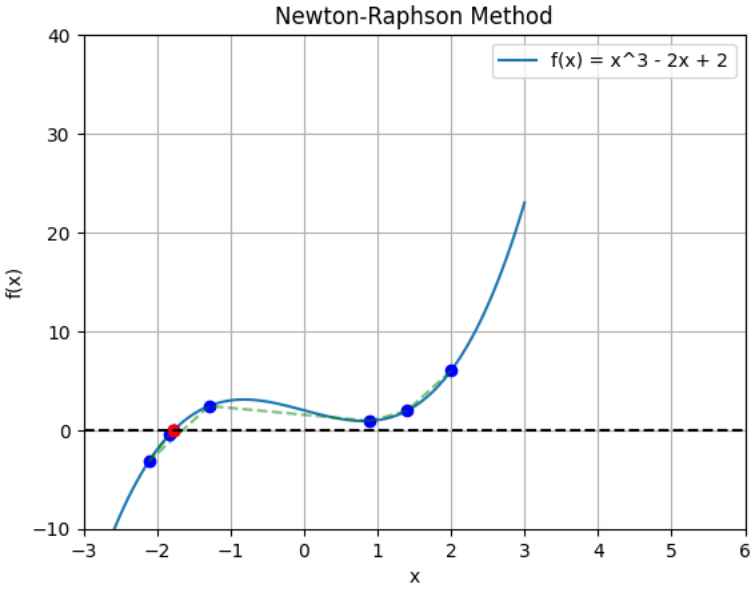
When solving a nonlinear equation for $T(x,y,z,t=1)$, what do you suppose the initial guess should be?

Answer: The initial guess for should be the solution at the preceeding time step!

## Example: find the root of $x^3-2x+2$

There is a real root at x~-1.769.

```python
# prompt: Use a newton-raphson method to find the root of x^3-2x+2 form an initial guess of 5 and max it

import numpy as np
import matplotlib.pyplot as plt

def f(x):
  """The function whose root we want to find."""
  return x**3 - 2*x + 2

def df(x):
  """The derivative of the function."""
  return 3*x**2 - 2

def newton_raphson(f, df, x0, max_iter=8, tolerance=1e-6):
  """
  Finds a root of the function f using the Newton-Raphson method.

  Args:
    f: The function whose root we want to find.
    df: The derivative of the function.
    x0: The initial guess for the root.
    max_iter: The maximum number of iterations.
    tolerance: The tolerance for the root.

  Returns:
    The root of the function, or None if the method fails to converge.
  """
  x = x0
  guesses = [x]
  for i in range(max_iter):
    x_new = x - f(x) / df(x)
    guesses.append(x_new)

    if abs(x_new - x) < tolerance:
      return x_new, guesses

    x = x_new
  return None, guesses

x0 = 2
root, guesses = newton_raphson(f, df, x0)


# Plot the function and the guesses
x_vals = np.linspace(-3, 3, 100)
y_vals = f(x_vals)
plt.plot(x_vals, y_vals, label="f(x) = x^3 - 2x + 2")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Newton-Raphson Method")

for i, guess in enumerate(guesses):
  plt.plot(guess, f(guess), 'ro' if i == len(guesses) - 1 else 'bo')
  if i > 0:
    plt.plot([guesses[i-1], guess], [f(guesses[i-1]), f(guess)], 'g--', alpha=0.5)


plt.axhline(0, color='black', linestyle='--')  # Add horizontal line at y=0
plt.legend()
plt.xlim([-3, 6]) # Set x-axis limits
plt.ylim([-10, 40]) # Set y-axis limits
plt.grid(True)
plt.show()

if root:
  print(f"Root found: {root:.6f}")
else:
  print("Newton-Raphson method failed to converge.")
```




```
Newton-Raphson method failed to converge.
```

### Example: Find the root of \(\sqrt(x)\)

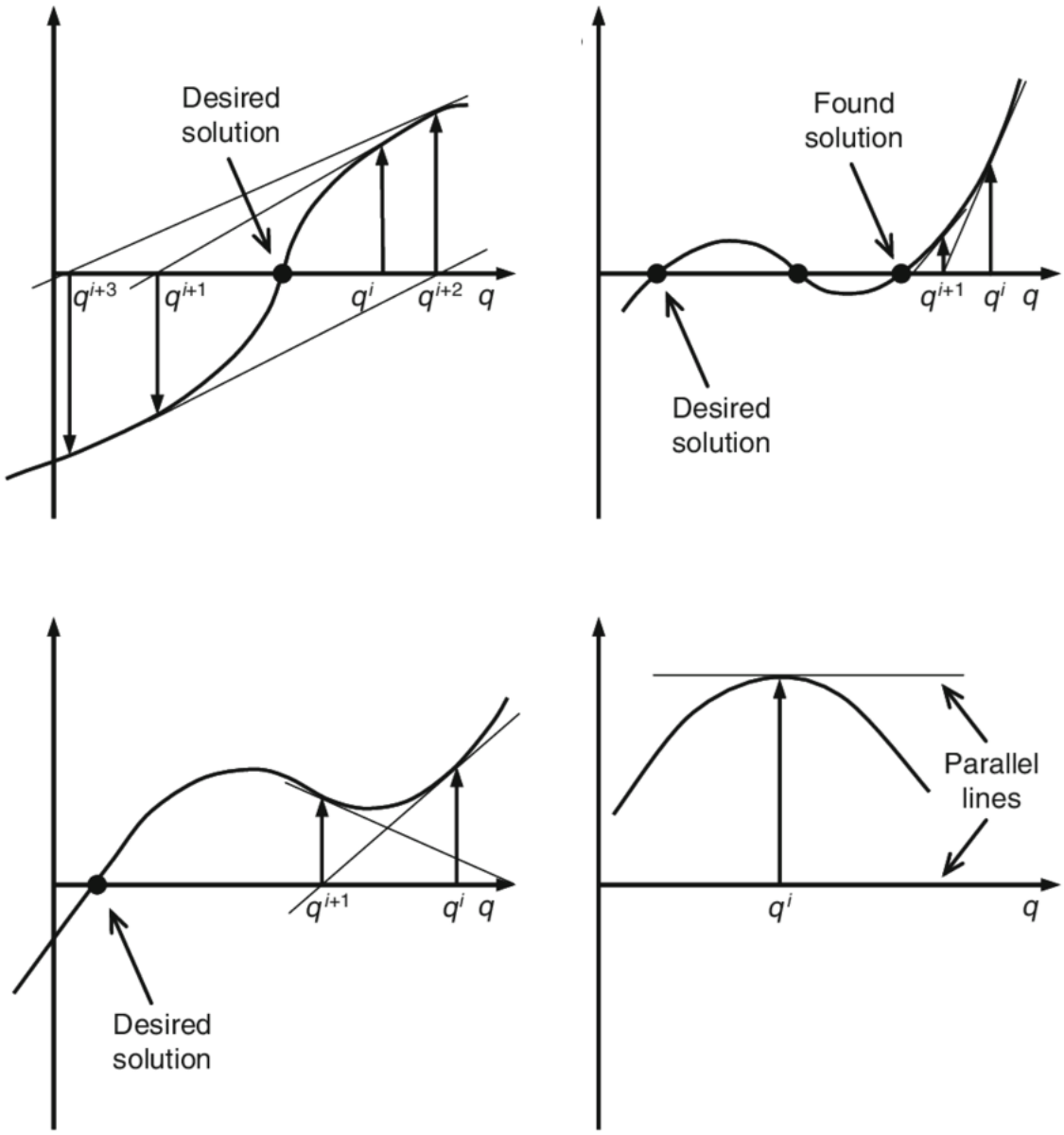
```python
import numpy as np
def f(x):
  return np.sqrt(x)

def jacobian(x):
  return 1/3*x**(-2./3)

def newton_raphson(x0, tolerance=1e-6, max_iterations=10):
  """
  Newton-Raphson method for solving a system of nonlinear equations.
  """
  x = x0
  for iter in range(max_iterations):
    f_x = f(x)
    print("Iteration, ", iter, " the guess is ", np.round(x,3), " with residual ", np.linalg.norm(f_x) )
    J_x = jacobian(x)
    delta_x = -f_x/J_x
    x = x + delta_x
    if np.linalg.norm(f_x) < tolerance:
      return x
  return None  # No solution found within the maximum iterations

newton_raphson(x0 = 2)
#newton_raphson(x0 = 2+0j)
```

```
Iteration,  0  the guess is  2  with residual  1.4142135623730951
Iteration,  1  the guess is  -4.735  with residual  nan
Iteration,  2  the guess is  nan  with residual  nan
Iteration,  3  the guess is  nan  with residual  nan
Iteration,  4  the guess is  nan  with residual  nan
Iteration,  5  the guess is  nan  with residual  nan
Iteration,  6  the guess is  nan  with residual  nan
Iteration,  7  the guess is  nan  with residual  nan
Iteration,  8  the guess is  nan  with residual  nan
Iteration,  9  the guess is  nan  with residual  nan
```

```
<ipython-input-17-26a7d35b8674>:3: RuntimeWarning: invalid value encountered in sqrt
  return np.sqrt(x)
<ipython-input-17-26a7d35b8674>:6: RuntimeWarning: invalid value encountered in scalar power
  return 1/3*x**(-2./3)
```

What went wrong here? What else could we try?

### Some common failure situations




Before we talk about mitigation strategies, let's generalize the Newton-Raphson method to N-D

## The N-D Newton-Raphson method

The Newton-Raphson method thus far has been described for scalar functions or scalar arguments (i.e.: 1-D).

Consider a system of $n$ unkowns $\vec{x}$ and a set of $n$ nonlinear equations that we wish to solve simultaneously:

$$\begin{split}\begin{align} f_1(\vec{x}) &= 0 \\ f_2(\vec{x}) &= 0 \\ \vdots \\ f_n(\vec{x}) &= 0 \end{align} \end{split}$$

which may be summarized as a vector function $\vec{f}(\vec{x})=\vec{0}$ also of dimension $n$. Since we have $n$ equations and $n$ unkowns, we can (hopefully) find an exact solution of the simultaneous set of equations, i.e.: a root.

The Newton-Raphson method generalized quite readily except the derivative must be replaced by the vector-derivative of a vector function (called the *Jacobian*):

$$\begin{split} J = \frac{\partial \vec{f}}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \end{split}$$
where we can see that $J$ is a square $n \times n$ matrix. The Newton-Raphson method takes the form: $ $ J \Delta \vec{x} = - \vec{f} $ which is .... (wait for it!)... a linear system solving for the vector of increments, $\Delta\vec{x}$!

This is an example of computational-thinking: we have broken down a muiltivariable non-linear vector function into a sequence of linear systems!

## Example: solve a system of nonlinear equations:

$$\begin{split} \begin{align} x^2 + y^2 - z &= 1 \\ x - y^2 + z^2 &= 1 \\ x y z &= 1 \end{align} \end{split}$$

#####Answer

Rewrite the equations as a system of nonlinear functions:

$$\begin{split}\begin{align} f_1(x, y, z) &= x^2 + y^2 - z - 1 \\ f_2(x, y, z) &= x - y^2 + z^2 -1 \\ f_3(x, y, z) &= x*y*z - 1 \end{align} \end{split}$$
or in vector form:

$$\begin{split}\vec{f}(\vec{x}) = \begin{bmatrix} f_1(x, y, z) \\ f_2(x, y, z) \\ f_3(x, y, z) \end{bmatrix} = \begin{bmatrix} x^2 + y^2 - z - 1 \\ x - y^2 + z^2 -1 \\ x y z - 1 \end{bmatrix} =\vec{0} \end{split}$$
The Jacobian is:

$$\begin{split} J = \frac{\partial \vec{f}}{\partial \vec{x}} = \begin{bmatrix} 2x & 2y & -1 \\ 1 & -2y & 2z \\ yz & xz & xy \end{bmatrix} \end{split}$$
Now for a given $\vec{x}$ we can solve for the increment.

```
# prompt: show newton's method to solve this system, using linalg.solve

import numpy as np

def f(x):
  """
  The system of nonlinear equations.
  """
  x, y, z = x
  return np.array([
      x**2 + y**2 - z - 1,
      x - y**2 + z**2 - 1,
      x * y * z - 1
  ])

def jacobian(x):
  """
  The Jacobian matrix.
  """
  x, y, z = x
  return np.array([
      [2 * x, 2 * y, -1],
      [1, -2 * y, 2 * z],
      [y * z, x * z, x * y]
  ])

def newton_raphson(x0, tolerance=1e-6, max_iterations=100):
  """
  Newton-Raphson method for solving a system of nonlinear equations.
  """
  x = x0
  for iter in range(max_iterations):
    f_x = f(x)
    print("Iteration, ", iter, " the guess is ", np.round(x,3), " with residual ", np.linalg.norm(f_x) )
    J_x = jacobian(x)

    #~~~~ What now? ####


    #~~ Answer
    #delta_x = np.linalg.solve(J_x, -f_x)
    #~~~~
    x = x + delta_x
    if np.linalg.norm(f_x) < tolerance:
      return x
  return None  # No solution found within the maximum iterations

# Initial guess
x0 = np.array([2, 2, 2])

# Solve the system
solution = newton_raphson(x0)

if solution is not None:
  print("Solution found:", solution)
else:
  print("No solution found within the maximum iterations.")
```

```
Iteration,  0  the guess is  [2 2 2]  with residual  8.660254037844387
Iteration,  1  the guess is  [1.04 1.61 1.6 ]  with residual  1.9930051233250754
Iteration,  2  the guess is  [1.07  1.091 1.066]  with residual  0.36503151275567536
Iteration,  3  the guess is  [1.001 1.008 1.007]  with residual  0.019821923184321227
Iteration,  4  the guess is  [1. 1. 1.]  with residual  9.992137180440437e-05
Iteration,  5  the guess is  [1. 1. 1.]  with residual  2.103116571058853e-09
Solution found: [1. 1. 1.]
```

Let's try a different initial guess:

```
solution = newton_raphson(np.array([3, 3 , 3]))
```

```
Iteration,  0  the guess is  [3 3 3]  with residual  29.597297173897484
Iteration,  1  the guess is  [1.054 2.533 2.524]  with residual  6.998575456719698
Iteration,  2  the guess is  [0.889 1.641 1.66 ]  with residual  1.6424438231583698
Iteration,  3  the guess is  [ 2.047  0.239 -0.06 ]  with residual  3.603276469109183
Iteration,  4  the guess is  [1.707 1.028 2.231]  with residual  5.517969890171211
Iteration,  5  the guess is  [1.238 0.984 1.279]  with residual  1.0856646419132276
Iteration,  6  the guess is  [1.031 0.998 1.016]  with residual  0.0930478495194166
Iteration,  7  the guess is  [1. 1. 1.]  with residual  0.0010714827370243664
Iteration,  8  the guess is  [1. 1. 1.]  with residual  1.8136147365183113e-07
```

Great, but something is funny with the residual… Let's keep going!

```
solution = newton_raphson(np.array([10, 10 , 10]))
```

```
Iteration,  0  the guess is  [10 10 10]  with residual  1016.7610338717747
Iteration,  1  the guess is  [1.037 9.488 9.486]  with residual  122.54179913742261
Iteration,  2  the guess is  [0.99  5.009 5.009]  with residual  31.151332424357115
Iteration,  3  the guess is  [0.91  2.803 2.812]  with residual  7.863318171708681
Iteration,  4  the guess is  [0.756 1.814 1.861]  with residual  1.8475446018143282
Iteration,  5  the guess is  [0.273 1.768 1.965]  with residual  0.24020703623206316
Iteration,  6  the guess is  [0.319 1.664 1.858]  with residual  0.01924168764896064
Iteration,  7  the guess is  [0.327 1.656 1.848]  with residual  0.00026048720664438785
Iteration,  8  the guess is  [0.327 1.656 1.848]  with residual  4.90450098732614e-08
```

Still converged but to a different root…

What about negatives?

```
solution = newton_raphson(np.array([-1, -1 ,-1]))
```

```
Iteration,  0  the guess is  [-1 -1 -1]  with residual  3.4641016151377544
Iteration,  1  the guess is  [-7.  5.  1.]  with residual  86.62563131083085
Iteration,  2  the guess is  [-3.924  2.106  0.99 ]  with residual  21.7413789453847
Iteration,  3  the guess is  [-2.54   0.452  1.005]  with residual  5.807927456281761
Iteration,  4  the guess is  [-1.921 -0.552  1.606]  with residual  1.686308408306664
Iteration,  5  the guess is  [-1.619 -0.393  1.659]  with residual  0.13083381603982347
Iteration,  6  the guess is  [-1.583 -0.383  1.652]  with residual  0.0015836394832936728
Iteration,  7  the guess is  [-1.583 -0.382  1.652]  with residual  2.8665073240160246e-07
```

Another root?

```
solution = newton_raphson(np.array([-10, 0,-10]))
```

```
Iteration,  0  the guess is  [-10   0 -10]  with residual  140.72313242676202
Iteration,  1  the guess is  [-4.786  0.01  -5.289]  with residual  35.1045008430222
Iteration,  2  the guess is  [-2.189  0.049 -2.946]  with residual  8.720052763994339
Iteration,  3  the guess is  [-0.908  0.203 -1.8  ]  with residual  2.2103101495737234
Iteration,  4  the guess is  [-0.125  0.844 -1.296]  with residual  1.3486694778260442
Iteration,  5  the guess is  [-1.024  0.136 -1.243]  with residual  1.6270099713785144
Iteration,  6  the guess is  [-0.291  0.885 -1.23 ]  with residual  1.408993850920174
Iteration,  7  the guess is  [-1.226 -0.019 -1.188]  with residual  2.1415268760927777
Iteration,  8  the guess is  [-0.531  0.676 -1.227]  with residual  1.216428132602447
Iteration,  9  the guess is  [-3.688 -2.373 -1.03 ]  with residual  23.600319365841493
Iteration,  10  the guess is  [-2.12  -0.752 -1.027]  with residual  6.304563296941992
Iteration,  11  the guess is  [-1.242  0.241 -1.154]  with residual  2.109349117256319
Iteration,  12  the guess is  [-0.356  0.836 -1.314]  with residual  1.3331778596549482
Iteration,  13  the guess is  [-1.391 -0.202 -1.172]  with residual  2.7375761952550013
Iteration,  14  the guess is  [-0.711  0.521 -1.209]  with residual  1.2436300157477957
Iteration,  15  the guess is  [ 1.97   3.045 -1.403]  with residual  17.682903586717295
Iteration,  16  the guess is  [ 0.43   1.799 -1.505]  with residual  4.740653742813579
Iteration,  17  the guess is  [-0.056  0.927 -1.135]  with residual  1.5069123725644382
Iteration,  18  the guess is  [-0.984  0.258 -1.274]  with residual  1.5340770119968596
Iteration,  19  the guess is  [-0.125  1.026 -1.26 ]  with residual  1.6777911160799737
Iteration,  20  the guess is  [-0.865  0.314 -1.208]  with residual  1.3487522282381879
Iteration,  21  the guess is  [ 0.135  1.306 -1.261]  with residual  2.5289659936255964
Iteration,  22  the guess is  [-0.533  0.632 -1.217]  with residual  1.1676163605960792
Iteration,  23  the guess is  [-5.092 -3.751 -0.999]  with residual  48.68184454962859
Iteration,  24  the guess is  [-2.826 -1.497 -0.992]  with residual  12.538686428669257
Iteration,  25  the guess is  [-1.703 -0.194 -1.021]  with residual  3.663497400946906
Iteration,  26  the guess is  [-0.837  0.528 -1.291]  with residual  1.413679669121953
Iteration,  27  the guess is  [ 0.892  1.976 -1.387]  with residual  6.488587658069775
Iteration,  28  the guess is  [ 0.262  1.056 -1.057]  with residual  1.9381372081194084
Iteration,  29  the guess is  [-0.871  0.564 -1.45 ]  with residual  1.5561820481695032
Iteration,  30  the guess is  [ 0.873  1.937 -1.413]  with residual  6.26802202959098
Iteration,  31  the guess is  [ 0.264  1.032 -1.055]  with residual  1.8833724744840956
Iteration,  32  the guess is  [-0.899  0.554 -1.465]  with residual  1.6043219785292986
Iteration,  33  the guess is  [ 0.689  1.762 -1.401]  with residual  5.025175697742625
Iteration,  34  the guess is  [ 0.033  0.961 -1.146]  with residual  1.598476653548893
Iteration,  35  the guess is  [-0.891  0.358 -1.295]  with residual  1.3938360075888543
Iteration,  36  the guess is  [ 0.159  1.293 -1.28 ]  with residual  2.503884598918348
Iteration,  37  the guess is  [-0.516  0.635 -1.22 ]  with residual  1.155646474549891
Iteration,  38  the guess is  [-4.362 -3.057 -1.052]  with residual  34.90941151659323
Iteration,  39  the guess is  [-2.445 -1.144 -1.045]  with residual  9.087482657747831
Iteration,  40  the guess is  [-1.464 -0.016 -1.093]  with residual  2.768029026950773
Iteration,  41  the guess is  [-0.64   0.619 -1.288]  with residual  1.2416888656491218
Iteration,  42  the guess is  [32.342 31.94  -3.679]  with residual  4436.189156259958
Iteration,  43  the guess is  [15.693 16.412 -3.681]  with residual  1107.961904636607
Iteration,  44  the guess is  [ 7.126  8.812 -3.71 ]  with residual  274.39194974597626
Iteration,  45  the guess is  [ 2.142  5.383 -4.023]  with residual  61.00142372146434
Iteration,  46  the guess is  [ 3.103  1.911 -0.705]  with residual  14.016583257210655
Iteration,  47  the guess is  [ 1.524  1.138 -0.475]  with residual  3.629989084885241
Iteration,  48  the guess is  [1.224 0.481 0.209]  with residual  1.0206120878632898
Iteration,  49  the guess is  [1.268 1.093 1.425]  with residual  1.5189063185361946
Iteration,  50  the guess is  [1.035 1.027 1.069]  with residual  0.19243332948279715
Iteration,  51  the guess is  [1.001 1.001 1.003]  with residual  0.006473943066547444
Iteration,  52  the guess is  [1. 1. 1.]  with residual  9.601721087105206e-06
Iteration,  53  the guess is  [1. 1. 1.]  with residual  2.1293781082294768e-11
```

Yikes! Note what's happening to guesses and the residual… odd behaviour indeed!

## Basins of attraction

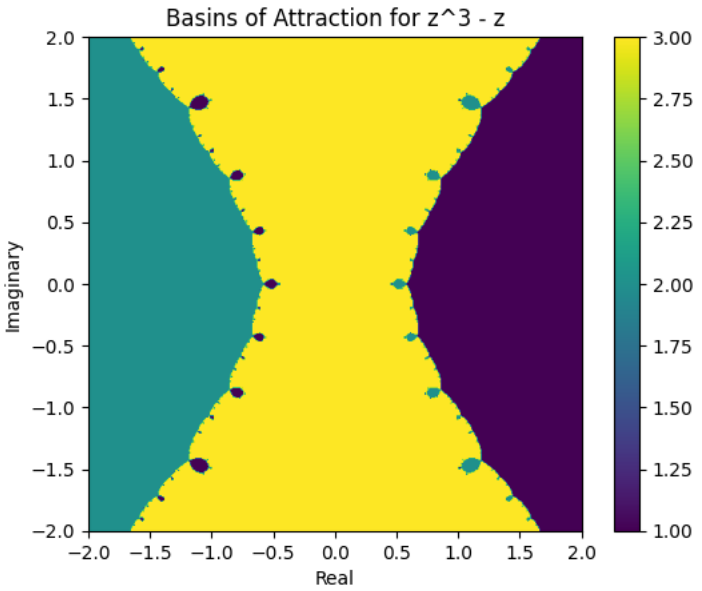
The previous examples show that depending on the initial guess, we may arrive at different roots! These are the so-called *basins of attraction* for each root. Newton's method *jumps* around the parameter space, and the increase in residual corresponds to a jump across a basin. This can lead to tempermental (but beautiful) behaviour.

### Tempermental but beautiful behaviour

The Newton-Raphson method naturally can handle complex numbers of the form \(x + y i\).

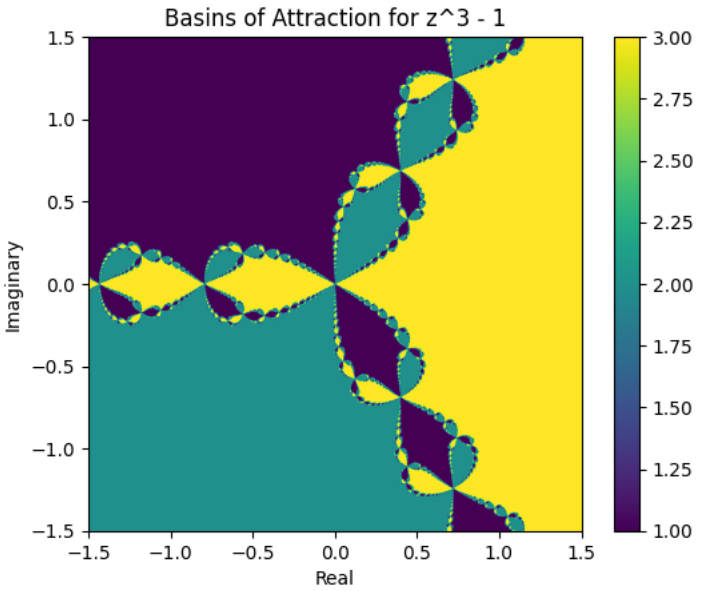
**Example: \(x^3-x\)**

```
# prompt: Show the basins of attraction for x**3-x in the complex plane

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import newton

def f(z):
  return z**3 - z

def df(z):
  return 3 * z**2 - 1

# Create a grid of complex numbers
real_range = (-2, 2)
imag_range = (-2, 2)
grid_size = 500
real_values = np.linspace(real_range[0], real_range[1], grid_size)
imag_values = np.linspace(imag_range[0], imag_range[1], grid_size)
z_grid = np.array([[complex(r, i) for r in real_values] for i in imag_values])

# Apply Newton-Raphson to each point in the grid
#roots = np.array([[newton(f, z, fprime=df) for z in row] for row in z_grid])
roots = newton(f, z_grid, fprime=df)

# Assign colors based on the root found
colors = np.zeros((grid_size, grid_size))
for i in range(grid_size):
  for j in range(grid_size):
    if roots[i, j] is None:
      colors[i, j] = 0
    elif abs(roots[i, j] - 1) < 0.5:
      colors[i, j] = 1
    elif abs(roots[i, j] - (-1)) < 0.5:
      colors[i, j] = 2
    elif abs(roots[i, j] - 0) < 0.5:
      colors[i, j] = 3
    else:
      colors[i, j] = 0  # Assign a default color


# Plot the basins of attraction
plt.imshow(colors, extent=[real_range[0], real_range[1], imag_range[0], imag_range[1]], origin='lower', (
plt.xlabel('Real')
plt.ylabel('Imaginary')
plt.title('Basins of Attraction for z^3 - z')
plt.colorbar()
plt.show()
```




**Example: \(x^3-1\)**

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import newton

def f(z):
  return z**3 - 1

def df(z):
  return 3 * z**2

# Create a grid of complex numbers
real_range = (-1.5, 1.5)
imag_range = (-1.5, 1.5)
grid_size = 1000
real_values = np.linspace(real_range[0], real_range[1], grid_size)
imag_values = np.linspace(imag_range[0], imag_range[1], grid_size)
z_grid = np.array([[complex(r, i) for r in real_values] for i in imag_values])

# Apply Newton-Raphson to each point in the grid
#roots = np.array([[newton(f, z, fprime=df) for z in row] for row in z_grid])
roots = newton(f, z_grid, fprime=df)

# Assign colors based on the root found
th = 1e-3
rs = np.roots([1,0,0,-1])
colors = np.zeros((grid_size, grid_size))
for i in range(grid_size):
  for j in range(grid_size):
    if roots[i, j] is None:
      colors[i, j] = 0
    elif abs(roots[i, j] - rs[0]) < th:
      colors[i, j] = 1
    elif abs(roots[i, j] - rs[1]) < th:
      colors[i, j] = 2
    elif abs(roots[i, j] - rs[2]) < th:
      colors[i, j] = 3
    else:
      colors[i, j] = 0  # Assign a default color


# Plot the basins of attraction
plt.imshow(colors, extent=[real_range[0], real_range[1], imag_range[0], imag_range[1]], origin='lower', (
plt.xlabel('Real')
plt.ylabel('Imaginary')
plt.title('Basins of Attraction for z^3 - 1')
plt.colorbar()
plt.show()
```




Beautiful, but like most beautiful things… often problematic… It implies small changes in initial guesses can find dramatically different roots, and numerical methods are prone to ‘small changes’ due to roundoff error…

Open in Colab

Goals:

- See the pitfalls of Newton's method
- Understand the ND Newton-Raphson method
- Awareness of linesearch algorithms for improved convergence.

# 'Global' convergence

There are several options to modify the Newton-Raphson method in order to enhance the robustness of root finding, but the improvement in robustness has to be weighed against the computational expense.

We **have** to assume our initial guess is reasonable, so the goal is to ensure the solution doesn't *wander*.

## Line search

*If* we trust that $\Delta \vec{x}$ is pointing in the right direction, then we should make sure it doesn't *step too far*.

These approaches are called *line search* alrogithms since we are moving along the direction prescribed by $\Delta \vec{x}$.

### Damped Newton-Raphson

The easiest modification is adding a damping term. Calculate $\Delta \vec{x}$ as usual from $J \Delta \vec{x} = - \vec{f}$ but update the step a scalar factor of the increment: $\( \vec{x}^{i+1} = \vec{x}^i+\alpha \Delta \vec{x} \)$

For $\alpha =1$ we recover the method.

For $0 \lt \alpha \lt 1$ the method is underdamped, and the solver forced to take small steps, keeping it closer to the guess but at the cost of convergence speed.

For $\alpha \gt 1$ the method is overdamped, and solver steps exagerated. This may seem like a good idea to speed things up but if you are not careful you will constantly overshoot your root or produce frenetic behaviour!

### Optimal step size

From the *damped* approach, we can also attempt to find a 'good' step size algorithmically. Near the root, we know we want $\alpha=1$ to get quadratic convergence, so this is often a starting point.

There are several approaches which levarage the information we have:

- $\vec{f}(\vec{x})$
- $J(\vec{x})$
- $\vec{f}(\vec{x}+\Delta \vec{x})$

### Fit a quadratic

With 3 pieces of information, we can fit a quadratic in the search direction and choose the minimum for the update.

### Backtracking

The backtracking routine starts with $\alpha=1$ and then subdivides it until

$$\vec{f}(\vec{x}+\alpha \Delta \vec{x}) - \vec{f}(\vec{x}) \approx J(\vec{x}) \alpha \Delta \vec{x}$$

### Bounded minimization

Find $\alpha$ in the range (0,1] that minimizes some measure of residual, e.g. $|\vec{f}(\vec{x}+\alpha \Delta \vec{x})|$.

This is a 1D minimizaqtion problem (similar to our 1D root finding) and can use similar tools e.g.: Secant method. The tradeoff here is the efficiency of this minimzation vs just taking another Newton step.

# Optimization

Optimzation asks us to:

Find $x$ that minimizes $f(\vec{x})$ subject to $\vec{g}(\vec{x}) = 0$ and $\vec{h}(\vec{x})\ge 0$

- $f$ is the *objective* function and typically a scalar
- $\vec{g}(\vec{x})$ is a list of *equality constraints*
- $\vec{h}(\vec{x})$ is a list of *inequality constraints*
- Maximization for $f$ is mearly minimization of $-f$.
- Optimization differs from root finding in the constraints and smoothness of $f$.

We saw with root finding that using gradient information to help search algorithms has important benefits. That remains true in optimization, but since they are often *high dimensional* ($\vec{x}$ is a large vector), and $f$ is not necessarily *well behaved*, great value is placed on methods that do not require (direct) gradient / Hessian calculations.

## Local vs Global minima

Even in 1D, optimization is confounded by *local minimima*. The lowest of the local minima is the *global minimum*.

There is no good way to ensure a local minimia is a global one without calculating and comparing them all!

There are lots of algorithms that attempt to find the global minimum. They either:

- launch multiple minimizers over the range of arguments
- typically involve preturbing *good* solutions in an attempt to escape from local minima basins.

Interestingly, many of these methods are modelled after nature! (Why?)

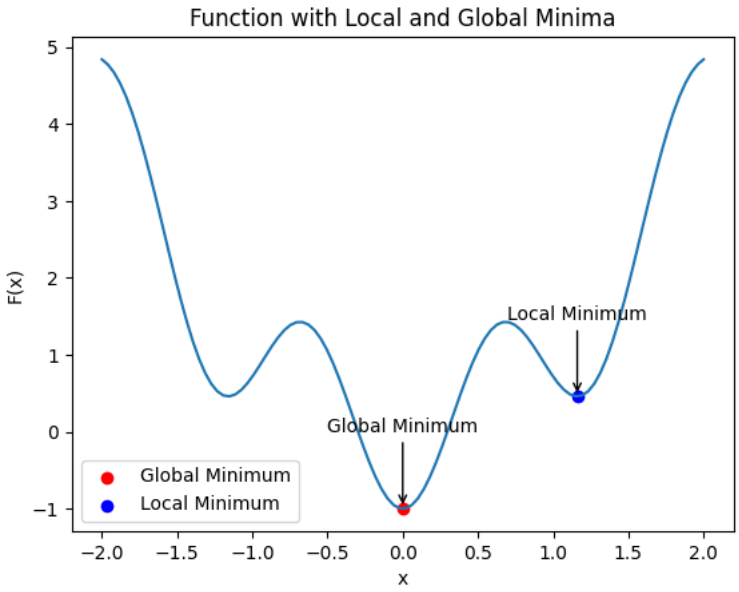
- Genetic algorithms - mutations, cross-overs, selection)
- Simulated annealing - metalurgical process that accepts worse solutions)
- Partical swarm - Mimics swarming behaviour to track multiple solutions
- Amoeba search (which we will talk about)

Introduction of constraints can further complicate things since sometimes the minimima is at a bound!

### Example: local and global minimima

Minimize $f = x^2 - cos(5 x)$

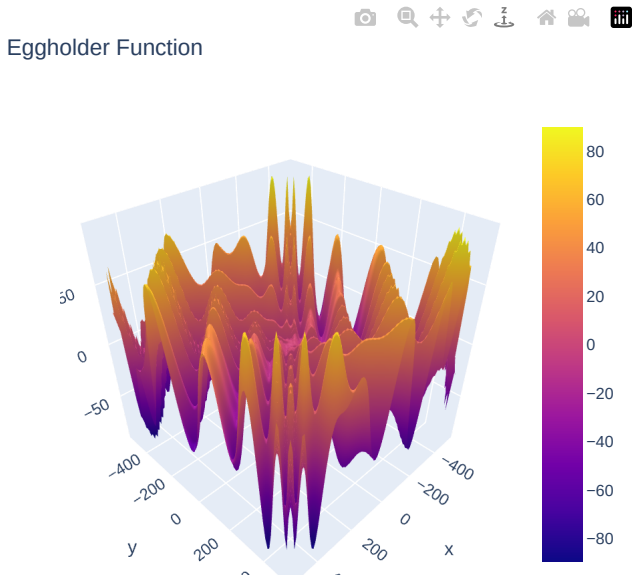
```python
# prompt: Draw a function with a clear local and global minima, labelled

import matplotlib.pyplot as plt
import numpy as np
import scipy as sp

x = np.linspace(-2, 2, 100)
f = lambda x: x**2 + -np.cos(5 * x)
y = f(x)

plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("F(x)")
plt.title("Function with Local and Global Minima")

# Annotate the global minimum
plt.scatter(0, np.min(y), color='red', label='Global Minimum')
plt.annotate('Global Minimum', xy=(0, np.min(y)), xytext=(0, np.min(y) + 1), ha='center', arrowprops=dic

# Annotate a local minimum (approximately)
opt =  sp.optimize.minimize(f,1)
local_min_x = opt.x
local_min_y = opt.fun
plt.scatter(local_min_x, local_min_y, color='blue', label='Local Minimum')
plt.annotate('Local Minimum', xy=(local_min_x, local_min_y), xytext=(local_min_x, local_min_y + 1), ha='

plt.legend()
plt.show()
```




## Example: Eggholder surfaces

Minimize \(y \cdot sin(\sqrt{\|x + y\|}) + x \cdot sin(\sqrt{\|x - y\|})\)

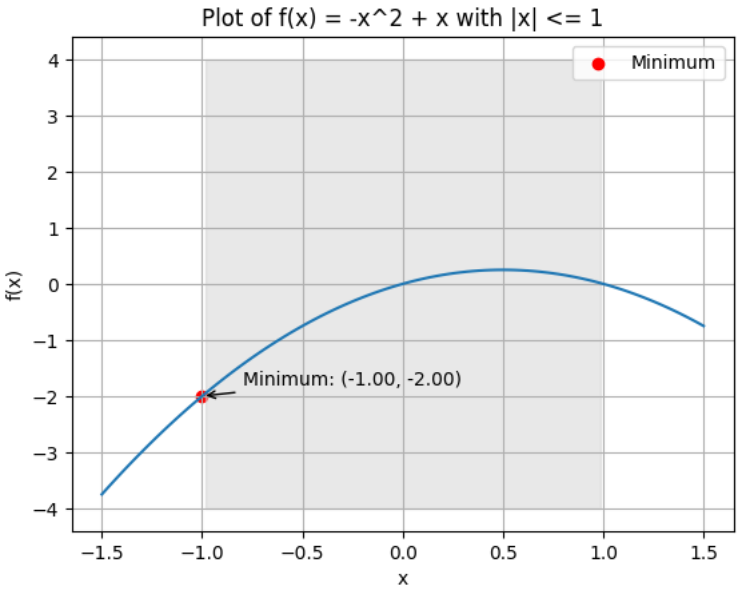
```
11# prompt: Draw a 2D eggholder surface using plotly

import plotly.graph_objects as go
import numpy as np

def eggholder(x, y):
  return y* np.sin(np.sqrt(abs(x + y ))) + x * np.sin(np.sqrt(abs(x - y)))


x = np.linspace(-512, 512, 100)
y = np.linspace(-512, 512, 100)
X, Y = np.meshgrid(x, y)
Z = eggholder(X, Y)/10

fig = go.Figure(data=[go.Surface(z=Z, x=X, y=Y)])
fig.update_layout(title='Eggholder Function', autosize=False,
                  width=500, height=500,
                  margin=dict(l=65, r=50, b=65, t=90))
fig.show()
```




## Example: Constrained minimization

Minimize \(x^2+x\) such that \(|x\| \le 1\)

```python
# prompt: Plot -x^2+x and find the minimum with |x|\le 1 and show the minimium

import matplotlib.pyplot as plt
import numpy as np
import scipy.optimize as optimize

# Define the function
def f(x):
  return -x**2 + x

# Define the bounds for x
bounds = (-1, 1)

# Create a range of x values
x = np.linspace(-1.5, 1.5, 100)

# Calculate the corresponding y values
y = f(x)

# Find the minimum using scipy.optimize.minimize_scalar
result = optimize.minimize_scalar(f, bounds=bounds)

# Get the x value of the minimum
x_min = result.x

# Get the y value of the minimum
y_min = f(x_min)


# Plot the function
plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Plot of f(x) = -x^2 + x with |x| <= 1')

# Plot the minimum
plt.scatter(x_min, y_min, color='red', label='Minimum')
plt.annotate(f'Minimum: ({x_min:.2f}, {y_min:.2f})', xy=(x_min, y_min), xytext=(x_min + 0.2, y_min + 0.2

plt.fill_between(x, -4, 4, where=(np.abs(x) <= 1), color='lightgray', alpha=0.5)
plt.legend()
plt.grid(True)
plt.show()
```




## Bracketed minimization

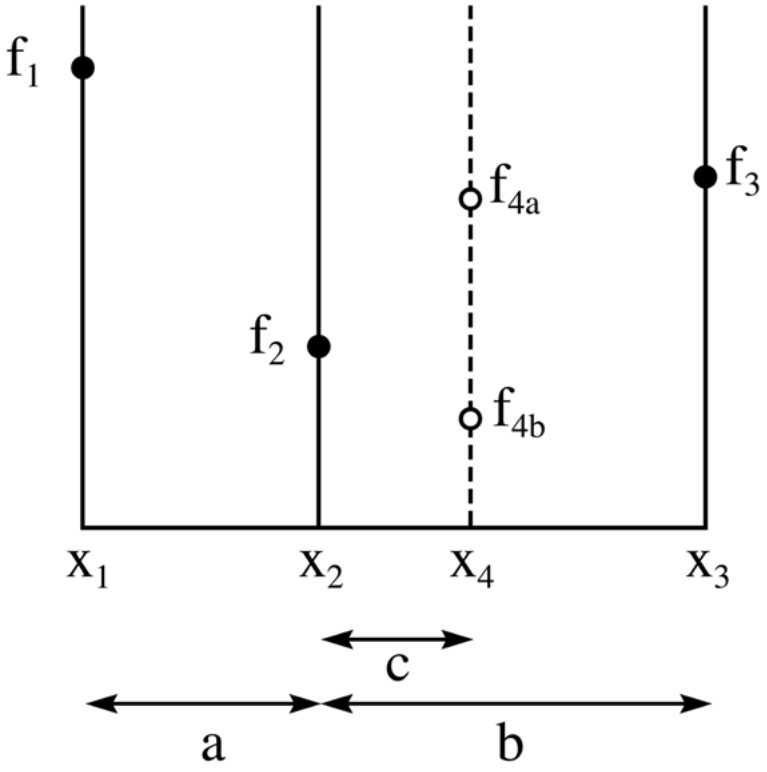
Like in root finding, you will often have *some idea* of where the optimium is. In similar fashion, we can iteratively improve our brackets until they are *close enough*

## Golden section search

In the bisection root finding algorithm, two points were sufficient since they identify a sign change. For minimization, we need to determine a minimum, so need 3 points.

> Given 3 points (\(x_1, x_2, x_3\))
>
> Choose a point \(x_4\)
>
> Evaluate the quadratic at \(x_4\)
>
> If \(f(x_4) \lt f(x_2)\), select <\(x_2, x_4, x_3\)>.
>
> If \(f(x_4) \gt f(x_2)\), select <\(x_1, x_2, x_4\)>
>
> Repeat until a tolerance is met

How do we choose \(x_4\)?




### The Golden ratio

To make a robust method, we want both of these paths to result in the same reduction of the search interval, and ideally reuse the function evaluations. This *self-similarity* implies we want to maintain the ratio of subdivision:

In the tripplet <\(x_1, x_2, x_4\)> with \(f_{4a}\), we want, \(\frac{c}{a}=\frac{a}{b}\).

In the tripplet <\(x_2, x_4, x_3\)> with \(f_{4b}\), we want \(\frac{c}{b-c}=\frac{a}{b}\)

Using these two equations, we can elimiante \(c\) and find, $\({\frac{b}{a}}^2 - \frac{b}{a} = 1\)$

and ignoring the negative root, $\(\frac{b}{a} = \varphi = \frac{1+\sqrt{5}}{2} =1.618033988...\)$

the *Golden ratio*!

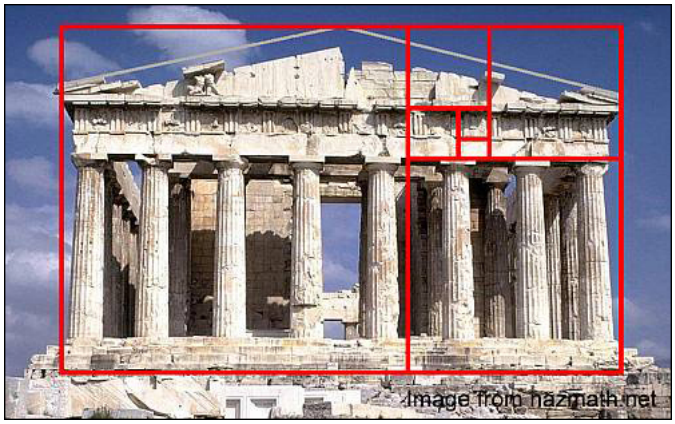
**Aside: The divine proportion**

The Golden ratio appears **all over the place!**

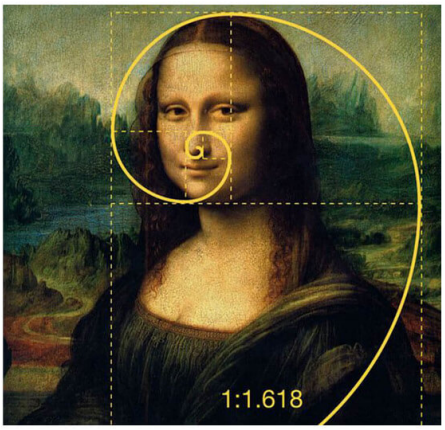
```
import numpy as np

def fibonacci(n):
  """
  Computes the Fibonacci sequence up to n terms.
  """
  fib_sequence = [0, 1]
  while len(fib_sequence) < n:
    next_fib = fib_sequence[-1] + fib_sequence[-2]
    fib_sequence.append(next_fib)
  return fib_sequence

n = 15  # Number of terms in the Fibonacci sequence
fib_sequence = fibonacci(n)
print("Fibonacci sequence:", fib_sequence)

golden_ratios = []
for i in range(2, n):
  golden_ratios.append(fib_sequence[i] / fib_sequence[i-1])

print("Golden ratios:", golden_ratios)
```

```
Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
Golden ratios: [1.0, 2.0, 1.5, 1.6666666666666667, 1.6, 1.625, 1.6153846153846154, 1.619047619047619, 1.6
```

The Greeks knew it!




Nature knows it (kinda)



## Convergence rate

Each iteration of the Golden section method reduces the bracketed interval by $\varphi^{-1} = 0.618$, which becomes its order of convergence.

The Golden section method is gauranteed to converge but does so slowly at a sub-linear rate.

---

# Brent's method

We've seen this concept before in root finding with bisection. Later we saw the value of using the function values to approximate the derivative and guess at the root. Since we had two points, we fit a line (which approximated the derivative) which is the Secant method.

Now we have 3 points so we can fit a parabola and find $f_{min} = f(x_{min})$!

## Inverse quadratic interpolation

BUT there is one problem, we want to estimate $x_{min}$ not $f_{min}$, so what we actually need to approximate is the *inverse quadratic*.

$$ x_4 = \frac{f_2 f_3}{[f_1-f_2][f_1-f_3]} x_1 + \frac{f_1 f_3}{[f_2-f_1][f_2-f_3]} x_2 + \frac{f_3 f_2}{[f_3-f_1][f_3-f_2]} x_3 $$

The convergence order of inverse quadratic approxiation is $\sim 1.84$.

## Combination with Golden section

Brent's method combines the robustness of the Golden Section method with the accelaration of inverse quadratic approximation (still without any derivative information!).

Candidate steps from inverse quadratic interpolation, Golden section, and sometimes Secant, are generated and compared against various criteria for robustness and progression.

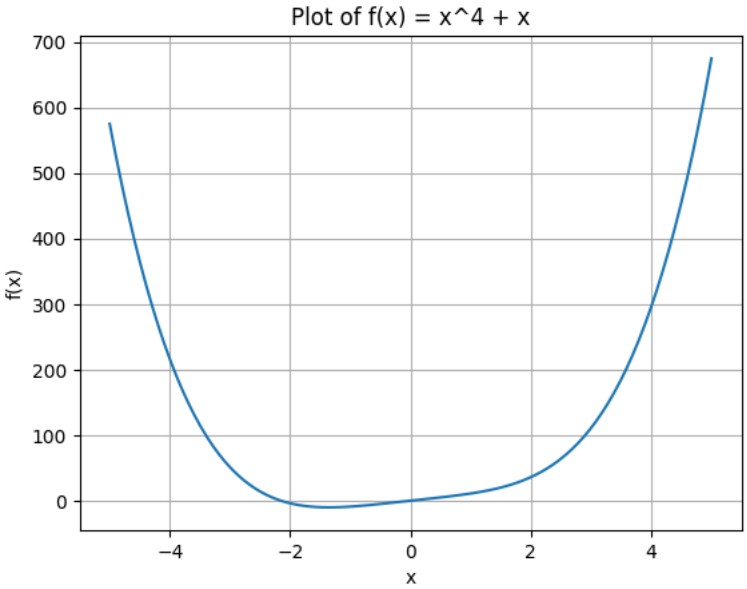
This complicated decision tree makes the order of convergence difficult to assess, but for well behaved functions it typically has $\gt 1.3$.

### Example: Unimodal minimization

Minimize $x^4+10x$

```
# prompt: Plot and minimize x^4+x on the interval -5 5 using brents method

def f(x):
  print(x)
  return x**4 + 10*x

x = np.linspace(-5, 5, 100)
y = f(x)

plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Plot of f(x) = x^4 + x')
plt.grid(True)
plt.show()


result = optimize.minimize_scalar(f, bracket=(-5, 0, 5), method = "Brent")
print("Minimum found at x =", result.fun)
print("Minimum function value =", f(result.x))
```

```
[-5.         -4.8989899  -4.7979798  -4.6969697  -4.5959596  -4.49494949
 -4.39393939 -4.29292929 -4.19191919 -4.09090909 -3.98989899 -3.88888889
 -3.78787879 -3.68686869 -3.58585859 -3.48484848 -3.38383838 -3.28282828
 -3.18181818 -3.08080808 -2.97979798 -2.87878788 -2.77777778 -2.67676768
 -2.57575758 -2.47474747 -2.37373737 -2.27272727 -2.17171717 -2.07070707
 -1.96969697 -1.86868687 -1.76767677 -1.66666667 -1.56565657 -1.46464646
 -1.36363636 -1.26262626 -1.16161616 -1.06060606 -0.95959596 -0.85858586
 -0.75757576 -0.65656566 -0.55555556 -0.45454545 -0.35353535 -0.25252525
 -0.15151515 -0.05050505  0.05050505  0.15151515  0.25252525  0.35353535
  0.45454545  0.55555556  0.65656566  0.75757576  0.85858586  0.95959596
  1.06060606  1.16161616  1.26262626  1.36363636  1.46464646  1.56565657
  1.66666667  1.76767677  1.86868687  1.96969697  2.07070707  2.17171717
  2.27272727  2.37373737  2.47474747  2.57575758  2.67676768  2.77777778
  2.87878788  2.97979798  3.08080808  3.18181818  3.28282828  3.38383838
  3.48484848  3.58585859  3.68686869  3.78787879  3.88888889  3.98989899
  4.09090909  4.19191919  4.29292929  4.39393939  4.49494949  4.5959596
  4.6969697   4.7979798   4.8989899   5.        ]
```




```
-5
0
5
-1.9098300000000001
-3.09016987422
-1.0343458767574745
-1.181577493941968
-1.304005930779925
-1.5354101272036402
-1.3493130244641753
-1.360162925456641
-1.3572686570310966
-1.357202998346505
-1.3572088473834198
-1.3572088082219729
-1.3572087881252826
Minimum found at x = -10.1790660622309
-1.3572088082219729
Minimum function value = -10.1790660622309
```

### Example: Non differentiable function

Let's try a non-differential function: \(f(x) = \|x\|\):

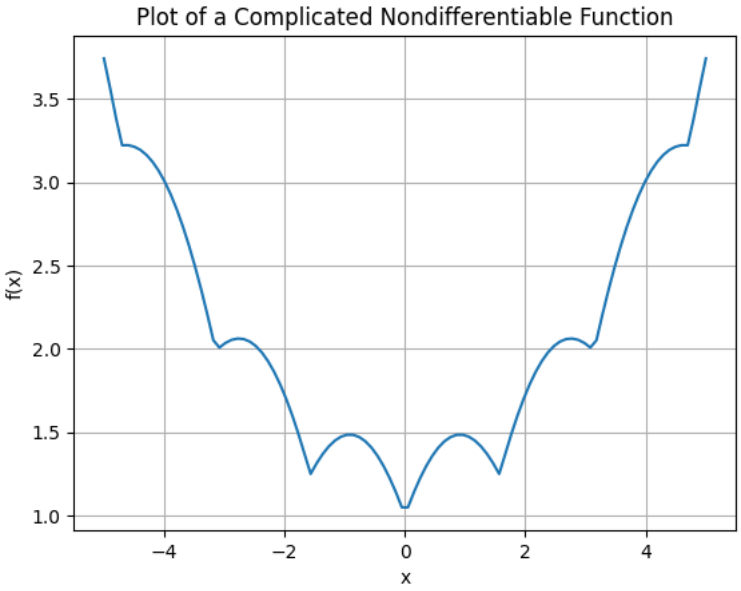
```
# prompt: Use minimize_scalar on a nondifferentiable function

def nondiff_func(x):
  print(x)
  return abs(x)

#~~ What should we use for the bracket? What happens if we use a different method?
bracket = (-5, 5)

##

result = optimize.minimize_scalar(nondiff_func, bracket=bracket, method="Brent")
print("Minimum found at x =", result.x)
print("Minimum function value =", nondiff_func(result.x))
```

```
-5
5
21.18034
11.180339748440002
1.1803399999999997
-1.1803397484400002
-2.0351634599791169e-07
6.288998898266019e-08
0.4508497873081514
0.07788236151205088
0.01345381387145999
0.002324054369767623
0.0004014354205322895
6.931111691218845e-05
1.193804613339047e-05
2.0270904619436587e-06
3.150147269686505e-07
1.926087441788894e-08
-3.182935454321158e-08
3.0662337662403348e-09
-2.8394285480784205e-09
-3.279242240095256e-10
5.0637275275494315e-11
1.2024926045433706e-09
1.8487990512897325e-10
-3.023893922134733e-11
-2.0238938773811033e-11
3.0613588511369835e-12
2.1233741344083055e-11
-6.938641194171126e-12
Minimum found at x = 3.0613588511369835e-12
3.0613588511369835e-12
Minimum function value = 3.0613588511369835e-12
```

### Example: A more treacherous function

```python
# prompt: Plot and Do minimization on a more complicated nondifferentiable function

import matplotlib.pyplot as plt
import numpy as np
import scipy.optimize as optimize


def complicated_nondiff_func(x):
  print(x)
  return np.abs(np.sin(x)) + np.abs(np.cos(x))+.1*x**2


x = np.linspace(-5, 5, 100)
y = complicated_nondiff_func(x)

plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Plot of a Complicated Nondifferentiable Function')
plt.grid(True)
plt.show()

result = optimize.minimize_scalar(complicated_nondiff_func, bracket=(-5, 5), method="Brent")
print("Minimum found at x =", result.x)
print("Minimum function value =", complicated_nondiff_func(result.x))
```

```
[-5.         -4.8989899  -4.7979798  -4.6969697  -4.5959596  -4.49494949
 -4.39393939 -4.29292929 -4.19191919 -4.09090909 -3.98989899 -3.88888889
 -3.78787879 -3.68686869 -3.58585859 -3.48484848 -3.38383838 -3.28282828
 -3.18181818 -3.08080808 -2.97979798 -2.87878788 -2.77777778 -2.67676768
 -2.57575758 -2.47474747 -2.37373737 -2.27272727 -2.17171717 -2.07070707
 -1.96969697 -1.86868687 -1.76767677 -1.66666667 -1.56565657 -1.46464646
 -1.36363636 -1.26262626 -1.16161616 -1.06060606 -0.95959596 -0.85858586
 -0.75757576 -0.65656566 -0.55555556 -0.45454545 -0.35353535 -0.25252525
 -0.15151515 -0.05050505  0.05050505  0.15151515  0.25252525  0.35353535
  0.45454545  0.55555556  0.65656566  0.75757576  0.85858586  0.95959596
  1.06060606  1.16161616  1.26262626  1.36363636  1.46464646  1.56565657
  1.66666667  1.76767677  1.86868687  1.96969697  2.07070707  2.17171717
  2.27272727  2.37373737  2.47474747  2.57575758  2.67676768  2.77777778
  2.87878788  2.97979798  3.08080808  3.18181818  3.28282828  3.38383838
  3.48484848  3.58585859  3.68686869  3.78787879  3.88888889  3.98989899
  4.09090909  4.19191919  4.29292929  4.39393939  4.49494949  4.5959596
  4.6969697   4.7979798   4.8989899   5.        ]
```



```
-5
5
21.18034
11.180339748440002
1.1803399999999997
0.09458844428123059
-1.851371125427094
-0.2178561188045674
0.2955773973951216
0.013367139936113784
-0.03125309331092177
0.00941889115377949
-0.0031614237350019517
-0.005118141940542849
0.0007171089314151327
0.004040893879762758
0.0001702967013971977
-0.0005786354954037355
2.4438122178259777e-05
-9.778851886602696e-05
1.3597865341453028e-05
-1.5788125925539715e-05
2.3734158011648776e-06
-4.6525327979423743e-07
-6.3180696527990555e-06
-7.928505706579973e-07
3.176832330622026e-07
1.102903179170109e-06
5.5181149917313e-08
-3.251689813735922e-08
-4.969018548934073e-08
5.638184896694283e-10
2.1425782105760735e-08
-2.724701120183268e-09
3.707823274770505e-09
1.9492149616025546e-10
-5.01574131468465e-10
-7.111615274257643e-11
-4.8692901721861595e-11
2.3219656378561172e-11
1.313591135533361e-11
-4.650324178731215e-12
-1.465032424755601e-11
Minimum found at x = -4.650324178731215e-12
-4.650324178731215e-12
Minimum function value = 1.0000000000046503
```

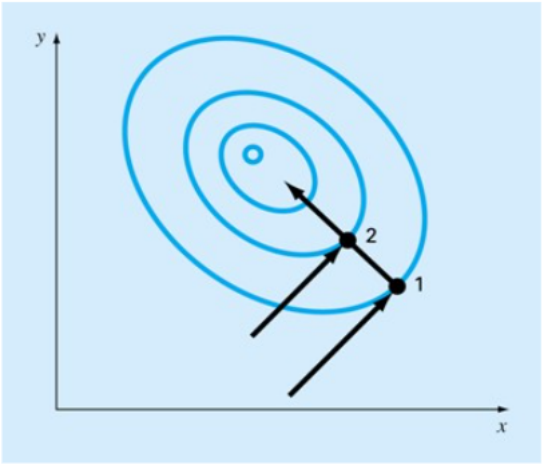
# Open methods

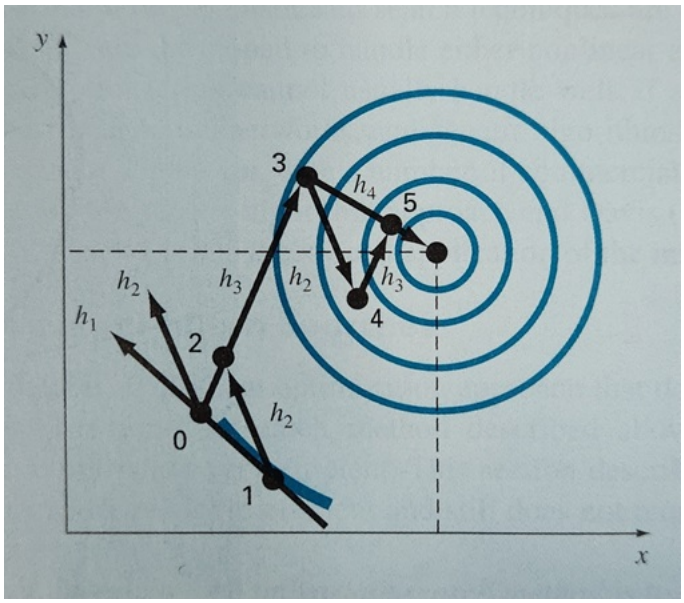
The easiest way to generalize to ND is to take our 1D methods and use them as a search routine (this should sound familiar!)

NB: We have moved beyond *bounds* which are a type of constraint. We are now into *unconstrained minimization*.

Powell's method uses a starting point and some initial direction vectors. The function is optimized along each direction vector.

Subsequent direction vectors are obtained through an important observation: The line connecting the results of the previous two searchs is directed towards the minimum (i.e.: It is conjugate to the other lines).




The algorithm is (Taken from Chapra and Canale - Numerical Methods for Engineers):

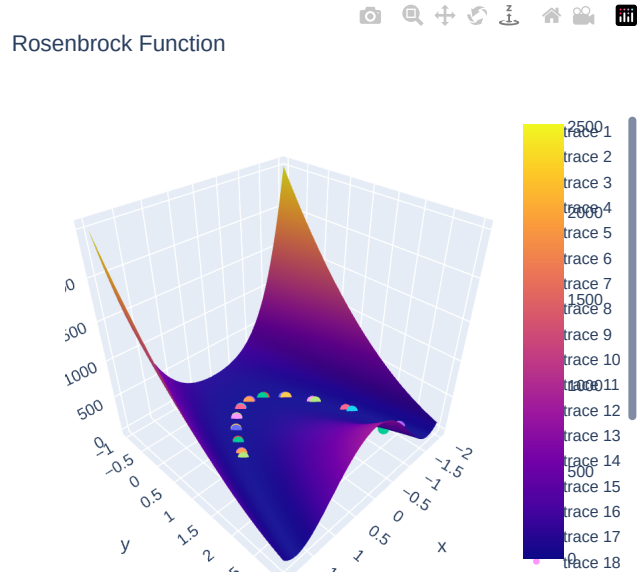
1. Start at 0 with directions \(h_1\) and \(h_2\)
2. Search from 0 along \(h_1\) to get to point 1.
3. Search from 1 along \(h_2\) to get to 2.
4. Define \(h_3\) from 0 to 2.
5. Search from 2 along \(h_3\) to find 3
6. From 3, search along \(h_2\) to get 4
7. From 4 search along \(h_3\) to get 5.
8. Use points 5 and 3 to define \(h_4\).
9. \(h_3\) and \(h_4\) are conjugate and therefore the solution can now be found as a combination of them.

Powell showed that this method generates conjugate directions *without needing to know anything about the function or its derivative*. Further, this method has quadratic convergence near the minimum!

## Example: 2D optimization

```
# prompt: Minimize a rosenbrock function with powels method, showing the guesses

import matplotlib.pyplot as plt
import numpy as np
import scipy as sp
import plotly.graph_objects as go
import scipy.optimize as optimize

def rosenbrock(x):
  return ((1 - x[0])**2 + 100 * (x[1] - x[0]**2)**2)

x0 = np.array([-1, 2.5])
result = optimize.minimize(rosenbrock, x0, method='Powell',options={'disp': True, 'return_all': True})
print('The minimum is, ', result.fun, ' found at ', result.x)
guesses = [x0]
for i in range(result.nit):
  guesses.append(result.allvecs[i])

x = np.linspace(-2, 2, 100)
y = np.linspace(-1, 3, 100)
X, Y = np.meshgrid(x, y)
Z = rosenbrock([X, Y])

fig = go.Figure(data=[go.Surface(z=Z, x=X, y=Y)])
fig.update_layout(title='Rosenbrock Function', autosize=False,
                  width=500, height=500,
                  margin=dict(l=65, r=50, b=65, t=90))

for guess in guesses:
  fig.add_trace(go.Scatter3d(
      x=[guess[0]], y=[guess[1]], z=[rosenbrock(guess)],
      mode='markers',
      marker=dict(
          size=5,
      )
  ))
fig.show()
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 26
         Function evaluations: 684
The minimum is,  2.0830858278492343e-30  found at  [1. 1.]
```



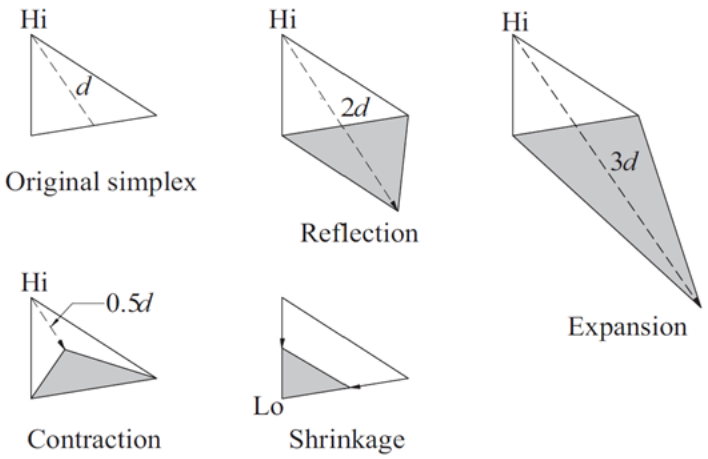
## Nelder-Mead downhill simplex (amoeba method)

The Nelder-Mead method works by moving an N-D *simplex* downhill until it surrounds a minimum, then contracts until a specified threshold is reached.

A simplex is the simplest nD polygon. For 2D this is a triangle, for 3D a tetrahedron, etc.

The algorithm proceeds identifying the ‘Hi’ and ‘Lo’ points:

1. Reflection: Move ‘Hi’ *through* the opposite face, such that the volume of the simplex remains constant.
2. Expansion: Move ‘Hi’ further to increase the simplex volume.
3. Contraction: Move ‘Hi’ a fraction towards the opposite face.
4. Shrinkage: Move all verticies towards ‘Lo’.



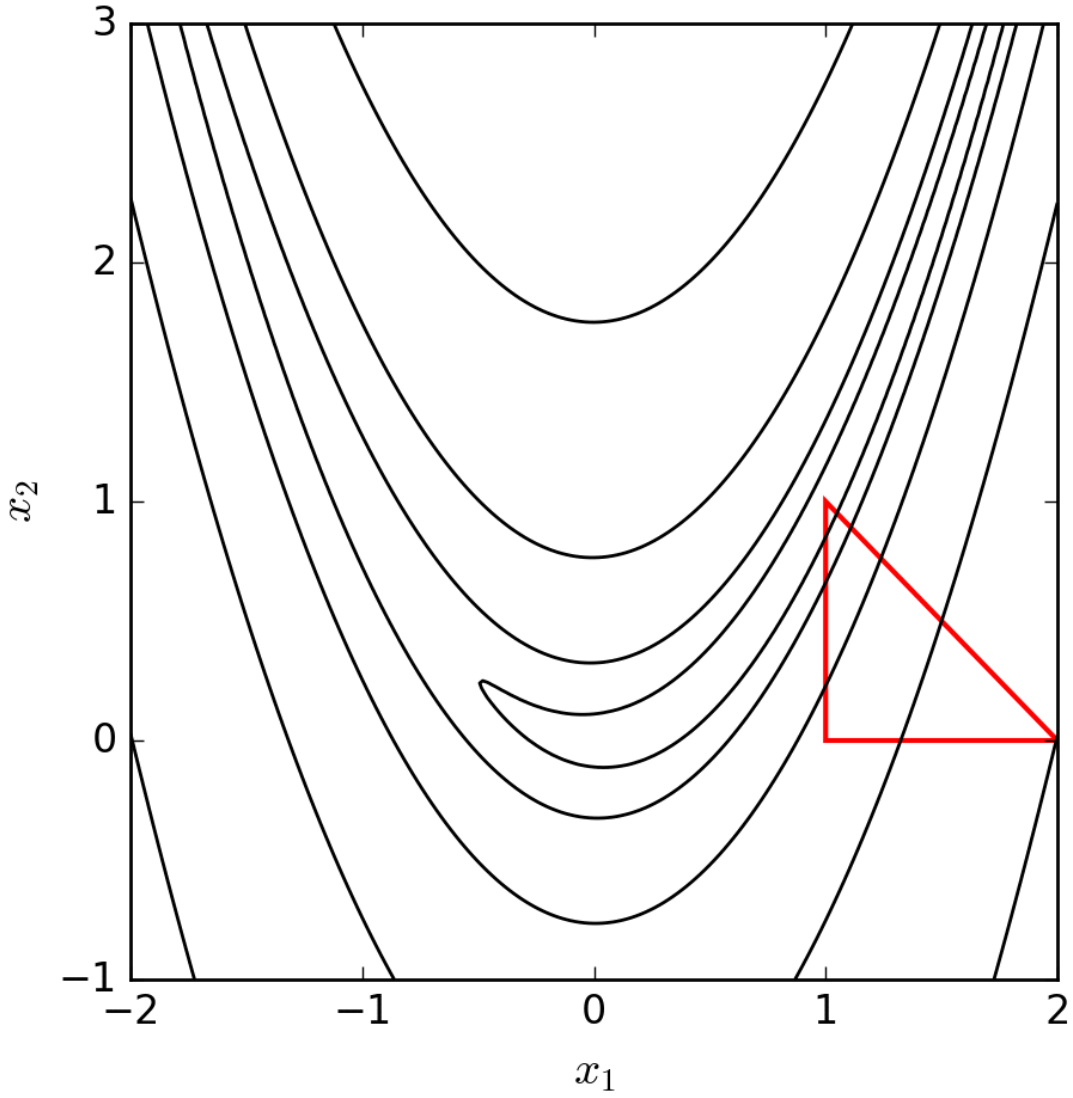

Algorithm:

```
Try reflection.
if new vertex ≤ old Lo: accept reflection
  Try expansion.
  if new vertex ≤ old Lo: accept expansion.
else:
  if new vertex > old Hi:
    Try contraction.
    if new vertex ≤ old Hi: accept contraction.
  else: use shrinkage.
```

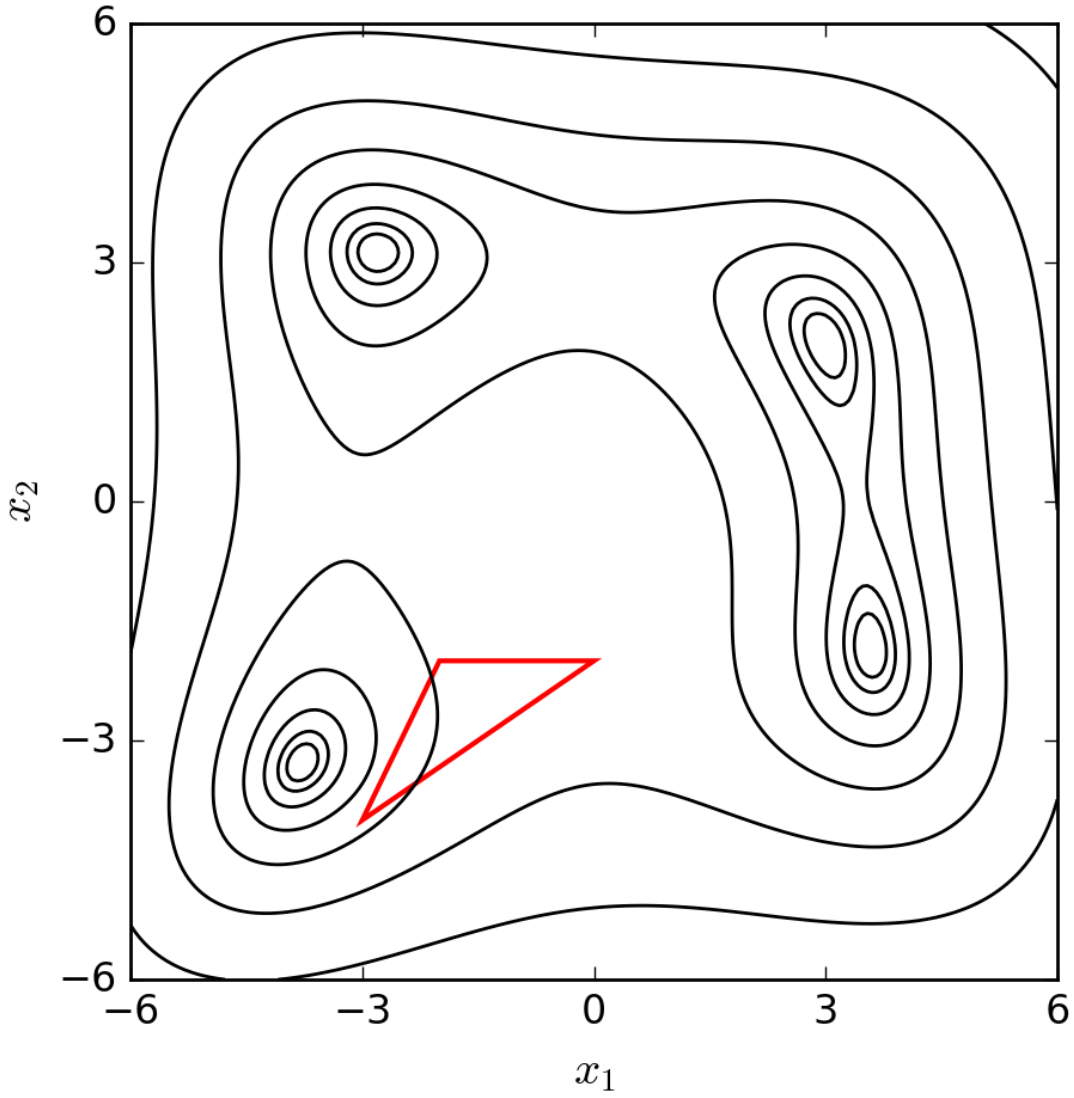
It is sometimes known as the ‘Amoeba method’ since it appears to behave like a cell traversing an energy landscape.

It is much slower than Powell’s method, but it is generally preferred for small dimensional problems due to its robustness.

### Example: Nelder-Mead on the Rosenbrock function




**Example: Nelder-Mead on the Himmelblau function**




## Gradient-based optimization

If information about the gradient is available, it can accelarate convergence substantially.

### Newton's optimization method

Newton's optimization routine aims to find the root of the gradient, which is the extremal. Since we are now focussed on scalar \(f(\vec{x})\) the gradient is a vector and we will need the Hessian matrix $\(H = \frac{\partial^2 f}{{\partial \vec{x}}^2}\)$

The increment is now solved as:

$$ H \Delta \vec{x} = - \nabla f \]

Noting that \(H\) must be symmetric.

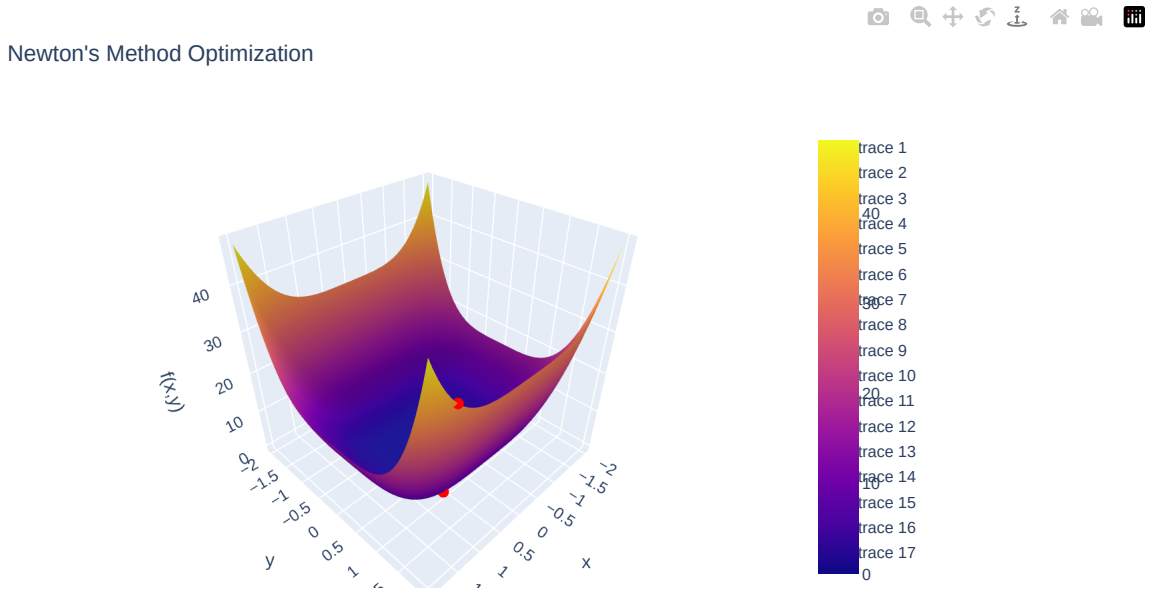
Netwon's optimization method has excellent convergence criteria but requires calculation of the Hessian which can be computationally expensive.

```
import numpy as np
from scipy.linalg import solve
import plotly.graph_objects as go

def newton_method(f, grad_f, hessian_f, x0, tol=1e-6, max_iter=100):
  x = x0
  print(x)
  guesses = [x]
  for _ in range(max_iter):
    grad = grad_f(x)
    hess = hessian_f(x)

    # ~~ What goes here?

    ###
    delta_x = solve(hess, -grad, assume_a = 'sym')
    ###
    x = x + delta_x
    guesses.append(x)
    print(x)
    if np.linalg.norm(grad) < tol:
      break

  # Create a surface plot of the function
  x = np.linspace(-2, 2, 100)
  y = np.linspace(-2, 2, 100)
  X, Y = np.meshgrid(x, y)
  Z = f([X, Y])

  fig = go.Figure(data=[go.Surface(x=X, y=Y, z=Z)])

  # Add markers for each guess
  for guess in guesses:
    fig.add_trace(go.Scatter3d(
        x=[guess[0]],
        y=[guess[1]],
        z=[f(guess)],
        mode='markers',
        marker=dict(
            size=5,
            color='red'
        )
    ))

  fig.update_layout(
      title='Newton\'s Method Optimization',
      scene=dict(
          xaxis_title='x',
          yaxis_title='y',
          zaxis_title='f(x,y)'
      )
  )
  fig.show()
```

### Example: Minimize \(x^4 + y^4\)

```
def f(x):
  return x[0]**4 + 2*x[1]**4

def grad_f(x):
  return np.array([4*x[0]**3, 8*x[1]**3])

def hessian_f(x):
  return np.array([[12*x[0]**2, 0], [0, 24*x[1]**2]])

# Initial guess
x0 = np.array([1.5, 2])

# Perform Newton's method
newton_method(f, grad_f, hessian_f, x0)
```

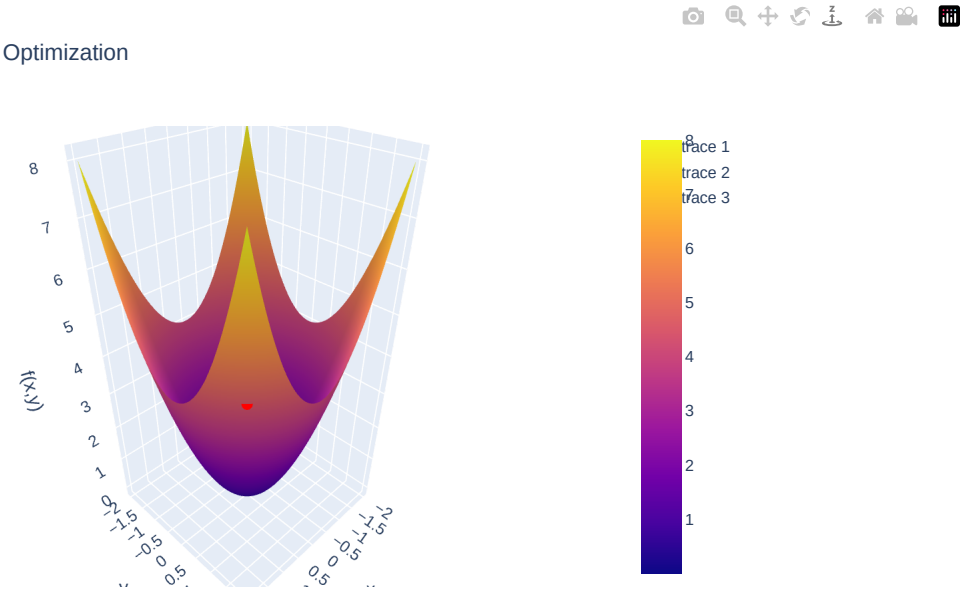
```
[1.5 2. ]
[1.         1.33333333]
[0.66666667 0.88888889]
[0.44444444 0.59259259]
[0.2962963  0.39506173]
[0.19753086 0.26337449]
[0.13168724 0.17558299]
[0.0877915  0.11705533]
[0.05852766 0.07803688]
[0.03901844 0.05202459]
[0.02601229 0.03468306]
[0.01734153 0.02312204]
[0.01156102 0.01541469]
[0.00770735 0.01027646]
[0.00513823 0.00685097]
[0.00342549 0.00456732]
[0.00228366 0.00304488]
```




## Example Minimize \(x^2+y^2\)

```python
import numpy as np
from scipy.linalg import solve
import plotly.graph_objects as go

def f(x):
  return x[0]**2 + x[1]**2

def grad_f(x):
  return np.array([2*x[0], 2*x[1]])

def hessian_f(x):
  return np.array([[2, 0], [0, 2]])

# Initial guess
x0 = np.array([1.5, 1.5])

# Perform Newton's method
newton_method(f, grad_f, hessian_f, x0)
```

```
[1.5 1.5]
[0. 0.]
[0. 0.]
```

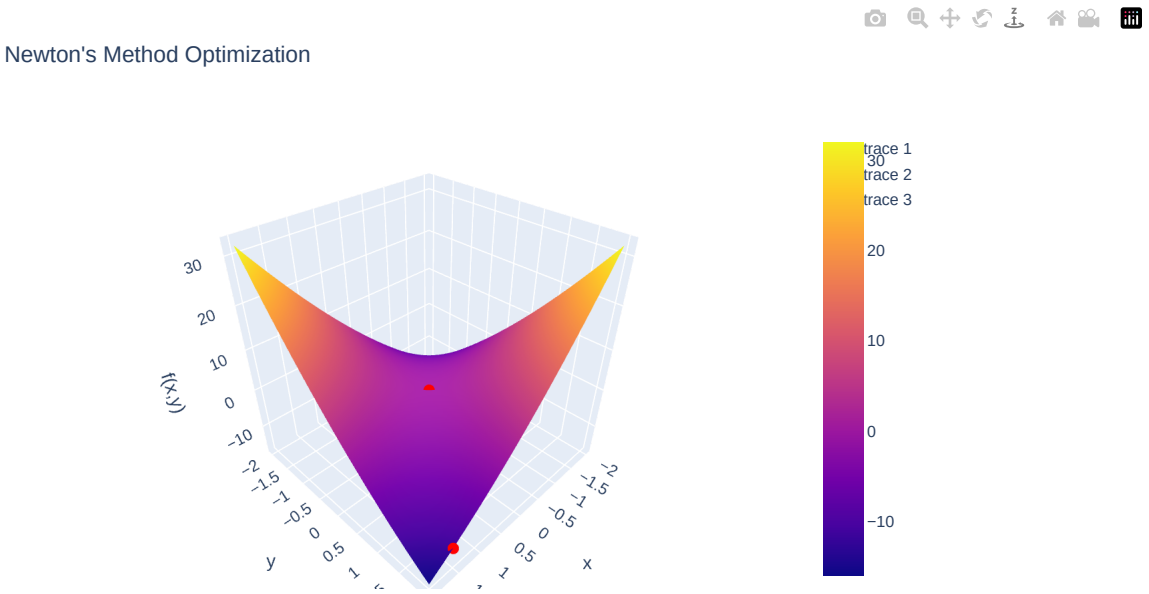



Why does this converge so fast?

## Example: \(x^2 - 6 x y +y^2\)

```python
def f(x):
  return x[0]**2 + x[1]**2 - 6*x[0]*x[1]

def grad_f(x):
  return np.array([2*x[0] - 6*x[1], 2*x[1] - 6*x[0]])

def hessian_f(x):
  return np.array([[2, -6], [-6, 2]])

# Initial guess
x0 = np.array([1.5, 2.])

# Perform Newton's method
newton_method(f, grad_f, hessian_f, x0)
```

```
[1.5 2. ]
[-2.22044605e-16  0.00000000e+00]
[0. 0.]
```




Yeehaw Giddyup!

## Gradient decent methods

If you have information about the gradient (exactly or approximately), moving down the gradient is an intuitive approach to reach the minimum. While fairly fool-proof, as we saw that the steepest decent can lead to *zig-zagging* which motivated constructing orthogonal / conjugate directions which limit their interference with each other.

Recal each step is incremented: $\(\vec{x}^{i+1} = \vec{x}^i+a \vec{p}\)\( where \)a\( is the step length and \)\vec{p}$ is the step direction.

The steepest decent, \(p=-\nabla f\) maximizes the change in \(f\) *in the immediate neighbourhood*, but a different direciton may permit longer step lengths. In general:

\[f(\vec{x}^{1+1})-f(\vec{x}^i) \le -a \|\nabla{f}\| \|p\| \bigg[cos(\theta) -\max_{t ∈ [0,1]} \frac{\|\nabla f(\vec{x}^i-t a \vec{p}) - \nabla f(\vec{x}) \|}{\|\nabla f(\vec{x})\|}\bigg]\]
The second term assesses the rate of change of \(\nabla f\) and involves new quantities, so an exact calculation is typically avoided. Approximations to this term (or alternative algorithmic tools) give rise to different methods.

## Stoichastic gradient decent

Machine learning training involves optimizing a model by finding an appropriate set of parameters. The parameter space can be very high dimensional, and the resulting function can be highly complex. In this case the gradient may be approximated by randomly sampling the change in subsets of parameters. The step length \(a\) is renamed the *learning rate*.

# Indefinite systems

The quadratic \(x^2 - 6 x y +y^2\) is an example of a problem with an *indefinite* matrix. Such functions are neither convex nor concave, and the *extremal point* is neither a minimum nor a maximum, but a *saddle point* (as visualized).

Saddle point problems are an important subclass of optimization as we'll see when we deal with constraints.

Newton's method, and generally optimizers that employ gradients, typically can identify saddle points, but direct methods often fail unless one modifies the objective function (e.g.: Minimize \(f^2\))

## Eigenvalues / functions

The fact that \(f(x,y) = x^2 - 6 x y +y^2\) is indefinite may come as a surprise considering the quadratics along the axes, \(f(x,0) = x^2\) and \(f(0,y) = y^2\) are obviously positive definite.

The key is to realize that we could rotate the coordinate system by \(45^0\) (which obviously wouldn't change the extremal), and write the same function as \(f(x',y') = 8 {x'}^2 - 4 {y'}^2\) from which the saddle point is clear.

*But how would you know that?*

The \(x',y'\) coordiate system is *special* in that its Hessian doesn't have diagonal terms. They are the *eigenvectors* of the Hessian, and the \(8\) and \(-4\) are the *eigenvalues*. With this in mind, definiteness is defined:

| Definiteness | Eigenvalues |
|---|---|
| Positive Definite | All eigenvalues are positive. |
| Negative Definite | All eigenvalues are negative. |
| Indefinite | Eigenvalues are positive and negative. |
| Positive Semi-Definite | All eigenvalues are non-negative (positive or zero). |
| Negative Semi-Definite | All eigenvalues are non-positive (negative or zero). |

```
# prompt: what are the eigen functions and values of x^2 - 6xy + y^2

import numpy as np

def hessian_f(x):
  return np.array([[2, -6], [-6, 2]])

hessian = hessian_f([0,0])

eigenvalues, eigenvectors = np.linalg.eig(hessian)

print("Eigenvalues:", eigenvalues)
print("Eigenvectors:", eigenvectors)
```

```
Eigenvalues: [ 8. -4.]
Eigenvectors: [[ 0.70710678  0.70710678]
 [-0.70710678  0.70710678]]
```

Newton's optimization routine is highly sensative to initial guesses. Furthermore, the calculation and inversion of the Hessian can be computationally expensive and we don't necessarily *trust* that it will give us a good result. We will return to it later in its more common form…

# Trust region methods

Trust region method define a region around the current guess in which the actual function is *trusted* to behave like some model function (to within some tolerance). They are designed to improve robustness for complex surfaces, including indefinate and ill-conditioned surfaces.

They are somewhat *dual* to line search:

- Line search: Choose a direction and search along the line for the next step.
- Trust region: Choose a step length (or limit) and then choose the direction.

The model function can be linear (based on the gradient at the current point) or quadratic (based on the gradient and the Hessian).

Algorithms differ by the criteria for updating the trust region, selection of the next guess, and what to do if the trust region fails.

(E.g.: If the new step deviates too far from the model, does one shrink the trust region for the *next step* or reject the current step and recalculat?).

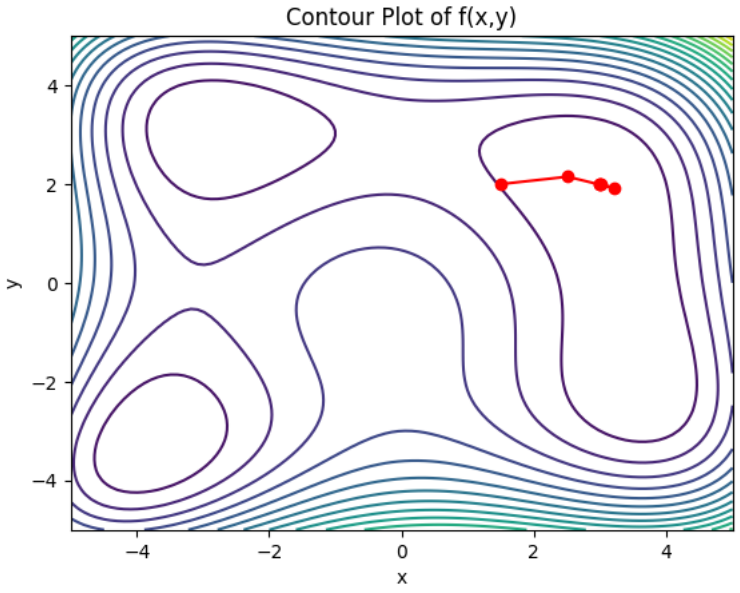
#Comparison of optimization methods discussed *so far*

| Category | Optimization Technique | Method Name |
|---|---|---|
| **Line Search** | Broyden-Fletcher-Goldfarb-Shanno (BFGS) | `BFGS` |
| | Newton-Conjugate-Gradient | `Newton-CG` |
| | Limited-memory BFGS | `L-BFGS-B` |
| | Sequential Least Squares Programming | `SLSQP` |
| **Trust Region** | Trust-Region Newton-Conjugate-Gradient | `trust-ncg` |
| | Trust-Region Truncated Generalized Lanczos | `trust-krylov` |
| | Trust-Region Nearly Exact | `trust-exact` |
| **Direct** | Nelder-Mead Simplex | `Nelder-Mead` |
| | Powell's Method | `Powell` |
| **Gradient Descent** | Conjugate Gradient | `CG` |

## Example - Compare optimizers

```
def f(x):
       return (((x[0]**2+x[1]-11)**2) + (((x[0]+x[1]**2-7)**2)))

def grad_f(x):
  """Gradient of the function f."""
  x1, x2 = x
  df_dx1 = 4 * (x1**2 + x2 - 11) * x1 + 2 * (x1 + x2**2 - 7)
  df_dx2 = 2 * (x1**2 + x2 - 11) + 4 * (x1 + x2**2 - 7) * x2
  return np.array([df_dx1, df_dx2])

def hess_f(x):
  """Hessian of the function f."""
  x1, x2 = x
  d2f_dx1dx1 = 12 * x1**2 + 4 * x2 - 44 + 2
  d2f_dx1dx2 = 4 * x1 + 4 * x2
  d2f_dx2dx1 = 4 * x1 + 4 * x2
  d2f_dx2dx2 = 2 + 4 * x1 + 12 * x2**2 - 28
  return np.array([[d2f_dx1dx1, d2f_dx1dx2], [d2f_dx2dx1, d2f_dx2dx2]])
```

```
from scipy.optimize import minimize
import matplotlib.pyplot as plt

# Create a grid of x and y values
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)

# Calculate the function values for each point on the grid
Z = f([X, Y])

# Initial guess
x0 = np.array([1.5, 2])

guesses = [x0]
def callback(xk):
  guesses.append(xk)

# BFGS
# trust-ncg
# Newton-CG
# Nelder-Mead
# CG
# trust-exact
result = minimize(f, x0, method='trust-exact', jac=grad_f, hess = hess_f, callback=callback)

# Create a contour plot
plt.contour(X, Y, Z, levels=20)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Contour Plot of f(x,y)')

# Plot the path of the guesses
x_guesses = [guess[0] for guess in guesses]
y_guesses = [guess[1] for guess in guesses]
plt.plot(x_guesses, y_guesses, marker='o', linestyle='-', color='red')

plt.show()
```




## Nonlinear least-squares regression

There is one more important unconstrained minimization that we should discuss: non-linear least-squares, which is the extension of the linear least-square regression we did for curve fitting.

Now, we have several measurements in pairs \((x_i, y_i)\) and we wish to fit a function of \(x\) which depends in some complex, nonlinear fashion on a set of parameters \(\beta\). E.g.: \(f(x,\beta) = sin(\beta_1 x) e^{\beta_2 x^3}\)

Our goal is to choose the set of *parameteres* $\beta$ that minimizes the of the sum of the least squares of the residuals, $r_i = f(x_i,\beta) - y_i$:

$$\begin{align} \min_\beta F = \min_\beta \sum_i r_i^2 = \min_\beta \sum_i [f(x_i,\beta) - y_i]^2 \end{align}$$

Note: $F$ is necessarily symmetric and positive semidefinite, therefore any extremum is a minimum.

## Gauss-Newton's regression:

Let's consider Newton's optimization:

$$ \frac{\partial^2 F}{\partial \beta^2} \Delta \beta = - \frac{\partial F}{\partial \beta} $$

Since $F$ is the sum of squared $r_i$ we can expand: $
$ \begin{align} \frac{\partial F}{\partial \beta} = 2 r \frac{\partial r}{\partial \beta} = 2 r J \end{align} $$

where $J = \frac{\partial r}{\partial \beta}$ is the Jacobian of the residuals with respect to the parameters. Since we typically have more data points (residuals) than parameters, this is rectangular.

The Hessian is expanded: $
$ \begin{align} \frac{\partial^2 F}{\partial \beta^2} &= 2 \sum_i \frac{\partial r_i}{\partial \beta_j} \frac{\partial r_i}{\partial \beta_k} + r \frac{\partial^2 r_i}{\partial \beta_j \partial \beta_k} \\ &= 2\bigg[ J J^T + r \frac{\partial^2 r_i}{\partial \beta_j \partial \beta_k} \bigg] \end{align} $$
$

The second derivative in the Hessian is troublesome for computation. Luckily, near the root $r\rightarrow 0$ and we can disregard the term entirely:

$$\begin{align} \frac{\partial^2 F}{\partial \beta^2} \approx 2 \sum_i J J^T \end{align} $$

Our minimization now becomes,

$$\begin{split} \begin{align} J J^T \Delta \beta &= -J r \\ \Delta \beta &= -[J J^T]^{-1} J r \\ &= -J^\dagger r \end{align} \end{split}$$

where $J^\dagger$ is the pseudoinverse of the Jacobian!

This is an interesting result since it suggests we could write the original problem as solving $J \Delta \beta = -r$ which is the Newton-Raphson method for *root finding*, not minimization… Remember however, that this is the *pseudoinverse*, not the true inverse. Indeed, if we had exactly as many data points as parameters, $J$ would be square, and we *would* be able to find the exact solution which is the root! The fact that this is analagously true for least squares regression comes from 1) approximating the Hessian as the product of Jacobians, and 2) properties of the pseudo-inverse in performing linear least-squares.

## The Levenberg-Marquardt algorithm

The problem with Gauss-Newton is that the Hessian approximation is only good near the root, and the method fails unless the initial guess is good.

Levenberg-Marquardt blends Gauss-Newton with Gradient decent:

$$ [J J^T + \lambda I ] \Delta \beta = -J r $$

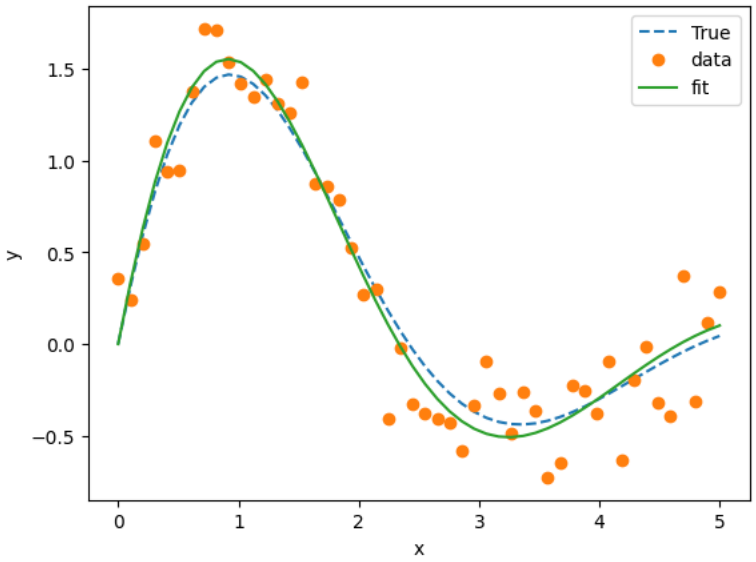
When $\lambda$ is small, the method reduces to Gauss-Newton. When it is large, the $J J^T$ term can be disregarded and the method is simply gradient decent. Determination of $\lambda$ is a matter of heuristics and implementation.

The Levenberg-Marquart algorithm is the go-to for common lst solvers including scipy, numpy, and tools like Excel.

### Example: Nonlinear curve fit

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Define the function to fit
def func(x, a, b, c):
  return a * np.sin(b * x) * np.exp(-c * x)

# Generate some sample data
x_data = np.linspace(0, 5, 50)
y_data = func(x_data, 2.5, 1.3, 0.5) + np.random.normal(0, 0.2, 50)

# Perform the curve fit
popt, _ = curve_fit(func, x_data, y_data)

# Print the fitted parameters
print("Fitted parameters:", popt)

# Generate the fitted curve
y_fit = func(x_data, *popt)

# True function
y_true = func(x_data, 2.5, 1.3, .5)

# Plot the data and the fitted curve
plt.plot(x_data, y_true, '--', label='True')
plt.plot(x_data, y_data, 'o', label='data')
plt.plot(x_data, y_fit, '-', label='fit')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

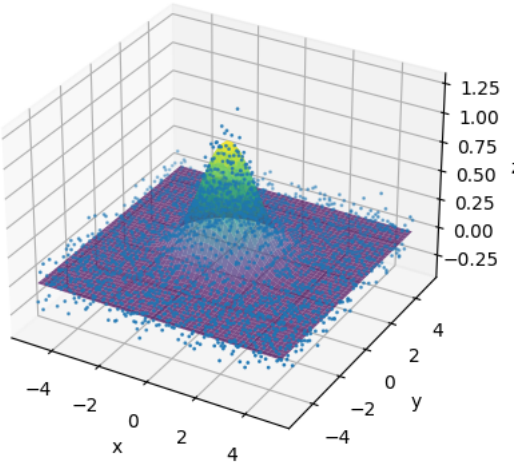
```
Fitted parameters: [2.55266109 1.34621434 0.47897151]
```




**Example: Nonlinear 2D curve fit**

```python
# prompt: make a 2D example of a nonlineaar curve_fit

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from mpl_toolkits.mplot3d import Axes3D

# Define the function to fit (e.g., a 2D Gaussian)
def func(X, a, b, c, d, e, f):
  x, y = X
  return a * np.exp(-((x - b)**2 / (2 * c**2) + (y - d)**2 / (2 * e**2))) + f

# Generate some sample data
x_data = np.linspace(-5, 5, 50)
y_data = np.linspace(-5, 5, 50)
X, Y = np.meshgrid(x_data, y_data)
Z_data = func((X, Y), 1, 0, 1, 0, 1, 0) + np.random.normal(0, 0.1, (50, 50))

# Flatten the data for curve_fit
x_data_flat = X.flatten()
y_data_flat = Y.flatten()
z_data_flat = Z_data.flatten()

# Perform the curve fit
initial_guess = [1, 0, 1, 0, 1, 0]  # Provide an initial guess for the parameters
popt, _ = curve_fit(func, (x_data_flat, y_data_flat), z_data_flat, p0=initial_guess)

# Print the fitted parameters
print("Fitted parameters:", popt)

# Generate the fitted surface
Z_fit = func((X, Y), *popt)


# Plot the data and the fitted surface
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x_data_flat, y_data_flat, z_data_flat, label='data', s = 1)
ax.plot_surface(X, Y, Z_fit, cmap='viridis', alpha=0.7, label='fit')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.show()
```

```
Fitted parameters: [ 9.99242341e-01 -1.10886030e-03  1.03302831e+00  6.17376908e-04
  9.68851674e-01 -3.79420297e-03]
```




## Equality constrained optimization

So far we have discussed how to find \(x\) that minimizes \(f(x\)), which is called ‘unconstrained minimization’. We now add *equality constraints*:

Find \(x\) that minimizes \(f(x)\) subject to \(g(x) = 0\). Note that \(g(x)\) can be a vector function if there are multiple equality constraints.

E.g.: If you wanted to minimize something on the unit circle, your constraint would be,

\[\begin{split}\begin{align} x^2+y^2&=1 \\ x^2+y^2-1 &= 0 \\ g(x,y) &=0 \end{align} \end{split}\]

## Penalty method

One way to enforce the constraints is to add them to the objective function as a penalty:

$\( f(x) + \lambda g(x) \)\( where \)\lambda\( is some suitably chosen constant. If the constraint is violate, i.e.: \)g(x) \ne 0 \(, the combined objective is penalized. By increasing \)\lambda$, one enforces the penalty more severely.

Recall that the solvability of a system is related to its landscape. An excessively large \(\lambda\) will enforce the condition but introduce significant terrain to the surface. However, too small a \(\lambda\) will not affect the constraint at all! Algoithms that use this method have heuristic mecahnisms for adjusting the penalty depending on the violation of the constraints.

Note however, that there is a Goldilocks value that is *just right* in that it is the smallest value that is large enough to affect the constraint…

## Method of undetermined Lagrange Multipliers

At the optimium, we know that \(g(x)=0\) in which case it doesn't matter what \(\lambda\) is, as long as it is *big enough to enforce the constraint*.

In this case, why not add \(\lambda\) as an unknown variable to be solved for? In this case, \(\lambda\) are called the Lagrange Mulitpliers. The summation of the objective function and the penalized equality constraints is called the Lagrangian,

\[ L(x, \lambda) = f(x) + \lambda \cdot g(x) \]

and is now a function of \(\lambda\) as well.

The system is solved for the stationary point, $\(\frac{\partial L}{\partial [x,\lambda] = 0}\)\( which implies, starting with \)\lambda$:

\[\frac{\partial L}{\partial \lambda} = 0 = g(x)\]

and thus the constraints are enforced.

The derivative wrt \(x\) is more interesting and reads,

\[\begin{split} \begin{align} \frac{\partial L}{\partial x} = 0 &= \nabla f(x) + \lambda \nabla g(x) \\ \nabla f(x) &= -\lambda \nabla g(x) \end{align} \end{split}\]
This means that along the curve defined by \(g(x)=0\) there is a point where the the gradient \(\nabla f\) is exactly perpendicular to the curve (and therefore parallel to \(\nabla g\)). The difference in magnitude, being the force with which one wants to keep to the curve, is \(\lambda\).

> Consider a 2D analogy: You are hiking on a mountain along a path (\(g(x)=0\)). As you go, the path leads you along the height of the mountain (\(f(x)\)). You want to find the lowest point, so you simply walk 'downhill' along the path until you reach the lowest point. Here, the ground slopes exactly away (perpendicular) from the path. What's more, for you to stay on the path, you will have to push your legs *exactly uphill* with precisely the necessary force \(\lambda\), so as to not fall to your death but stay on the path.

### Quadratic programming with equality constraints

The interesting thing about this technique is that we have replaced a constrained optimization with an unconstrained method, which can now be solved using our established techniques!

Unfortuantely, except in certain cases this doesn't usually work… One case where it works well is for a quadratic objective function with a set of equality constraints.

$$g(x) = A x-b = 0$$

As we discussed, quadratic functions can be solved directly through Newton's method. With a quadratic objective function $f(x)$ and linear constraints $g(x) = Ax-b$ the Lagrangian is also quadratic.

$$ L(x, \lambda) = f(x) + \lambda \cdot [Ax-b] $$

We have,
$$\begin{align} \frac{\partial f}{\partial x} &= H x + c \\ \frac{\partial L}{\partial x} &= H x + c + \lambda A \end{align} $$

and the Jacobian of the Lagrangian can be written as a block matrix,

$$\begin{split} \begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} = \begin{bmatrix} -c \\ b \end{bmatrix} \end{split}$$
which is, once again, a (square) linear system!

NB: Don't be scared off by the 0 block on the diagonal. For well-formed problems this matrix is non-singular *even if H is not!*

**Example: Minimize $x y +yz$ subject to $x+2y = 6$ and $x=3z$**

We have the objective function, $f(x) = xy+yz$ with Hessian $$H=\begin{bmatrix} 0 & 1 & 0 \\ 1& 0 & 1 \\ 0 & 1& 0\end{bmatrix} $$ and $$c = \begin{bmatrix} 0 \\0 \\ 0 \end{bmatrix}$$

The constraints are $$ g = \begin{bmatrix} 1 & 2 & 0 \\ 1 & 0 & -3 \end{bmatrix} x - \begin{bmatrix} 6 \\ 0 \end{bmatrix} = 0$$

Our KKT matrix then is,

```
import scipy as sp
A = np.array([
    [0, 1, 0, 1, 1],
    [1, 0, 1, 2, 0],
    [0, 1, 0, 0, -3],
    [1, 2, 0, 0, 0],
    [1, 0,-3, 0, 0]])

b = np.array([0,0,0,6,0])

sp.linalg.solve(A, b)
```

```
array([ 3. ,  1.5,  1. , -2. ,  0.5])
```

Lets dig into this matrix a little more:

```
evals, evects= sp.linalg.eig(A)
print(np.round(evals,2))
```

```
[-3.34+0.j -0.82+0.j -1.9 +0.j  2.81+0.j  3.26+0.j]
```

Note that the eigenvalues are of opposite sign! This means that the system is indefinite, and the optimum is a saddle point.

This is typical of Lagrange Multiplier problems and is the reason why standard optimization techniques have difficulty. However, the block structure of the KKT matrix does make it ammenable to block matrix inversion (and the zero block helps!).

Constrained optimization algorithms exploit these properties.

## Meaning of the Lagrange Multipliers

The Lagrange Multipliers are also physically important parameters and this is why they are sought-after over penalty methods. They give the *value* of relaxing a constraint.

- In engineering, they are the *constraint force* associated with some imposed condition.
- In economics, this is *marginal cost / shadow price* of the constraint and tell you the worth of doing a little bit more.
- In thermodynamics they are the *conjugate state variable* corresponding to a particular equilibrium.

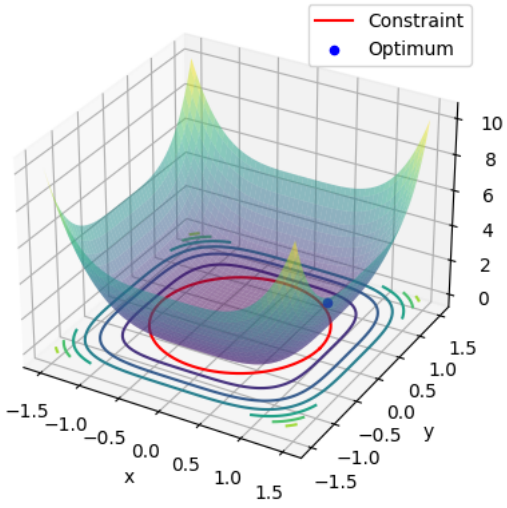
#General constrained optimization

The algorithms for solving the general optimizaiton problem subject to nonlinear constraints and inequality constraints, generally involve defining *feasible* regions for which inequalities are satisfied, and searching *interior points* for the constrained optimum. The derivations can be very sophisticated, well beyond the scope of this course!

### Example: Minimize \(x^4 + y^4\) on the unit circle.

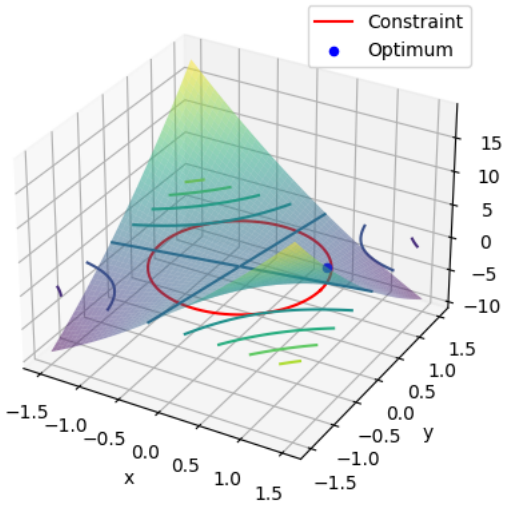
```
# prompt: Optimize x^4+y^4 subject to x^2+y^2-1, plot the objective, constraint,  and the optimization pa

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from mpl_toolkits.mplot3d import Axes3D


def objective(x):
  return x[0]**4 + x[1]**4

def constraint(x):
  return x[0]**2 + x[1]**2 - 1

# Initial guess
x0 = [0.5, 0.5]

# Constraints
cons = ({'type': 'eq', 'fun': constraint})

# Optimization
sol = minimize(objective, x0, method='SLSQP', constraints=cons)

# Print results
print(sol)

# Plot the objective function and constraint
x = np.linspace(-1.5, 1.5, 100)
y = np.linspace(-1.5, 1.5, 100)
X, Y = np.meshgrid(x, y)
Z = X**4 + Y**4

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.5)
ax.contour(X, Y, Z, zdir='z', offset=0, cmap='viridis')

# Plot the constraint
theta = np.linspace(0, 2 * np.pi, 100)
x_constraint = np.cos(theta)
y_constraint = np.sin(theta)
ax.plot(x_constraint, y_constraint, 0, color='red', label='Constraint')

# Plot the optimum
ax.scatter(sol.x[0], sol.x[1], sol.fun, color='blue', marker='o', label='Optimum')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('Objective')
ax.legend()
plt.show()
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: 0.49999938348959805
       x: [ 7.071e-01  7.071e-01]
     nit: 23
     jac: [ 1.414e+00  1.414e+00]
    nfev: 93
    njev: 23
```



```python
# prompt: minimize x^2-6xy+y^2 on the unit circle using trust-const and plot hte surface and the constrai

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from mpl_toolkits.mplot3d import Axes3D

def objective(x):
  return x[0]**2 - 6*x[0]*x[1] + x[1]**2

def constraint(x):
  return x[0]**2 + x[1]**2 - 1

# Initial guess
x0 = [0., 0.]

# Constraints
cons = ({'type': 'eq', 'fun': constraint})

# Optimization using trust-constr
sol = minimize(objective, x0, method='trust-constr', constraints=cons)

# Print results
print(sol)

# Plot the objective function and constraint
x = np.linspace(-1.5, 1.5, 100)
y = np.linspace(-1.5, 1.5, 100)
X, Y = np.meshgrid(x, y)
Z = X**2 - 6*X*Y + Y**2

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.5)
ax.contour(X, Y, Z, zdir='z', offset=0, cmap='viridis')

# Plot the constraint
theta = np.linspace(0, 2 * np.pi, 100)
x_constraint = np.cos(theta)
y_constraint = np.sin(theta)
ax.plot(x_constraint, y_constraint, 0, color='red', label='Constraint')

# Plot the optimum
ax.scatter(sol.x[0], sol.x[1], sol.fun, color='blue', marker='o', label='Optimum')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('Objective')
ax.legend()
plt.show()
```

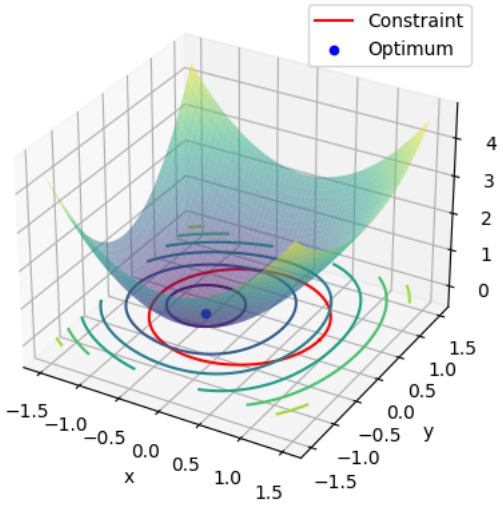
```
         message: `gtol` termination condition is satisfied.
         success: True
          status: 1
             fun: -2.000000000000006
               x: [ 7.071e-01  7.071e-01]
             nit: 6
            nfev: 18
            njev: 6
            nhev: 0
        cg_niter: 4
    cg_stop_cond: 1
            grad: [-2.828e+00 -2.828e+00]
 lagrangian_grad: [-4.441e-16 -4.441e-16]
          constr: [array([ 3.109e-15])]
             jac: [array([[ 1.414e+00,  1.414e+00]])]
     constr_nfev: [18]
     constr_njev: [0]
     constr_nhev: [0]
               v: [array([ 2.000e+00])]
          method: equality_constrained_sqp
      optimality: 4.440892098500626e-16
constr_violation: 3.1086244689504383e-15
  execution_time: 0.012612104415893555
       tr_radius: 5.6000000000000005
  constr_penalty: 54232153.21343036
            niter: 6
```



```python
# prompt: Minimize a complex 2D function on the unit disk using trust-const

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from mpl_toolkits.mplot3d import Axes3D
import scipy as sp

def objective(x):
  """Objective function to minimize."""
  return np.sin(x[0]) * np.cos(x[1]) + x[0]**2 + x[1]**2

def constraint(x):
  """Constraint function: x^2 + y^2 <= 1 (unit disk)."""
  return x[0]**2 + x[1]**2 - 1

# Initial guess
x0 = [0.5, 0.5]

# Constraints
cons = sp.optimize.NonlinearConstraint(lambda x: x[0]**2 + x[1]**2 - 1, -np.inf, 0)

# Optimization using trust-constr
sol = minimize(objective, x0, method='trust-constr', constraints=cons)

# Print results
print(sol)

# Plot the objective function and constraint
x = np.linspace(-1.5, 1.5, 100)
y = np.linspace(-1.5, 1.5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(X) * np.cos(Y) + X**2 + Y**2

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.5)
ax.contour(X, Y, Z, zdir='z', offset=0, cmap='viridis')

# Plot the constraint (unit disk boundary)
theta = np.linspace(0, 2 * np.pi, 100)
x_constraint = np.cos(theta)
y_constraint = np.sin(theta)
ax.plot(x_constraint, y_constraint, 0, color='red', label='Constraint')

# Plot the optimum
ax.scatter(sol.x[0], sol.x[1], sol.fun, color='blue', marker='o', label='Optimum')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('Objective')
ax.legend()
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/scipy/optimize/_differentiable_functions.py:231: UserWarning: del
  self.H.update(self.x - self.x_prev, self.g - self.g_prev)
/usr/local/lib/python3.10/dist-packages/scipy/optimize/_differentiable_functions.py:504: UserWarning: del
  self.H.update(delta_x, delta_g)
```

```
           message: `xtol` termination condition is satisfied.
           success: True
            status: 2
               fun: -0.23246557515821553
                 x: [-4.502e-01 -5.316e-10]
               nit: 55
              nfev: 156
              njev: 52
              nhev: 0
          cg_niter: 48
      cg_stop_cond: 4
              grad: [ 5.588e-09 -2.049e-08]
   lagrangian_grad: [ 1.440e-09 -2.049e-08]
            constr: [array([-7.973e-01])]
               jac: [array([[-9.004e-01, -1.490e-08]])]
      constr_nfev: [156]
      constr_njev: [0]
      constr_nhev: [0]
                 v: [array([ 4.607e-09])]
            method: tr_interior_point
       optimality: 2.0489096710195873e-08
  constr_violation: 0.0
    execution_time: 0.08072233200073242
         tr_radius: 1.0000000000000005e-09
    constr_penalty: 1.0
 barrier_parameter: 2.048000000000001e-09
 barrier_tolerance: 2.048000000000001e-09
             niter: 55
```



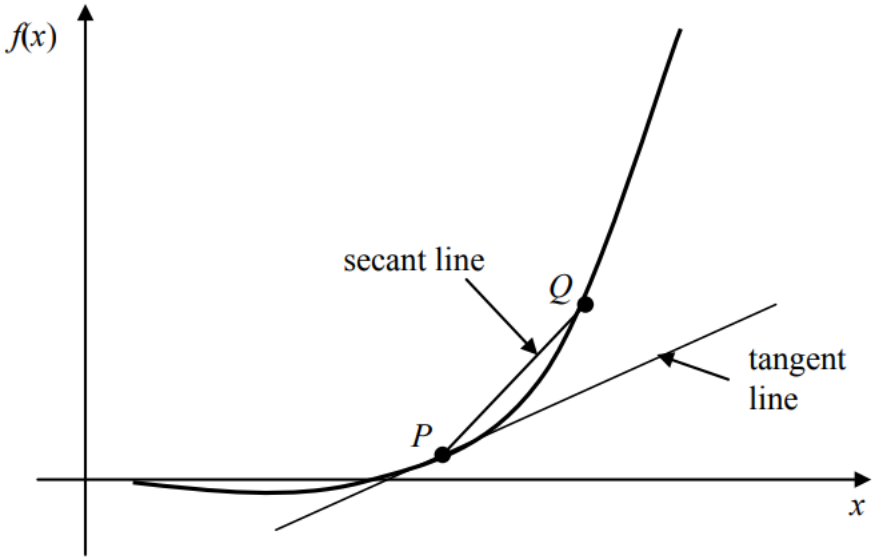

# Differentiation and integration

## Finite difference

Finite difference

### First order derivatives

We have already approached this in the context of the *secant* search methods which approximate the tangent of the curve.




Note that the secant becomes the tangent when $Q \rightarrow P$. Most methods are characterized in terms of the *step size*, in this case the $x$ distance between P and Q.

This notion of the secant approaching the tangent motivates the *forward difference* formula.

## Forward difference

The forward difference is so named because it starts at $x$ and looks *forward* to obtain derivative information.

Consider the Taylor expansion of $f(x)$ from a point $x_i$, with the step size $h$:

$$\begin{aligned} f(x+h) &= f(x) + f'(x)\, h + f''(x) \frac{h^2}{2} + f'''(x) \frac{h^3}{6} \end{aligned}$$

Consider truncating the series after the first two terms:

$$f(x+h) = f(x) + f'(x)\, h + O(h^2)$$

and solve for $f'(x) = \frac{f(x+h) - f(x)}{h} + O(h^2)$

#### Example - Rocket velocity

A rocket has velocity:

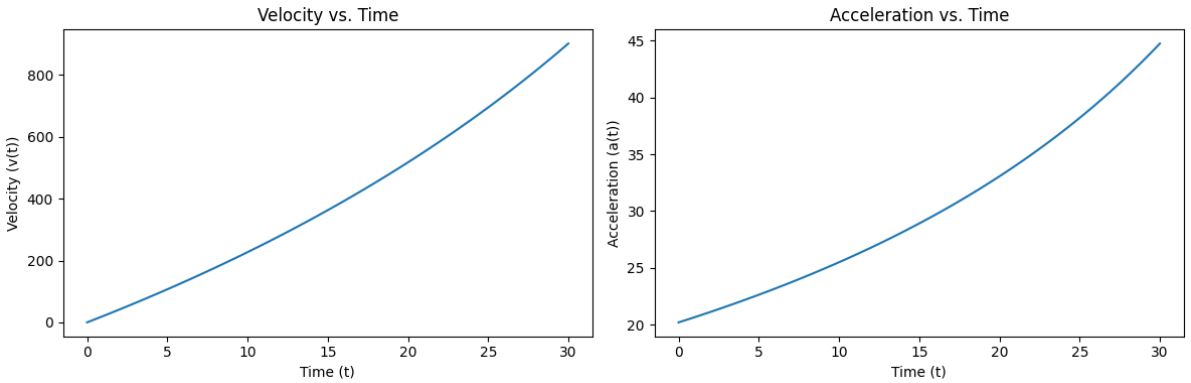
$$v(t) = 2000 \ln \bigg[\frac{14 \times 10^4}{14 \times 10^4 - 2100t}\bigg] - 9.8\, t$$

What is the acceleration as a function of $t$?

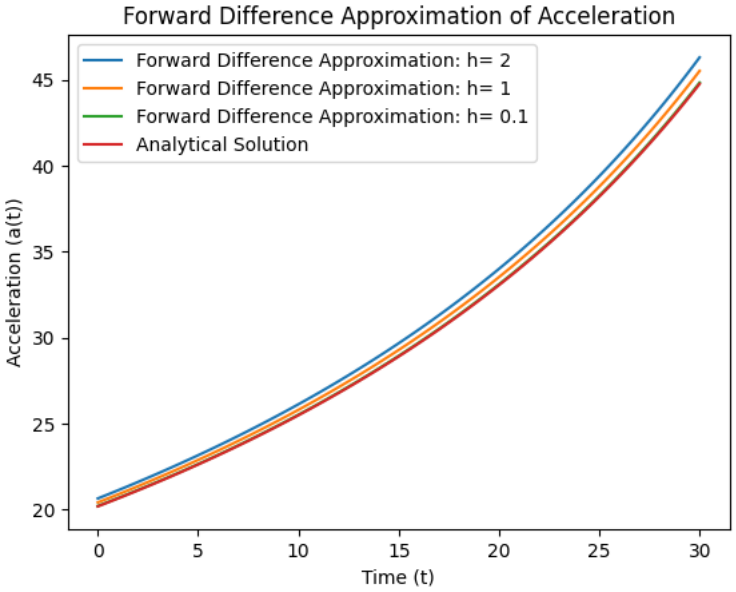
Analytically we can find, $a(t) = v'(t) = a(t) = \frac{42 \times 10^5} {14 \times 10^4 - 2100\, t} - 9.8$

```
# prompt: write a function for the v(t) and a(t) above
import numpy as np

def v(t):
  return 2000 * np.log((14 * 10**4) / (14 * 10**4 - 2100 * t)) - 9.8 * t


def a(t):
  return (42 * 10**5) / (14 * 10**4 - 2100 * t) - 9.8
```

```
# prompt: Plot v(t) and a(t) from t = 0 to 30 side by side

import matplotlib.pyplot as plt
import numpy as np

# Define the time range
t = np.linspace(0, 30, 100)

# Calculate v(t) and a(t)
vt = v(t)
at = a(t)

# Create subplots side by side
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

# Plot v(t)
ax1.plot(t, vt)
ax1.set_xlabel('Time (t)')
ax1.set_ylabel('Velocity (v(t))')
ax1.set_title('Velocity vs. Time')

# Plot a(t)
ax2.plot(t, at)
ax2.set_xlabel('Time (t)')
ax2.set_ylabel('Acceleration (a(t))')
ax2.set_title('Acceleration vs. Time')

# Adjust layout and display the plot
plt.tight_layout()
plt.show()
```




```
def forward_difference(f, x, h):
  return (f(x + h) - f(x)) / h

a_fd_h2 = forward_difference(v, t, h = 2)
a_fd_h1 = forward_difference(v, t, h = 1)
a_fd_h0p1 = forward_difference(v, t, h = 0.1)

# Plotting the results
plt.plot(t, a_fd_h2, label='Forward Difference Approximation: h= 2')
plt.plot(t, a_fd_h1, label='Forward Difference Approximation: h= 1')
plt.plot(t, a_fd_h0p1, label='Forward Difference Approximation: h= 0.1')
plt.plot(t, a(t), label='Analytical Solution')
plt.xlabel('Time (t)')
plt.ylabel('Acceleration (a(t))')
plt.title('Forward Difference Approximation of Acceleration')
plt.legend()
plt.show()
```




## Backward difference

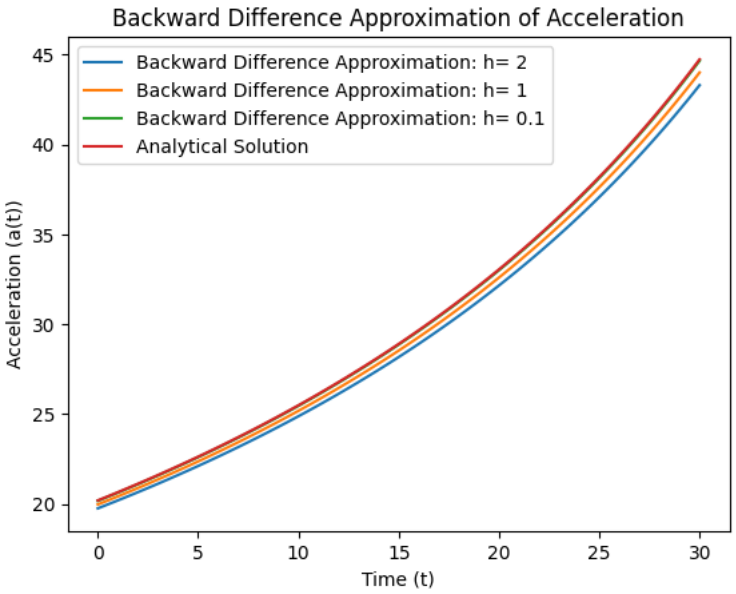
By contrast, the *backward difference* steps backwards. Consider replacing \(h\) with \(-h\),

$$ f(x-h) = f(x) - f'(x) h + O(h^2) $$

and solve for:

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

```
def backward_difference(f, x, h):
  return (f(x)- f(x - h)) / h

a_bd_h2 = backward_difference(v, t, h = 2)
a_bd_h1 = backward_difference(v, t, h = 1)
a_bd_h0p1 = backward_difference(v, t, h = 0.1)

# Plotting the results
plt.plot(t, a_bd_h2, label='Backward Difference Approximation: h= 2')
plt.plot(t, a_bd_h1, label='Backward Difference Approximation: h= 1')
plt.plot(t, a_bd_h0p1, label='Backward Difference Approximation: h= 0.1')
plt.plot(t, a(t), label='Analytical Solution')
plt.xlabel('Time (t)')
plt.ylabel('Acceleration (a(t))')
plt.title('Backward Difference Approximation of Acceleration')
plt.legend()
plt.show()
```




Note that the approximations are approaching the analytic solution from the other direction.

*Can we combine forward and backward differences to get a better result*?

## Central difference

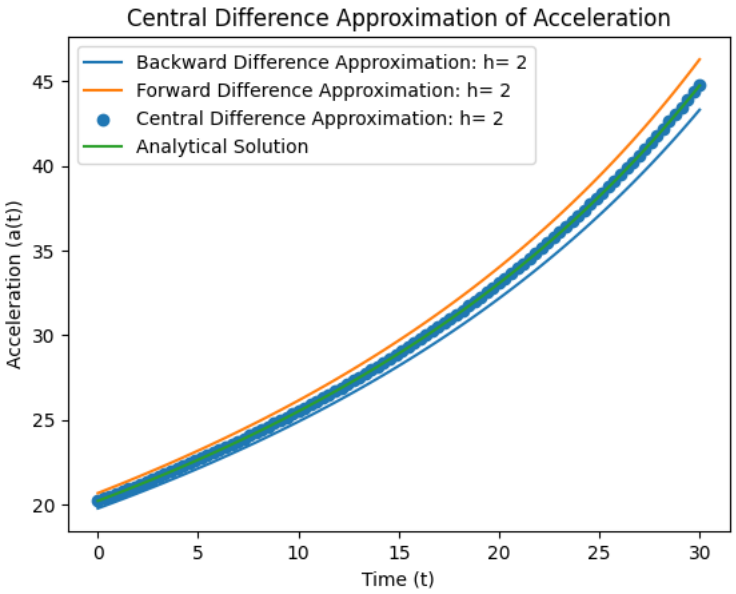
Let's examine the forward and backward expansions:

\[\begin{split} \begin{align} f(x+h) &= f(x) + f'(x) h + f''(x) \frac{h^2}{2} + f'''(x)\frac{h^3}{6} + O(h^4) \\ f(x-h) &= f(x) - f'(x) h + f''(x) \frac{h^2}{2} - f'''(x)\frac{h^3}{6} + O(h^4) \\ \end{align} \end{split}\]
and subtract them: $\( f(x+h)-f(x-h) = 2 f'(x) h + f''' \frac{h^3}{3} + O(h^5) \)\( Note that the even powers of \)h$ cancel!

Rearranging this equation leads to the central difference formula: $\( f'(x) = \frac{f(x+h)-f(x-h)}{2h} + O(h^2) \)$

which is *second order accurate in \(h\)* since the next correction is \(h^2\)!

```
def central_difference(f, x, h):
  return (f(x+h)- f(x - h)) / (2*h)

a_cd_h1 = central_difference(v, t, h = 1)
a_cd_h2 = central_difference(v, t, h = 2)

# Plotting the results
plt.plot(t, a_bd_h2, label='Backward Difference Approximation: h= 2')
plt.plot(t, a_fd_h2, label='Forward Difference Approximation: h= 2')
plt.scatter(t, a_cd_h2, label='Central Difference Approximation: h= 2')
plt.plot(t, a(t), label='Analytical Solution')
plt.xlabel('Time (t)')
plt.ylabel('Acceleration (a(t))')
plt.title('Central Difference Approximation of Acceleration')
plt.legend()
plt.show()
```




## Comparison

Lets comare the error in forward, backward and central difference for \(h=2\):

```
np.stack([a_fd_h2-a(t), a_bd_h2-a(t), a_cd_h2-a(t)]).T[1:5,:]
```

```
array([[ 0.46345415, -0.44519649,  0.00912883],
       [ 0.46776   , -0.44924984,  0.00925508],
       [ 0.47212613, -0.4533588 ,  0.00938366],
       [ 0.47655369, -0.4575244 ,  0.00951465]])
```

Let's see how it scales with decreasing step size between \(h=2\) and \(h' = 1\):

```
np.stack([a_fd_h2-a(t), a_fd_h1-a(t)]).T[1:5,:]
```

```
array([[0.46345415, 0.22936654],
       [0.46776   , 0.23148645],
       [0.47212613, 0.23363589],
       [0.47655369, 0.2358154 ]])
```

```
np.stack([a_bd_h2-a(t), a_bd_h1-a(t)]).T[1:5,:]
```

```
array([[-0.44519649, -0.22480399],
       [-0.44924984, -0.22686082],
       [-0.4533588 , -0.22894601],
       [-0.4575244 , -0.23106008]])
```

Note the error roughly cuts in half; \(Error \sim \frac{h}{h'}\). Now take a look at the central difference:

```
np.stack([a_cd_h2-a(t), a_cd_h1-a(t)]).T[1:5,:]
```

```
array([[0.00912883, 0.00228127],
       [0.00925508, 0.00231281],
       [0.00938366, 0.00234494],
       [0.00951465, 0.00237766]])
```

See how it reduces by a factor of 4: $Error \sim \bigg[\frac{h}{h'}\bigg]^2$

The central difference algorithm:

- finds a more accurate solution in the same number of function calls.
- becomes more accurate with step size quadratically.
- requires information both before and after the point (which can be a problem at boundaries).

## Second order derivatives

We can similarly derive higher order derivatives in forward, backward, and central approximations.

Once obtained, we can also use the second order derivative to improve the first order derivatives!

### Forward difference (second order)

Consider the Taylor expansion: $\( f(x+2h) = f(x)+ 2 f'(x) h + f''(x)\frac{4 h^2}{2} +O(h^3) \)$

and recall:

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

Subtracting twice the second from the first we get:

$$ \begin{align} f(x+2h)-2 f(x+h) &= -f(x) + f''(x) h^2 +O(h^3) \end{align} $$

and we get the forward approximation of the second derivative: $\( f''(x) = \frac{f(x+2h)-2 f(x+h) + f(x)}{h^2} +O(h) \)$

#### Alternative derivation

Note we could also arrive at this considering the 1st derivative of the 1st derivatives: $
\( \begin{align} f'(x) &= \frac{f(x+h) - f(x)}{h} \\ f'(x+h) &= \frac{f(x+2h) - f(x+h)}{h} \\ \end{align} \)\( and therefore, \)
\( \begin{align} f''(x) &= \frac{f'(x+h) - f'(x)}{h} \\ &= \frac{f(x+2h) - 2 f(x+h) + f(x)}{h^2} \end{align} \)$

which lends itself to a recursive program.

#### A more accurate first derivative

We can now revisit the approximation of $f'(x)$ and use our approximation of $f''(x)$ to improve it!

*Be careful to watch the $h$'s in the following*

$$\begin{split} \begin{align} f(x+h) &= f(x) + f'(x) h + f''(x) \frac{h^2}{2} + O(h^3) \\ &= f(x) + f'(x) h + \frac{f(x+2h)-2 f(x+h) + f(x)}{h^2} \frac{h^2}{2}+ O(h^3) \end{align} \end{split}$$

Collecting terms we reach $\( f'(x) = \frac{-f(x+2h) + 4f(x+h) -3 f(x)}{2h} + O(h^2) \)$

This is a nice result, but notice that we now need to do three function calls to achieve the same accuracy as the central difference (2 function calls).

The general formula for $n$th order forward derivatives is: $\(f^n(x) = \sum_{i=0}^n(-1)^{n-i} {n \choose i} f(x+ih) \)$

which one could use to successively improve the previous derivatives.

#### Example: Compare the error for the improved forward difference method

```
# prompt: generate a function for the second order accurate forward difference formula

def forward_difference_2(f, x, h):
  return (-f(x + 2 * h) + 4 * f(x + h) - 3 * f(x)) / (2 * h)

a_fd2_h2 = forward_difference_2(v, t, h = 2)
a_fd2_h1 = forward_difference_2(v, t, h = 1)

np.stack([a_fd_h2-a(t), a_fd2_h2-a(t), a_fd2_h1-a(t), a_cd_h2-a(t)]).T[1:5,:]
```

```
array([[ 0.46345415, -0.01955861, -0.00472106,  0.00912883],
       [ 0.46776   , -0.01983546, -0.00478709,  0.00925508],
       [ 0.47212613, -0.02011757, -0.00485436,  0.00938366],
       [ 0.47655369, -0.02040504, -0.00492289,  0.00951465]])
```

### Backward difference (second order)

Following a similar derivation, the backward second derivative, $\( f''(x) = \frac{f(x)- 2 f(x-h) + f(x-2h)}{h^2} +O(h) \)$ and *second order accurate first derivative*,

$$ f'(x) = \frac{3 f(x) - 4f(x+h) -f(x-2h)}{2h} + O(h^2) $$

and generally, $\(f'(x) = \sum_{i=0}^n(-1)^{i} {n \choose i} f(x-ih) \)$

### Central difference

Finally, subtracting hte forward and backward derivatives gives the central difference:

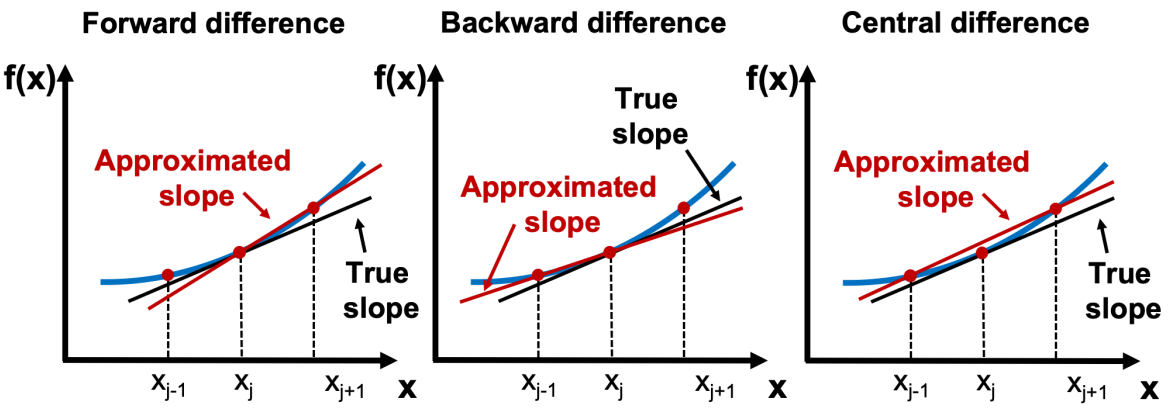
$$ f''(x) = \frac{f(x+h) - 2 f(x) + f(x-h)}{h^2} +O(h^2) $$

and the second order accurate first derivative,

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

and generally, $\(f'(x) = \sum_{i=0}^n(-1)^{i} {n \choose i} f\bigg(x+\bigg[\frac{n}{2}-1\bigg]h\bigg) \)$

### Summary




The accuracy of the approximaiton can be improved in two ways:

1. Decrese step size
2. Use approximated higher order derivatives to *correct* lower order derivatives, but this involves increasing numbers of function calls.

As tempting as 2) is, decresing step size is usually the better approach.

#Discrete data

Numerical methods are often applied to discrete data (or a *discretized function*).

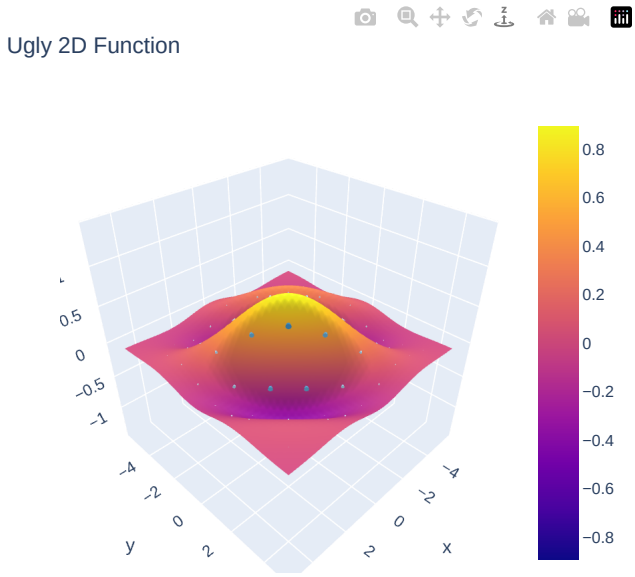
## Higher dimensions

Finite difference generalizes readily to multiple dimensions, by computing the elements of the gradient individually:

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x + he_i) - f(x - he_i)}{2h}$$

Boundary cases are handelled similarly but with the bounded dimension calculated by forward / backward difference.

```
# prompt: Give me an example of an ugly 2D function, plot it as a surface in plotly and show its gradient

import numpy as np
import plotly.graph_objects as go
import plotly.figure_factory as ff

# Define the function
def ugly_function(x, y):
  return (np.sin(x) * np.cos(y) + np.cos(x) * np.sin(y)) * np.exp(-(x**2 + y**2) / 10)

# Create a grid of x and y values
x = np.linspace(-5, 5, 50)
y = np.linspace(-5, 5, 50)
X, Y = np.meshgrid(x, y)

# Calculate the function values and gradients
Z = ugly_function(X, Y)
dZdx, dZdy = np.gradient(Z)

# Create the surface plot
fig = go.Figure(data=[go.Surface(z=Z, x=X, y=Y)])
fig.update_layout(title='Ugly 2D Function', autosize=False,
                  width=500, height=500,
                  margin=dict(l=65, r=50, b=65, t=90))

# Add gradient arrows to the plot
arrow_x = X[::5, ::5].flatten()
arrow_y = Y[::5, ::5].flatten()
arrow_z = Z[::5, ::5].flatten()
arrow_u = dZdx[::5, ::5].flatten()
arrow_v = dZdy[::5, ::5].flatten()


fig.add_trace(go.Cone(
    x=arrow_x,
    y=arrow_y,
    z=arrow_z,
    u=arrow_u,
    v=arrow_v,
    w=np.zeros_like(arrow_u),
    sizemode="absolute",
    sizeref=0.1,
    showscale=False,
    colorscale='Blues'
))

fig.show()
```




# Even vs uneven spacing

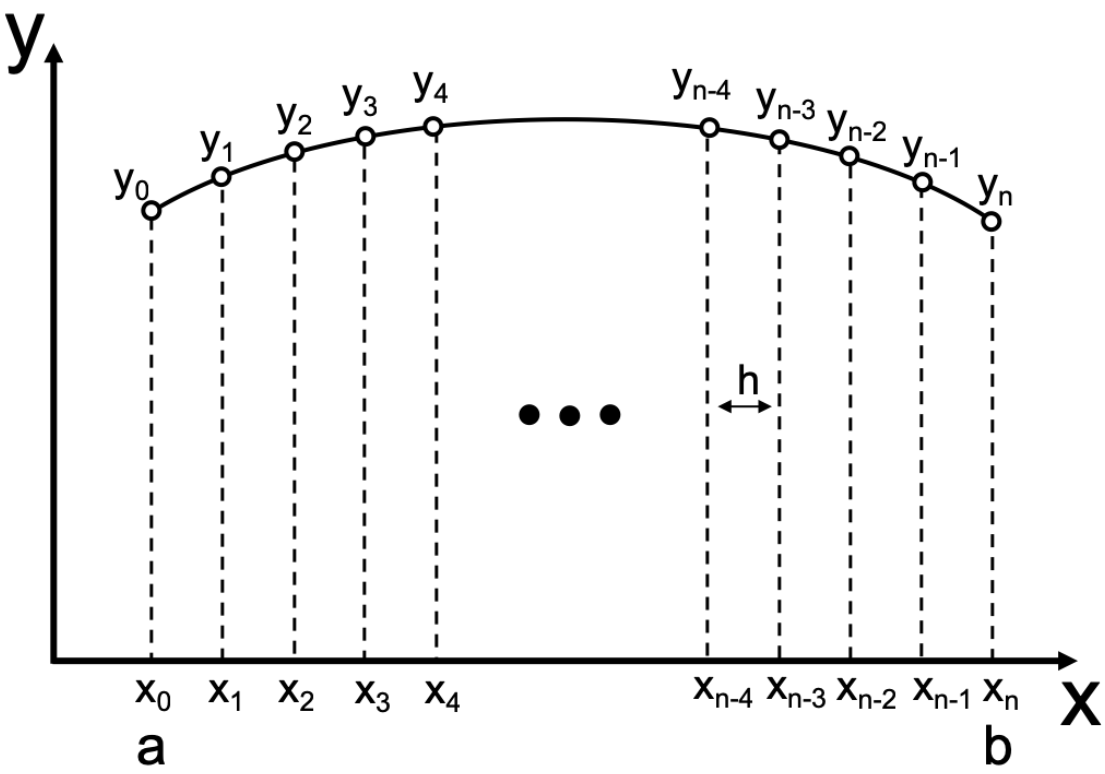
## Evenly spaced data

As long as this data is evenly spaced with step size $h$, we can use finite difference directly.

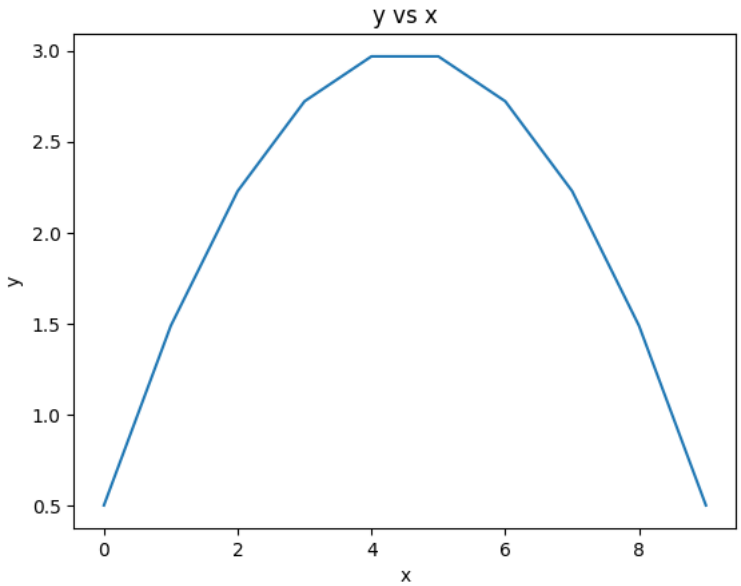
BUT: Be aware of the boundaries! The central difference is the superior choice for the interior points, but you will need forward and backward differences at the boundaries.

#### Example: Find the derivative of a discretized function

Say we have data for a function $y(x)$ that looks like a parabola. We measured it every $h$ steps in the range [a,b], so we have data for $x_0 = a$ to $x_n = b$



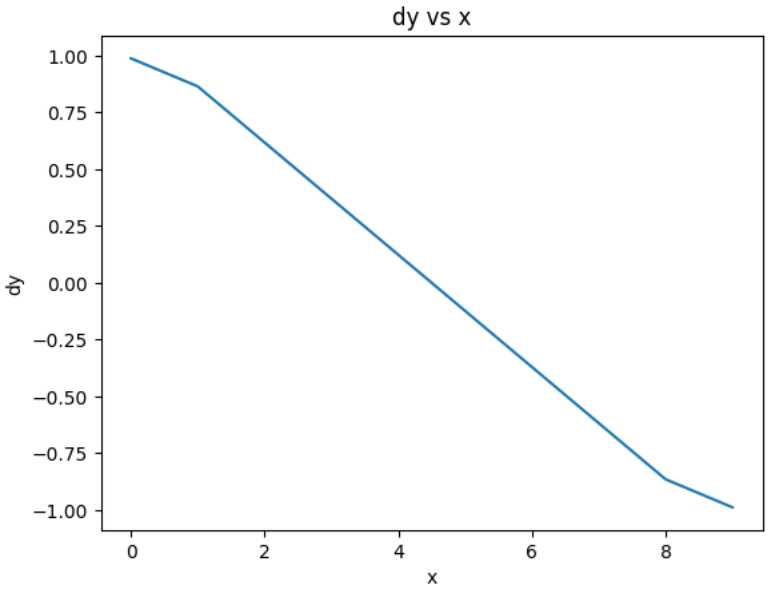

```
x = np.array([0,1,2,3,4,5,6,7,8,9])
y = np.array([0.5, 1.48765432, 2.22839506, 2.72222222, 2.9691358, 2.9691358,
 2.72222222, 2.22839506, 1.48765432, 0.5 ])

plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y')
plt.title('y vs x')
plt.show()
```




If we just applied central difference to this function would encounter problems at the endpoints.

The numpy function *gradient* identifies the endpoints and treats them with forward / backward difference:

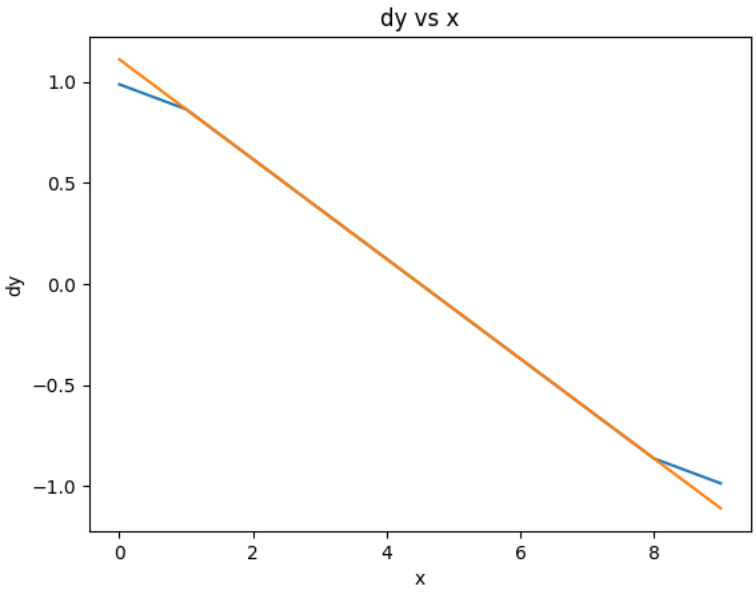
```
dy = np.gradient(y,x)

plt.plot(x, dy)
plt.xlabel('x')
plt.ylabel('dy')
plt.title('dy vs x')
plt.show()
```




This looks good in the middle but the ends are funny…

What do you think?

```
dy2 = np.gradient(y,x, edge_order = 2)

plt.plot(x, dy)
plt.plot(x, dy2)
plt.xlabel('x')
plt.ylabel('dy')
plt.title('dy vs x')
plt.show()
```




Bingo! Using the higher order edge cases solved the problem!

Note, this is the foundation of the finite difference method for solving differential equations.

### Unevenly spaced data

When data is unevenly spaced we can always do first-order forward / backward difference. If we want higher order generally have to resort to a local polynomial fitting.

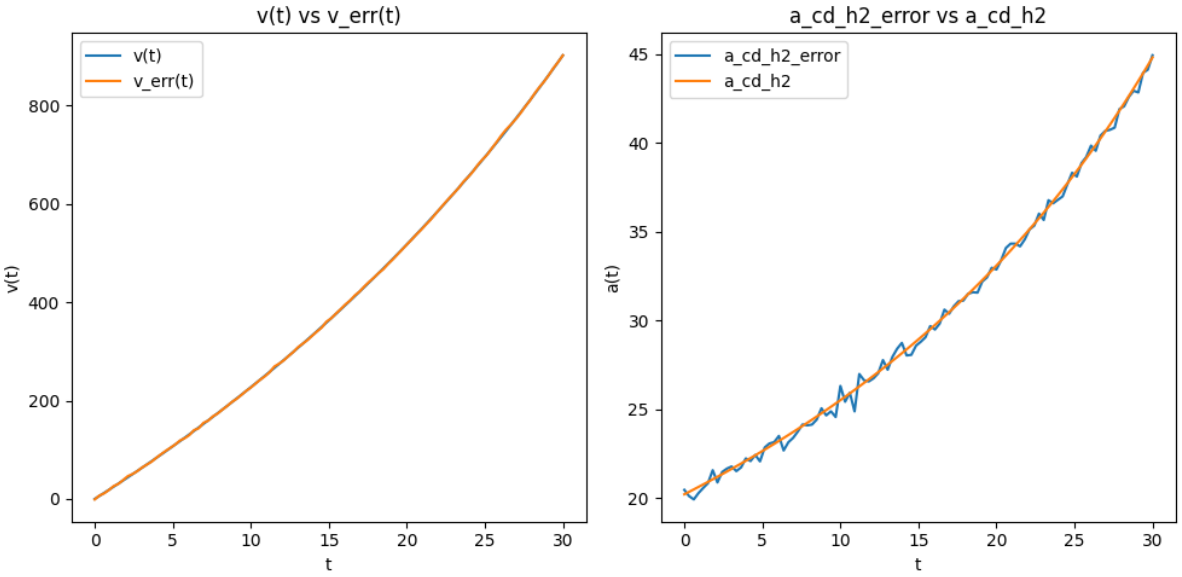
Recall the cubic Lagrange interpolation. We can take the (analytic) derivative and find:

$$ f'(x) = f(x_{i-1}) \frac{2x - x_i-x_{i+1}}{(x_{i-1}-x_i) (x_{i-1}-x_{i+1})} +f(x_{i}) \frac{2x - x_{i-1}-x_{i+1}}{(x_{i}-x_{i-1}) (x_{i}-x_{i+1})} + f(x_{i+1}) \frac{2x - x_{i-1}-x_{i}}{(x_{i+1}-x_{i-1}) (x_{i+1}-x_{i})} $$
which is of comparable accuracy to the centeral difference and recovers it if the points are equally spaced.

## Uncertainty amplification

Let's see what happens when we introduce error into our data.

```
# prompt: Plot beside v and v_err and another plot with a_cd_h2_error and a_cd_h2  beside eachother

def v_err(t):
  return v(t)+np.random.normal(0,1, len(t))

a_cd_h2_error = central_difference(v_err, t, h=2)

# Create subplots
fig, axes = plt.subplots(1, 2, figsize=(10, 5))


# Plot v and v_err
axes[0].plot(t, v(t), label='v(t)')
axes[0].plot(t, v_err(t), label='v_err(t)')
axes[0].set_xlabel('t')
axes[0].set_ylabel('v(t)')
axes[0].set_title('v(t) vs v_err(t)')
axes[0].legend()


# Plot a_cd_h2_error and a_cd_h2
axes[1].plot(t, a_cd_h2_error, label='a_cd_h2_error')
axes[1].plot(t, a_cd_h2, label='a_cd_h2')
axes[1].set_xlabel('t')
axes[1].set_ylabel('a(t)')
axes[1].set_title('a_cd_h2_error vs a_cd_h2')
axes[1].legend()

plt.tight_layout()
plt.show()
```

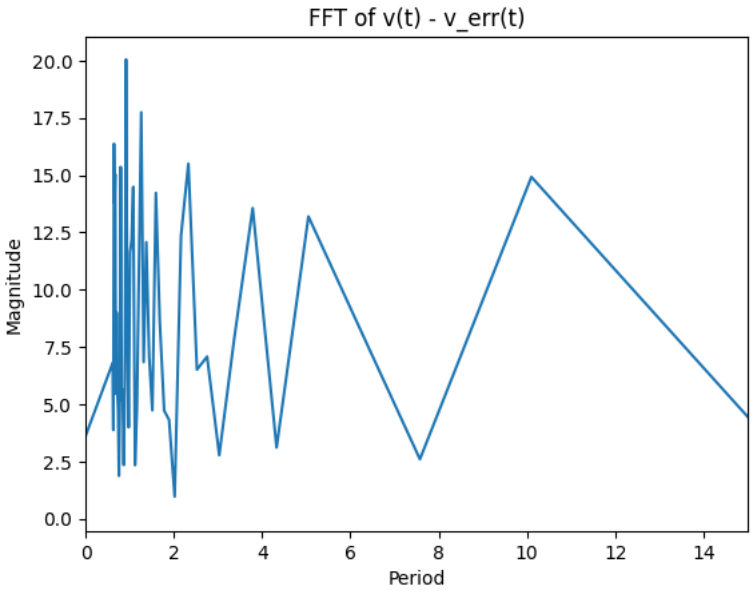



Yikes! This is because differentiation tends to *amplify* error.

Consider a Fourier analysis of the signal and you'll see that the noise has a period on the order of the order of the sampling distance (which is much higher frequency than the signal). This means the noise has a much larger derivative!

```
# prompt: show the fft of v(t)-v_err(t) and show in period

# Calculate the difference between v(t) and v_err(t)
difference = v(t) - v_err(t)

# Perform FFT
fft_result = np.fft.fft(difference)
frequencies = np.fft.fftfreq(len(difference), d=(t[1]-t[0]))

# Plot the FFT in terms of period
plt.plot(1/frequencies, np.abs(fft_result))
plt.xlabel('Period')
plt.ylabel('Magnitude')
plt.title('FFT of v(t) - v_err(t)')
plt.xlim(0, 15)  # Adjust x-axis limits as needed
plt.show()
```

```
<ipython-input-15-8f59ff1ddeb7>:11: RuntimeWarning: divide by zero encountered in divide
  plt.plot(1/frequencies, np.abs(fft_result))
```



With this in mind, there are several options:

- Prefilter the data: Apply a moving average, or low-pass filter to reduce high-frequency noise
- Regularization of the derivative: By requiring smoothness of the derivative, we can *damp* out frenetic behaviour.
- Savitzky-Golay Filter: Use linear least squares to fit a low-degree polynomial to successive windows and take that derivative
- integrate instead.

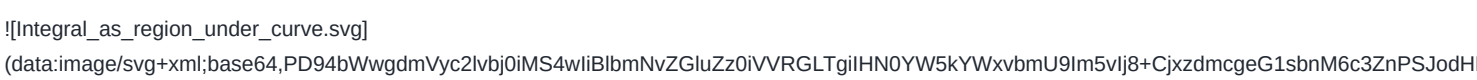
Just as derivation amplifies noise, integration tends to smooth it. So a common approach is to integrate a noisy signal and fit the integral to some expected form.

# Numerical integration

Numerical integration approximates analytic integration, and is particularly useful because:

- Analytic integrals may be hard to find if they exist at all!
- Integration tends to damp experimental noise (in contrast with differentiation which tends to amplify it)

In 1D, integration is simply finding the area under the curve $s = \int_a^b f(x) \ dx$ in the range [a,b]:




For definite integrals (i.e.: with finite limits), numerical integration is called *numerical quadrature*.

> Aside: The integral sign $\int$ looks like an elongated ‘S’ because that ‘summa’ (latin for summation) is exactly what we are doing!

The methods discussed in this section consider 2 cases:

- The function $f(x)$ is available
- The data $f(x), x$ is known at a set of points.

## Newton-Cotes integration formulae

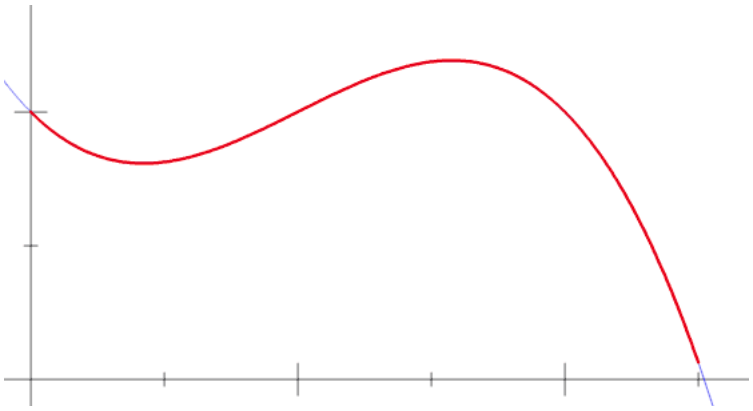
The Newton-Cotes formulae are the most common numerical integration methods. They operate on essentially the same idea:

1. Subdivide the integration domain
2. On each subdivision, approximate the function with a polynomial which can be integrated exactly
3. Sum all the subdivisions

Methods differ in the choice of the polynomial degree and the number of data points needed.

**NOTE** You may wonder why we don’t just fit a high degree polynomial and integrate that. As seen in the Curve Fitting notes, Runge’s Phenomena, which describes the tendency for high order polynomials to exhibit high oscilations torpedos this idea (Note to mention issues with higher dimensionality).

For example: a function being subdivided into a series of increasingly narrower rectangles:



Since we are doing a summation over subdivisions, there is no *obvious* need for consideration of continuity on the edges of our discretization; each block is independant. This implies trivial parallelization:

1. Divide and distribute the integration domain amoung nodes.
2. Each node finds the integral on its node.
3. Sum all nodes.

a scheme which can also be cast as a redundant program.

## Error

It is easy to see that as the step size, $h \rightarrow 0$, the number of boxes, $n\rightarrow \infty$, and the approximation becomes the analytic integral.

Different algorithms will *converge* to the true integral at different rates for decressing step size. As with other numerical methods, the error is characterized by orders of $h$; $O(h^k)$.

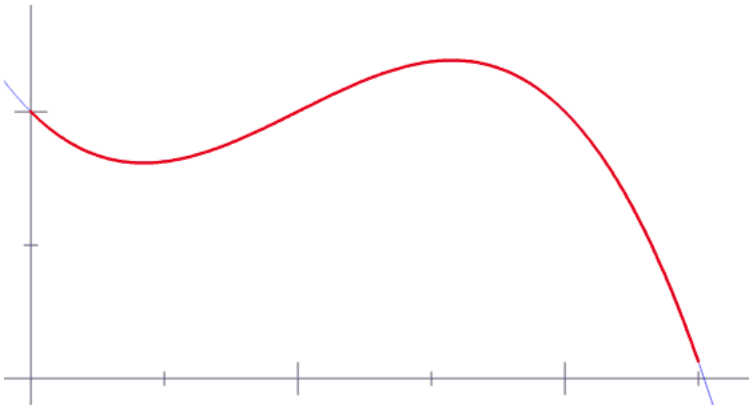
The error now has to be considered on two levels:

- integration on a single subdomain
- summation of subdomains

The approximation would be perfect if either the subdomain integration were perfect, or if each subdomain were infintesiamally small.

Data points are usually *equally spaced* which has benefits for error analysis *on both levels*. I.e.: One could have equally spaced data in a subdomain with a different spacing between subdomains.

For example:



## Use case

These formulae are valid for both the cases where the function is given and only sample points are available. However, these methods are typically only used for the case of discrete (tabulated) data.

## Reimanns integrals (order 0)

Reimann's integral is the simplest of the Newton-Cotes formulae with a polynomial of degree $0$, i.e. a constant over each subdivision. Variants exist for taking the function value at the left limit, middle, or right.

### Left and right Reimann integrals

If we take the function value at the left of the subdomain $[x_i, x_{i+1}]$ , we obtain the *left* Reimann integral,

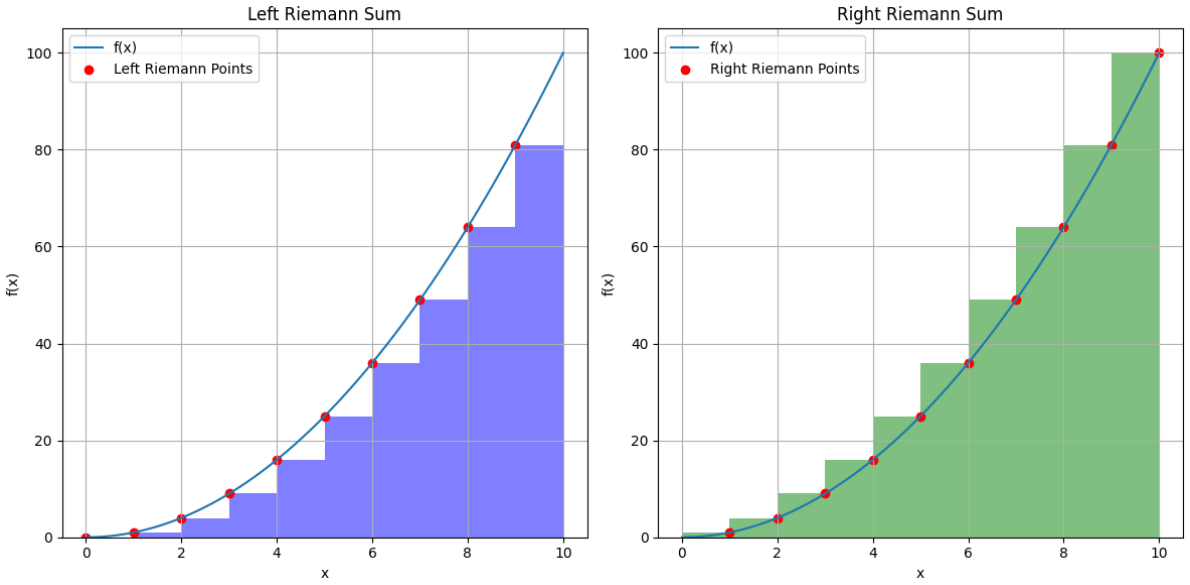
$$\int_a^b f(x) dx \approx \sum_{i = 0}^{n-1} hf(x_i),$$

or using the *right* limit,

$$\int_a^b f(x) dx \approx \sum_{i = 0}^{n-1} hf(x_{i+1}) = \sum_{i = 1}^{n} hf(x_i),$$

These correspond to taking the function values on the left and right of the subdivision as the constant over the entire interval:

```python
# prompt: can you make a figure that shows left, right reimann integrals? from 0 to 10, side by side figu

import numpy as np
import matplotlib.pyplot as plt

def f(x):
  """The function to integrate."""
  return x**2

def left_reimann(f, a, b, n):
  """Calculates the left Riemann sum."""
  dx = (b - a) / n
  x_values = np.linspace(a, b - dx, n)
  return np.sum(f(x_values) * dx)

def right_reimann(f, a, b, n):
  """Calculates the right Riemann sum."""
  dx = (b - a) / n
  x_values = np.linspace(a + dx, b, n)
  return np.sum(f(x_values) * dx)

# Integration interval
a = 0
b = 10
# Number of subdivisions
n = 10

# Calculate left and right Riemann sums
left_sum = left_reimann(f, a, b, n)
right_sum = right_reimann(f, a, b, n)

# Generate x and y values for the function
x_values = np.linspace(a, b, 100)
y_values = f(x_values)

# Create the left Riemann sum figure
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(x_values, y_values, label='f(x)')
dx = (b - a) / n
x_rect = np.linspace(a, b - dx, n)
for i in range(n):
  plt.bar(x_rect[i], f(x_rect[i]), width=dx, alpha=0.5, align='edge', color='blue')
plt.title('Left Riemann Sum')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.scatter(x_rect, f(x_rect), color='red', label='Left Riemann Points')
plt.grid(True)  # Turn on the grid
plt.legend()

# Create the right Riemann sum figure
plt.subplot(1, 2, 2)
plt.plot(x_values, y_values, label='f(x)')
x_rect = np.linspace(a + dx, b, n)
for i in range(n):
  plt.bar(x_rect[i]-dx, f(x_rect[i]), width=dx, alpha=0.5, align='edge', color='green')
plt.title('Right Riemann Sum')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.scatter(x_rect, f(x_rect), color='red', label='Right Riemann Points')
plt.grid(True)  # Turn on the grid
plt.legend()

plt.tight_layout()
plt.show()

print("True integral, 1/3 x^3 = 333. Left integral, ", left_riemann_sum, "Right integral, ", right_riema
```




```
True integral, 1/3 x^3 = 333. Left integral,  285.0 Right integral,  385.0
```

**Error**

To analyse the error, consider integrating the Taylor expansion of the left integral (the right being trivially similar),

\[\begin{split} \begin{align} f(x) &= f(x_i) + f^{\prime}(x_i)(x-x_i) + \cdots \\ \int_{x_i}^{x_{i+1}} f(x) dx &= \int_{x_i}^{x_{i+1}} (f(x_i) + f^{\prime}(x_i)(x-x_i) + \cdots)\ dx \\ &= \int_{x_i}^{x_{i+1}} f(x_i) dx + \int_{x_i}^{x_{i+1}} f^{\prime}(x_i)(x-x_i)dx + \cdots.\ \end{align}\end{split}\]
since the integral distributes. Integrating term by term we get,

\[\begin{split} \begin{align} \int_{x_i}^{x_{i+1}} f(x) dx &= hf(x_i) + \frac{h^2}{2}f^{\prime}(x_i) + O(h^3), \\ &= hf(x_i) + O(h^2)\end{align} \end{split}\]
For each subinterval, the left integral is \(O(h^2)\).

If we sum the \(O(h^2)\) error over the entire Riemann sum, we get \(nO(h^2)\). The relationship between \(n\) and \(h\) is

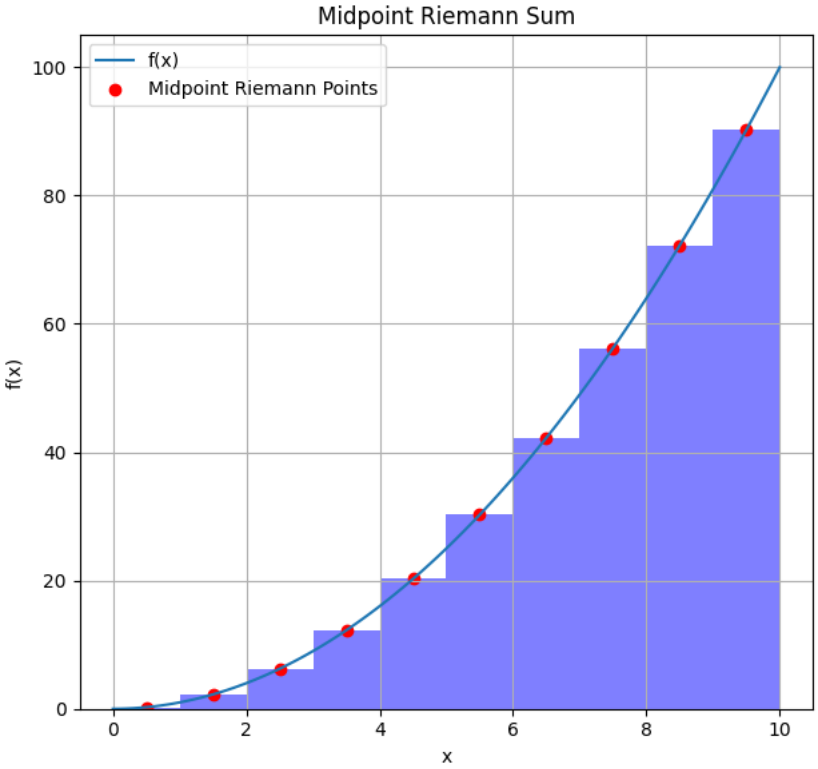
\[h = \frac{b - a}{n},\]
and so our total error becomes \(\frac{b - a}{h}O(h^2) = O(h)\) over the whole interval. Thus the overall accuracy is \(O(h)\).

**The Midpoint Reimann integral**

Rather than favour the left or right subdomain limit, let's take hte midpoint \(y_i = \frac{x_{i+1} + x_i}{2}\). The Midpoint Rule says

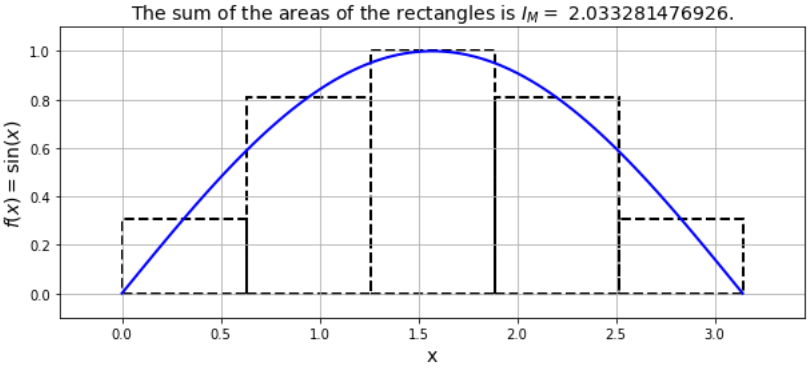
\[\int_a^b f(x)dx \approx \sum_{i = 0}^{n-1} hf(y_i).\]

```python
# prompt: Repeat the code above with the midpoint rule

import numpy as np
import matplotlib.pyplot as plt

def f(x):
  """The function to integrate."""
  return x**2

def midpoint_reimann(f, a, b, n):
  """Calculates the midpoint Riemann sum."""
  dx = (b - a) / n
  x_values = np.linspace(a + dx / 2, b - dx / 2, n)
  return np.sum(f(x_values) * dx)


# Integration interval
a = 0
b = 10
# Number of subdivisions
n = 10

# Calculate left, right, and midpoint Riemann sums
midpoint_sum = midpoint_reimann(f, a, b, n)

# Generate x and y values for the function
x_values = np.linspace(a, b, 100)
y_values = f(x_values)

# Create the midpoint Riemann sum figure
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(x_values, y_values, label='f(x)')
dx = (b - a) / n
x_rect = np.linspace(a + dx / 2, b - dx / 2, n)
for i in range(n):
  plt.bar(x_rect[i], f(x_rect[i]), width=dx, alpha=0.5, align='center', color='blue')
plt.title('Midpoint Riemann Sum')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.scatter(x_rect, f(x_rect), color='red', label='Midpoint Riemann Points')
plt.grid(True)  # Turn on the grid
plt.legend()

plt.tight_layout()
plt.show()


print("True integral, 1/3 x^3 = 333. Left integral, ", left_sum, "Right integral, ", right_sum, "Midpoin
```




```
True integral, 1/3 x^3 = 333. Left integral,  285.0 Right integral,  385.0 Midpoint integral,  332.5
```

Another example, approximate \(\int_0^\pi \sin(x) = 2\)




However, if you have discrete data in the range \([a,b]\) you will not be able to stricly calculate the integral since you would overlap on either side…

**Error**

The error can be deduced considering the Taylor series of \(f(x)\) around \(y_i\), which as before becomes,

\[\begin{split} \begin{align} f(x) &= f(y_i) + f^{\prime}(y_i)(x - y_i) + \frac{f''(y_i)(x - y_i)^2}{2!} + \cdots \\ \int_{x_i}^{x_{i+1}} f(x) dx &= \int_{x_i}^{x_{i+1}} \left(f(y_i) + f^{\prime}(y_i)(x - y_i) + \frac{f''(y_i)(x - y_i)^2}{2!} + \cdots\right) dx, \\ &= \int_{x_i}^{x_{i+1}} f(y_i)dx + \int_{x_i}^{x_{i+1}} f^{\prime}(y_i)(x - y_i)dx + \int_{x_i}^{x_{i+1}} \frac{f''(y_i)(x - y_i)^2}{2!}dx + \cdots. \end{align}\end{split}\]
but now there is a trick! Since \(x_i\) and \(x_{i+1}\) are symmetric about \(y_i\), all odd derivatives integrate to zero; e.g. \(\int_{x_i}^{x_{i+1}} f^{\prime}(y_i)(x - y_i)dx = 0\)

Therefore the midpoint rule becomes: $\(\int_{x_i}^{x_{i+1}} f(x) dx= hf(y_i) + O(h^3)\)\( which has \)O(h^3)
\( accuracy for one subinterval or \)O(h^2)$ over the whole interval.

## The Trapezoid rule (order 1)

The next polynomial degree is $1$; a line.

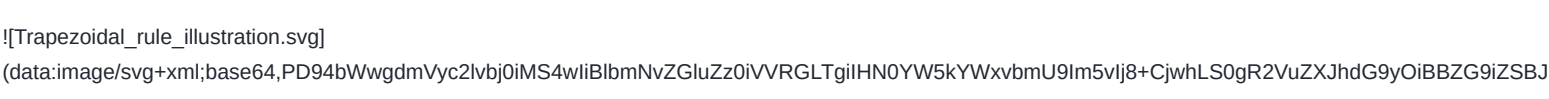
Given the points $x_i$, $x_{i+1}$, $f_i$, and $f_{i+1}$,

$$ f(x) = f_i + \frac{f_{i+1} - f_i}{x_{i+1} - x_i} (x - x_i) $$

and integrating,

$$\begin{split} \begin{align} \int_{x_i}^{x_{i+1}} f(x) \, dx &= \int_{x_i}^{x_{i+1}} \left( f_i + \frac{f_{i+1} - f_i}{x_{i+1} - x_i} (x - x_i) \right) dx \\ &= \frac{f_i + f_{i+1}}{2} (x_{i+1} - x_i) \\ &= h \frac{f_i + f_{i+1}}{2} \end{align} \end{split}$$

Graphically:




**Error**

The accuracy is calculated from the Taylor series expansion of $f(x)$ around the midpoint $y_i = \frac{x_{i+1} + x_i}{2}$, which is the midpoint between $x_i$ and $x_{i+1}$.

Calculating the error is a little more complicated since we'll need the function value at the midpoint. Begin with the Taylor series at $x_i$ and $x_{i+1}$, noting that $x_i - y_i = -\frac{h}{2}$ and $x_{i+1} - y_i = \frac{h}{2}$:

$$\begin{split} \begin{align} f(x_i) &= f(y_i) - \frac{hf^{\prime}(y_i)}{2} + \frac{h^2f''(y_i)}{8} - \cdots \\ f(x_{i+1}) &= f(y_i) + \frac{hf^{\prime}(y_i)}{2} + \frac{h^2f''(y_i)}{8} + \cdots. \end{align} \end{split}$$

Taking the average $\frac{f(x_{i+1})+f(x_i)}{2} = f(y_i) + O(h^2).$ cancels odd derivatives, and we can now find, $f(y_i) = \frac{f(x_{i+1})+f(x_i)}{2} + O(h^2).$

Now we take the Taylor series expanded about $y_i$ and integrate / distribute as before:

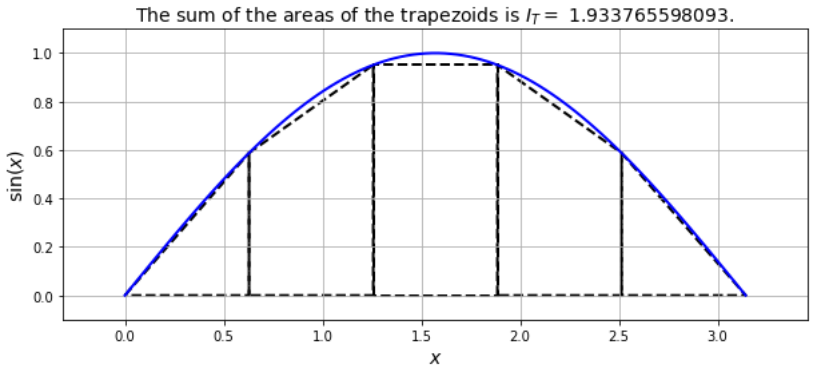
$$\begin{split} \begin{align} f(x) &= f(y_i) + f^{\prime}(y_i)(x - y_i) + \frac{f''(y_i)(x - y_i)^2}{2!} + \cdots \\ \int_{x_i}^{x_{i+1}} f(x) dx &= \int_{x_i}^{x_{i+1}} \left(f(y_i) + f^{\prime}(y_i)(x - y_i) + \frac{f''(y_i)(x - y_i)^2}{2!} + \cdots\right) dx \\ &= \int_{x_i}^{x_{i+1}} f(y_i) dx + \int_{x_i}^{x_{i+1}} f^{\prime}(y_i)(x - y_i)dx + \int_{x_i}^{x_{i+1}} \frac{f''(y_i)(x - y_i)^2}{2!} dx + \cdots \end{align} \end{split}$$

Once again, since $x_i$ and $x_{i+1}$ are symmetric around $y_i$, odd integrals evaluate to zero. Now we can insert our result from above,

$$\begin{split} \begin{align} \int_{x_i}^{x_{i+1}} f(x) dx &= hf(y_i) + O(h^3) \\ & = h \left(\frac{f(x_{i+1})+f(x_i)}{2} + O(h^2)\right) + O(h^3) \\ &= h \left(\frac{f(x_{i+1})+f(x_i)}{2}\right) + hO(h^2) + O(h^3) \\ &= h \left(\frac{f(x_{i+1})+f(x_i)}{2}\right) + O(h^3) \end{align} \end{split}$$

Therefore the trapezoid rule is is $O(h^3)$ over subintervals and totals $O(h^2)$ over the whole interval.

For the same function as the midpoint method (which approximated 2.033), the trapezoid method yields,



**Example**

Integrate \(sin(x)\) from 0 to \(\pi\) with the midpoint and trapezoid rules.

```
# prompt: integrate sin(x) from 0 to pi using the trapezoid and midpoint methods number of steps from 2 t

import numpy as np

def f(x):
  """The function to integrate."""
  return np.sin(x)

def trapezoid_method(f, a, b, n):
  """Approximates the integral of f from a to b using the trapezoid method."""
  h = (b - a) / n
  #####
  # What is the sampling space?

  #
  #####
  x = np.linspace(a, b, n + 1)
  #####

  integral = np.trapz(f(x), x, h)
  return integral

def midpoint_method(f, a, b, n):
  """Approximates the integral of f from a to b using the midpoint method."""
  h = (b - a) / n
  #####
  # What is the sampling space?

  #
  #####
  x = np.linspace(a + h / 2, b - h / 2, n)
  #####
  integral = h * np.sum(f(x))
  return integral

a = 0
b = np.pi
for n in [2,4,8,16,32]:
  trapezoid_result = trapezoid_method(f, a, b, n)
  midpoint_result = midpoint_method(f, a, b, n)
  print(f"n = {n}: Trapezoid error = {abs(trapezoid_result-2):.6f}, Midpoint error= {midpoint_result-2:.(
```

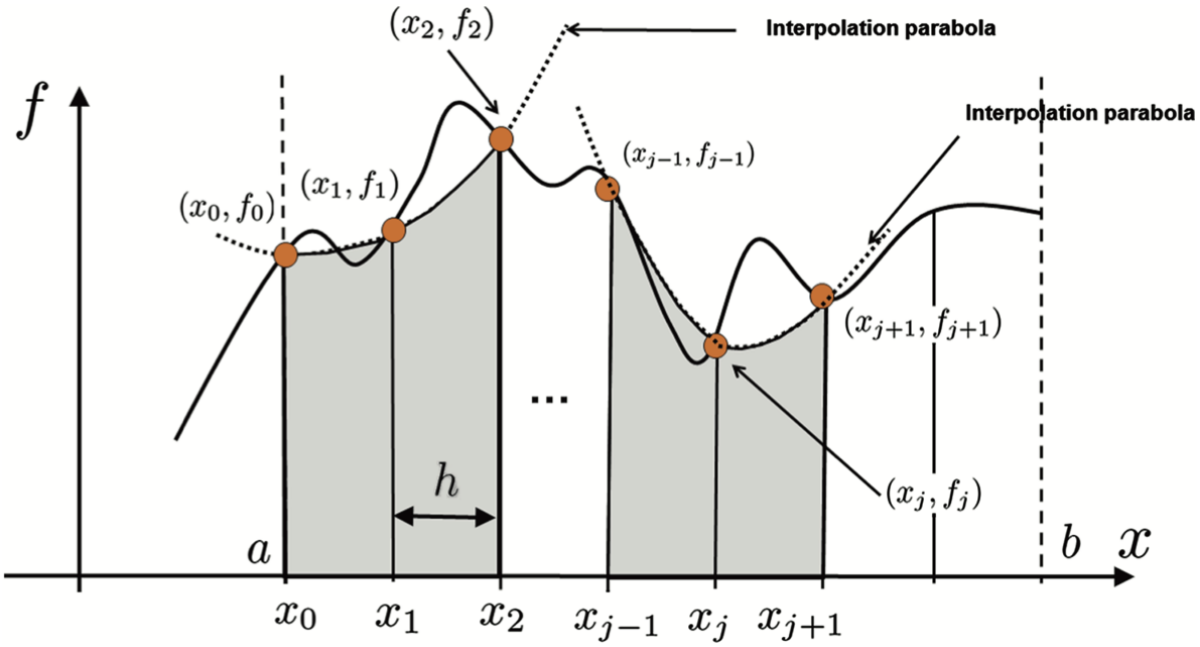
```
n = 2: Trapezoid error = 0.429204, Midpoint error= 0.221441
n = 4: Trapezoid error = 0.103881, Midpoint error= 0.052344
n = 8: Trapezoid error = 0.025768, Midpoint error= 0.012909
n = 16: Trapezoid error = 0.006430, Midpoint error= 0.003216
n = 32: Trapezoid error = 0.001607, Midpoint error= 0.000803
```

Notice we get quadratic convergence for both but that the midpoint method is actually more accurate!

## Simpson's 1/3 rule (order 2)

Continuing our polynomial degrees, we arrive at a quadratic (parabola). Now we need 3 data points per fit, which we can envision as 2 subdivisions back to back.

NOTE: Considering a pair of subintervals is just a conceptualization. In truth, we need to have an *even* number of data points to use this method. Each pair is separate and we are *not* imposing any kind of continuity or overlap between subsequent pairs.




We can fit a quadratic with Lagrange polynomials with \(x_i-x_{i-1} = x_{i+1}-x_{i} = h\):

\[f(x) \approx \frac{f(x_{i-1})}{2h^2} (x - x_i)(x - x_{i+1}) - \frac{f(x_i)}{h^2} (x - x_{i-1})(x - x_{i+1}) + \frac{f(x_{i+1})}{2h^2} (x - x_{i-1})(x - x_{i}).\]
and $\(\int_{x_{i-1}}^{x_{i+1}} f(x) dx = \frac{h}{3}(f(x_{i-1}) + 4f(x_i) + f(x_{i+1}).\)$

The \(\frac{1}{3}\) gives the method its name.

**Error analysis**

For the error, take the Taylor series approximation of \(f(x)\) around \(x_i\),

\[f(x) = f(x_i) + f^{\prime}(x_i)(x - x_i) + \frac{f''(x_i)(x-x_i)^2}{2!} + \frac{f'''(x_i)(x-x_i)^3}{3!} + \frac{f''''(x_i)(x-x_i)^4}{4!} + \cdots\]
and evaluate at \(x_{i-1}\) and \(x_{i+1}\), substituting for \(h\) where appropriate,

\[\begin{split} \begin{align} f(x_{i-1}) &= f(x_i) - hf^{\prime}(x_i) + \frac{h^2f''(x_i)}{2!} - \frac{h^3f'''(x_i)}{3!} + \frac{h^4f''''(x_i)}{4!} - \cdots \\ f(x_{i+1}) &= f(x_i) + hf^{\prime}(x_i) + \frac{h''(x_i)}{2!} + \frac{h^3f'''(x_i)}{3!} + \frac{h^4f''''(x_i)}{4!} + \cdots \end{align} \end{split}\]
Now consider \(\frac{f(x_{i-1}) + 4f(x_i) + f(x_{i+1})}{6}\) with the expansions above, $
\(\frac{f(x_{i-1}) + 4f(x_i) + f(x_{i+1})}{6} = f(x_i) + \frac{h^2}{6}f''(x_i) + \frac{h^4}{72}f''''(x_i) + \cdots\)
\( note that the odd terms cancel. Rearrange to find, \)
\(f(x_i) =\frac{f(x_{i-1}) + 4f(x_i) + f(x_{i+1})}{6} - \frac{h^2}{6}f''(x_i) + O(h^4).\)$

The integral of \(f(x)\) over two subintervals is,

\begin{eqnarray*}\int_{x_{i-1}}^{x_{i+1}} f(x) dx &=& \int_{x_{i-1}}^{x_{i+1}} \left(f(x_i) + f^{\prime}(x_i)(x - x_i) + \frac{f''(x_i)(x-x_i)^2}{2!}\right.\ &&\qquad\qquad\left. + \frac{f'''(x_i)(x-x_i)^3}{3!}+ \frac{f''''(x_i)(x-x_i)^4}{4!} + \cdots\right) dx.\end{eqnarray*}

Distributing the integral and dropping odd derivatives which are zero from symmetry,

\[\begin{split} \begin{align} \int_{x_{i-1}}^{x_{i+1}} f(x) dx &= \int_{x_{i-1}}^{x_{i+1}} f(x_i) dx + \int_{x_{i-1}}^{x_{i+1}}\frac{f''(x_i)(x-x_i)^2}{2!}dx + \int_{x_{i-1}}^{x_{i+1}}\frac{f''''(x_i)(x-x_i)^4}{4!}dx + \cdots \\ &= 2h f(x_i) + \frac{h^3}{3}f''(x_i) + O(h^5) \\ &= 2h \left(\frac{f(x_{i-1}) + 4f(x_i) + f(x_{i+1})}{6} - \frac{h^2}{6}f''(x_i) + O(h^4)\right) + \frac{h^3}{3}f''(x_i) + O(h^5) \\ &= \frac{h}{3}(f(x_{i-1}) + 4f(x_i) + f(x_{i+1})) + O(h^5) \end{align} \end{split}\]
having used the previously derived result.

This equation implies that Simpson's Rule is \(O(h^5)\) over a subinterval and \(O(h^4)\) over the whole interval. Because the \(h^3\) terms cancel out exactly, Simpson's Rule gains another *two* orders of accuracy!

**Example**

Integrate the sin(x) in \([0,\pi]\) with Simpson's 1/3 Rule and compare to midpoint and trapezoid

```
import scipy as sp

def f(x):
  """The function to integrate."""
  return np.sin(x)

def trapezoid_method(f, a, b, n):
  """Approximates the integral of f from a to b using the trapezoid method."""
  h = (b - a) / n
  x = np.linspace(a, b, n + 1)
  integral = np.trapz(f(x), x, h)
  return integral

def midpoint_method(f, a, b, n):
  """Approximates the integral of f from a to b using the midpoint method."""
  h = (b - a) / n
  x = np.linspace(a + h / 2, b - h / 2, n)
  integral = h * np.sum(f(x))
  return integral

def simps_method(f, a, b, n):
  """Approximates the integral of f from a to b using the simpsons method."""
  h = (b - a) / n
  x = np.linspace(a, b, n + 1)
  integral = sp.integrate.simpson(f(x), x = x, dx = h)
  return integral

a = 0
b = np.pi
for n in [2,4,8,16,32]:
  trapezoid_result = trapezoid_method(f, a, b, n)
  midpoint_result = midpoint_method(f, a, b, n)
  simps_result = simps_method(f, a, b, n)
  print(f"n = {n}: Trapezoid error = {abs(trapezoid_result-2):.6f}, Midpoint error= {midpoint_result-2:.(
```

```
n = 2: Trapezoid error = 0.429204, Midpoint error= 0.221441, Simpson's error= 0.094395
n = 4: Trapezoid error = 0.103881, Midpoint error= 0.052344, Simpson's error= 0.004560
n = 8: Trapezoid error = 0.025768, Midpoint error= 0.012909, Simpson's error= 0.000269
n = 16: Trapezoid error = 0.006430, Midpoint error= 0.003216, Simpson's error= 0.000017
n = 32: Trapezoid error = 0.001607, Midpoint error= 0.000803, Simpson's error= 0.000001
```

Yowza!

## Simpson's 3/8 rule (order 3)

The final polynomial we will discuss is a cubic fit to 4 data points (3 intervals). This results in

$$\int_{x_{i-1}}^{x_{i+1}} f(x)\,dx = \frac{3}{8}h(f(x_{i}) + 3f(x_{i+1}) + 3f(x_{i+2} + f(x_{i+3})) + O(h^5)$$

Note the \(3/8\) which lends its name. This function takes 3 intervals, and therefore a combination of the \(1/3\)rd and \(3/8\)th rules cover all possabilities without loss of error order.

Note that the order of error is the same as the \(1/3\) rule. In fact, this is generally true for higher order Newton-Cotes methods generally, and therefore usually the \(1/3\) and \(3/8\) rule are employed.

## Uneven data

Having the same step size within a subinterval has clear benefits for convergence due to the cancellation of higher order terms. However, the error of the total integral is determined by the largest of the subdomains. Therefore, it is optimimal to have equal spacing in and between the subdomains.

For uneven data, we could define subdivisions based on where the step size is equal, thereby making best use of our data. In the case where the step size is completely random, we can resort to the trapezoid rule.

### Multiple dimensions

Multiple integrals can be decomposed into a series of 1D integrals due to the properties of integration:

$$\int \int_R f(x, y) dA = \int \bigg[\int f(x, y) dy \bigg] dx \]$$

This defines a recursive algorithm for calculating a series of 1D integrals.

### Summary

| Rule | Subdomain Formula | Subdomain error | Total Integral Formula | Total error |
|---|---|---|---|---|
| Midpoint | $I_i = hf(x_i)$ | $O(h^3)$ | $I = h\sum_{i=1}^n f(x_i)$ | $O(h^2)$ |
| Trapezoid | $I_i = \frac{h}{2}[f(x_{i-1}) + f(x_i)]$ | $O(h^3)$ | $I = \frac{h}{2}[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)]$ | $O(h^2)$ |
| Simpson's 1/3 | $I_i = \frac{h}{3}[f(x_{i-1}) + 4f(x_i) + f(x_{i+1})]$ | $O(h^5)$ | $I = \frac{h}{3}[f(x_0) + 4\sum_{i=1,3,5}^{n-1} f(x_i) + 2\sum_{i=2,4,6}^{n-2} f(x_i) + f(x_n)]$ | $O(h^4)$ |
| Simpson's 3/8 | $I_i = \frac{3h}{8}[f(x_{i-1}) + 3f(x_i) + 3f(x_{i+1}) + f(x_{i+2})]$ | $O(h^5)$ | $I = \frac{3h}{8}[f(x_0) + 3\sum_{i=1,4,7}^{n-2} f(x_i) + 3\sum_{i=2,5,8}^{n-1} f(x_i) + 2\sum_{i=3,6,9}^{n-3} f(x_i) + f(x_n)]$ | $O(h^4)$ |

## Romberg rule

### Richardson extrapolation

Richardson extrapolation is an algorithm that conceptually uses successive applications of the trapezoid rule with differing step size to achieve superior results with less effort.

The exact integral can always be expressed: $$ I = I'(h) + E(h)$$ where $I'(h)$ is the approximation with step $h$ and the associated error $E(h)$. We know that $E(h) \propto h^2$ for the trapezoid rule. In fact, it is $E(h) \propto f'' h^2$.

Let's sample the interval twice with step sizes $h_1$ and $h_2$. If we assume $f''$ doesn't change much we can say,

$$ \begin{align} \frac{E(h_1)}{E(h_2)} = \frac{h_1^2}{h_2^2}. \end{align}\]

Now, since the exact integral is the same in both cases, $$
\begin{align} I'(h_1) + E(h_1) &= I'(h_2) + E(h_2) \\ I'(h_1) + E(h_2) \bigg(\frac{h_1}{h_2}\bigg)^2 &= I'(h_2) + E(h_2) \\ E(h_2) &=\frac{I'(h_1) -I'(h_2)}{1-(\frac{h_1}{h_2})^2} \\ \end{align} $$

and inserted into the formula for $h_2$,

$$\begin{split} \begin{align} I&\approx I(h_2)+ \frac{I'(h_1) -I'(h_2)}{1-(\frac{h_1}{h_2})^2} \\ \end{align} \end{split}\]

which can be shown to be accurate to $O(h^4)$!

For the special case where $h_1 = 2 h_2$ (which has advantages for overlapping point evaluations)

$$ I \approx \frac{4}{3} I(h_2) -\frac{1}{3} I(h_1) \]$$

This is an interesting result! Effectively what we have done is use a second estimate to estimate the next power in our expansion, leading to a higher order estimate!

## Romberg Integration Algorithm

In fact we can repeat this proceedure arbitrarily! Above we combined two order $O(h^2)$ to make $O(h^4)$. We can take this results, combine it with another sampling at $h_3 < h_2$ and combine to get an $O(h^6)$ estimate and so on! If we successively halve the step size, we can get:

$$\begin{split}\begin{align} I^4 &\approx \frac{4}{3} I(h_2) -\frac{1}{3} I(h_1) \\ I^6 &\approx \frac{16}{15} I(h_3) - \frac{1}{15} I^4 \\ I^8 &\approx \frac{64}{63} I(h_4) - \frac{1}{63} I^6 \\ \vdots \\ I_{j,k} &\approx \frac{4^{k-1} I_{j+1,k-1} - I_{j,k-1}}{4^{k-1}-1} \end{align}\end{split}$$

where the last line is the Romberg Integration Algorithm. The structure lends itself to redundant programming and parallelizes nicely!

This algorithm is able to integrate to an arbitrary accuracy and does so remarkably efficiently compared to the alternatives.

### Example

Approximate $\int_0^\pi \sin(x) dx$ using the Rhomberg rule and compare with Simpson's 1/3 rule

```
import numpy as np
import scipy as sp
def f(x):
  return np.sin(x)

tolerance = 1e-6

for n in [4,8,16,32,64, 128]:
  x = np.linspace(0, np.pi, n+1)
  f_x = f(x)
  rhomberg = sp.integrate.romb(f_x, dx = np.pi/n, show=True)
  print(f"Romberg with {n} intervals: {abs(rhomberg-2)}")
  simpson = sp.integrate.simpson(f_x, x=x)
  print(f"Simpson with {n} intervals: {abs(simpson-2)}")
```

```
Richardson Extrapolation Table for Romberg Integration
=======================================================
 0.00000
 1.57080  2.09440
 1.89612  2.00456  1.99857
=======================================================
Romberg with 4 intervals: 0.001429268176164289
Simpson with 4 intervals: 0.0045597549844207386
Richardson Extrapolation Table for Romberg Integration
=======================================================
 0.00000
 1.57080  2.09440
 1.89612  2.00456  1.99857
 1.97423  2.00027  1.99998  2.00001
=======================================================
Romberg with 8 intervals: 5.549979670949057e-06
Simpson with 8 intervals: 0.00026916994838765973
Richardson Extrapolation Table for Romberg Integration
=======================================================
 0.00000
 1.57080  2.09440
 1.89612  2.00456  1.99857
 1.97423  2.00027  1.99998  2.00001
 1.99357  2.00002  2.00000  2.00000  2.00000
=======================================================
Romberg with 16 intervals: 5.412709835894702e-09
Simpson with 16 intervals: 1.6591047935499148e-05
Richardson Extrapolation Table for Romberg Integration
=======================================================
 0.00000
 1.57080  2.09440
 1.89612  2.00456  1.99857
 1.97423  2.00027  1.99998  2.00001
 1.99357  2.00002  2.00000  2.00000  2.00000
 1.99839  2.00000  2.00000  2.00000  2.00000  2.00000
=======================================================
Romberg with 32 intervals: 1.3216094885137863e-12
Simpson with 32 intervals: 1.0333694131503535e-06
Richardson Extrapolation Table for Romberg Integration
=======================================================
 0.00000
 1.57080  2.09440
 1.89612  2.00456  1.99857
 1.97423  2.00027  1.99998  2.00001
 1.99357  2.00002  2.00000  2.00000  2.00000
 1.99839  2.00000  2.00000  2.00000  2.00000  2.00000
 1.99960  2.00000  2.00000  2.00000  2.00000  2.00000  2.00000
=======================================================
Romberg with 64 intervals: 4.440892098500626e-16
Simpson with 64 intervals: 6.453000178652246e-08
Richardson Extrapolation Table for Romberg Integration
=======================================================
 0.00000
 1.57080  2.09440
 1.89612  2.00456  1.99857
 1.97423  2.00027  1.99998  2.00001
 1.99357  2.00002  2.00000  2.00000  2.00000
 1.99839  2.00000  2.00000  2.00000  2.00000  2.00000
 1.99960  2.00000  2.00000  2.00000  2.00000  2.00000  2.00000
 1.99990  2.00000  2.00000  2.00000  2.00000  2.00000  2.00000  2.00000
=======================================================
Romberg with 128 intervals: 0.0
Simpson with 128 intervals: 4.032257194808153e-09
```

Note the error became zero! What does that mean?

## Gaussian quadrature

Until this point, we have been using techniques for which the data points are at the limits of our subdomains. Let's relax this condition and allow the evaluation points to move around in the subdomain.

As we saw previously, we can gain in accuracy by cancelling positive and negative errors. Gaussian quadrature seeks to find special points in the subdomain where errors cancel.

### The method of undetermind coefficients

The method of undetermind coefficients is an approach that begins with the trapezoid rule and can extend to Gaussian quadrature.

Consider the approximation of an integral:

#### Rederivation of the trapezoid rule

The Trapezoid rule must be exact when $f(x)$ is a straight line; examples of which are $f = 1$ and $f=x$. Therefore,

$$ I = c_0 + c_1 = \int_{(b-a)/2}^{(b-a)/2} dx = b-a$$
$$ I = -c_0 \frac{b-a}{2} + c_1 \frac{b-a}{2} = \int_{(b-a)/2}^{(b-a)/2} x dx = 0$$

With these 2 equations we can solve for $c_0 = c_1 = \frac{b-a}{2}$ and

$$ I = \frac{b-a}{2} f(a) + \frac{b-a}{2} f(b) $$

which is the trapezoid rule.

## The 2-point Gauss-Legendre formula

Now let's let the function evaluation points float within the subdomain:

$$ I\approx c_0 f(x_0) + c_1 f(x_1) $$

Since we now have 4 unknows, we require 4 conditions to determine them. Let's simply extend the conditions above to require that parabolas and cubics are also exactly integrated:

$$\begin{split} \begin{align} c_0 f(x_0) + c_1 f(x_1) &= \int_{-1}^1 1 dx &= 2 \\ c_0 f(x_0) + c_1 f(x_1) &= \int_{-1}^1 x dx &= 0 \\ c_0 f(x_0) + c_1 f(x_1) &= \int_{-1}^1 x^2 dx &= \frac{2}{3} \\ c_0 f(x_0) + c_1 f(x_1) &= \int_{-1}^1 x^3 dx &= 0 \\ \end{align} \end{split}$$

which can be solved simultaneously to find, $
$ \begin{align} c_0 &= c_1 = 1 \\ x_0 &= -\frac{1}{\sqrt{3}}\\ x_1 &= \frac{1}{\sqrt{3}} \end{align} $$

and therefore:

$$ I = f\bigg(\frac{-1}{\sqrt{3}}\bigg)+f\bigg(\frac{1}{\sqrt{3}}\bigg) $$

That is a remarkable result! A cubic function can be integrated **exactly** meerly through the sum of the function evaluated at 2 points?!?

```
import numpy as np

def two_point_gaussian_quadrature(f):
  #Calculates the integral of a function between -1 and 1 using 2-point Gaussian quadrature.
  return f(-1 / np.sqrt(3)) + f(1 / np.sqrt(3))

import sympy as sp

x = sp.var('x')

def quad_tester(f):
  fcn = sp.lambdify(x, f)
  fI = sp.integrate(f, (x, -1, 1))
  print('Function ', f, ' Integral is ', fI, ' error ', two_point_gaussian_quadrature(fcn)-fI)

quad_tester( x**2)
quad_tester( x**3-x**2+np.pi*x+17)
quad_tester( -10**9* x**3 + np.exp(8.314)*x**2 - np.pi**.5*x + 42)
```

```
Function  x**2  Integral is  2/3  error  2.22044604925031e-16
Function  x**3 - x**2 + 3.14159265358979*x + 17  Integral is  33.3333333333333  error  0
Function  -1000000000*x**3 + 4080.60279354035*x**2 - 1.77245385090552*x + 42  Integral is  2804.401862382
```

## Transformation to an arbitrary domain

To apply Gaussian quadrature to an arbitrary integration limits \([a,b]\) we simply do a coordinate transformation:

\[ x = a_0 + a_1 x_d\]

at the limits, $\(\begin{align} a &= a_0 + a_1 (-1) \\ b &= a_0 + a_1 (1) \end{align}\)$

Such that, $\(\begin{align} a_0 &= \frac{b+a}{2}\\ a_1 &= \frac{b-a}{2} \end{align}\)$

and therefore,

\[ x = \frac{[b+a] + [b-a] x_d}{2}\]

and $\( dx = \frac{b-a}{2} dx_d\)$

which can be inserted into the integrand function at which point to make a new function in the form Gaussian Quadrature expects.

```
def flex_gq(f, a, b):
  x0 = (-1 / np.sqrt(3))
  x1 = (1 / np.sqrt(3))

  x_d0 = ((b + a) + (b - a) * x0) / 2
  x_d1 = ((b + a) + (b - a) * x1) / 2
  dx = (b - a) / 2

  return (f(x_d0) + f(x_d1))*dx

def flex_gq_tester(f, a, b):
  fcn = sp.lambdify(x, f)
  fI = sp.integrate(f, (x, a, b))
  print('Function ', f, ' Integral is ', fI, ' error ', flex_gq(fcn,a,b)-fI)

flex_gq_tester( x**2, 0, 1)
flex_gq_tester( x**3-x**2+np.pi*x+17, 0, 1)
```

```
Function  x**2  Integral is  1/3  error  0
Function  x**3 - x**2 + 3.14159265358979*x + 17  Integral is  18.4874629934616  error  -3.55271367880050e
```

Jinkies!

## The n-point Gauss-Legendre formula

The method may be generalized to higher-point problems that will find higher degree polynomials exactly.

$\( I = \sum_i c_i f(x_i) \)\( with \)c_i$ given in the table below:

| Integration order | Coefficient (ci) | Gauss Point (xi) | Error |
|---|---|---|---|
| 2 | 1 | \(\pm \frac{1}{\sqrt{3}}\) | \(f^{(4)}\) |
| 3 | \(\frac{8}{9}\) | \(0\) | \(f^{(6)}\) |
| | \(\frac{5}{9}\) | \(\pm \frac{3}{5}\) | |
| 4 | \(\frac{18 \pm \sqrt{30}}{36}\) | \(\pm\sqrt{\frac{3}{7} \pm \frac{2}{7}\sqrt{\frac{6}{5}}}\) | \(f^{(8)}\) |
| \(\vdots\) | | | |

**Error**

Note that the error is given in terms of a derivative of \(f\), since the step size is no longer an issue. In general, the error is,

$$Error = \frac{2^{2n+1} [n+1]!^4}{[2n+3][2n+2]!^3} f^(2n+2)(\xi)$$

i.e.: proportional to the \([2n+2]th\) derivative evaluated at some point ξ in \([-1,1]\).

## N-D Gauss-Legendre quadrature

Due to the separable nature of N-D integration, extension of this concept to N-D is an exercise in repeated application to determine coefficients and Gauss-points on some standard N-D interval.

### Quadrilateral elements

Consider the double integral, $
\( \int_{-1}^{1} \int_{-1}^{1} f(\alpha, \beta) d\alpha d\beta = \int_{-1}^{1} \bigg[\int_{-1}^{1} f(\alpha, \beta) d\alpha \bigg] d\beta\)
$

which can now be broken down into repeated applications of 1D integration.

We will need to transform the original quadrilateral into our standard range, for which we use:

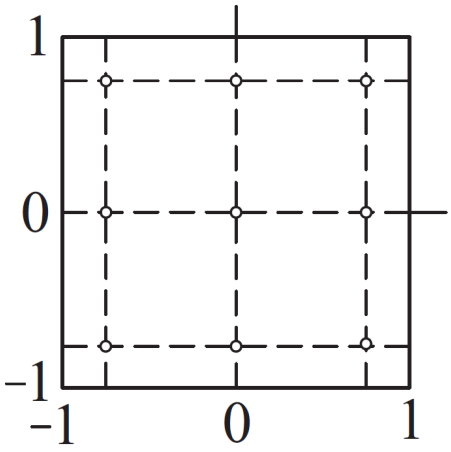
$$\begin{split}\begin{align} x(\alpha,\beta) &= \sum_{k=1}^4 N_k(\alpha,\beta)x_k \\ y(\alpha,\beta) &= \sum_{k=1}^4 N_k(\alpha,\beta)y_k \end{align}\end{split}$$
where \(<x_k, y_k>\) are the coordinates of hte corners of the quadralateral, and the *shape functions* are,

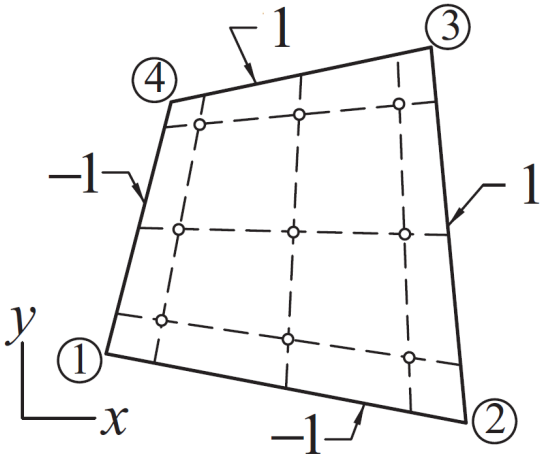
$$\begin{split}\begin{align} N_1(\alpha, \beta) = \frac{1}{4}[1-\alpha][1-\beta] \\ N_2(\alpha, \beta) = \frac{1}{4}[1+\alpha][1-\beta] \\ N_3(\alpha, \beta) = \frac{1}{4}[1+\alpha][1+\beta] \\ N_4(\alpha, \beta) = \frac{1}{4}[1-\alpha][1+\beta] \\ \end{align}\end{split}$$
Note the shape functions are *bilinear* (linear in each coordinate) such that straight lines remain straight upon mapping.

The mapping distorts the area of the quadrilateral,

$$ dx dy = \bigg\| \frac{\partial <x,y>} {\partial <\alpha, \beta>} \bigg\| d\alpha d\beta$$
which can be derived from the relations above.

An example of a quadralateral with a 3rd integration order is below showing the mapping between computational coordinates (left) and *real* coordinates (right). Gauss points are given in circles.



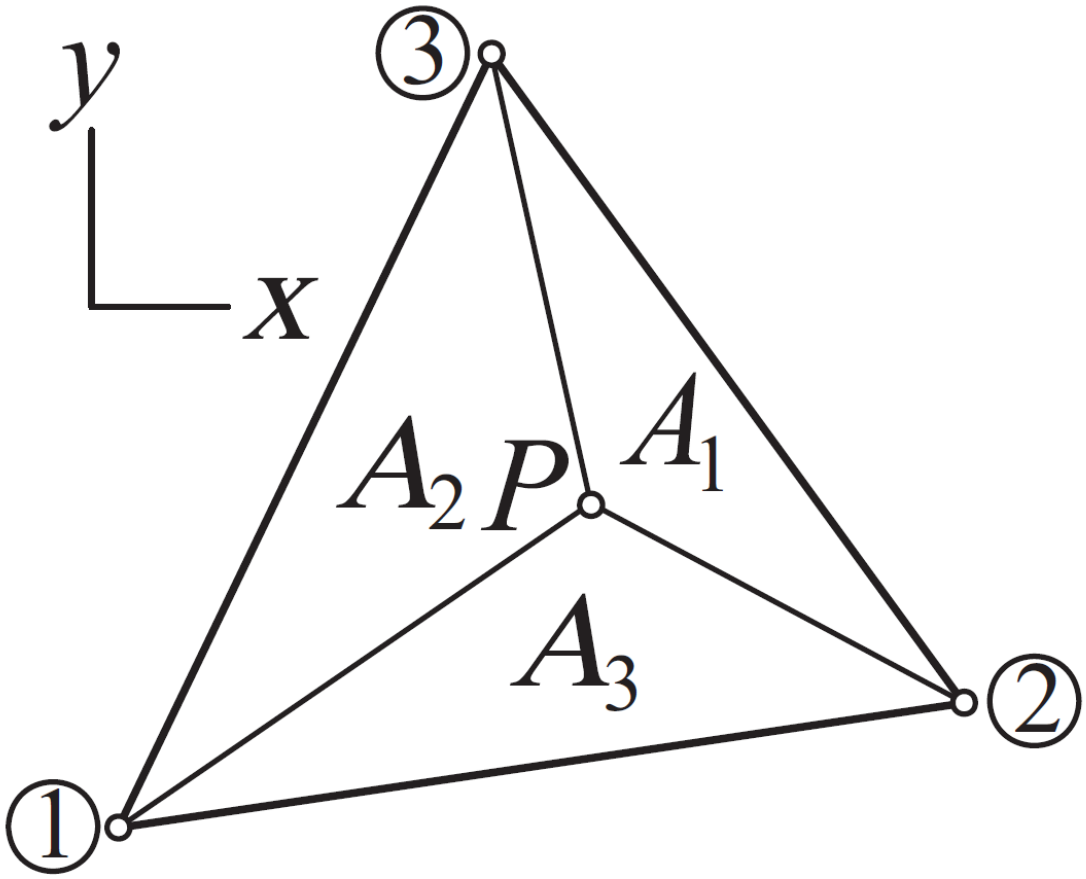




### Triangular elements

In some irregular domains, a quadralateral may be forced to reduce a side length to zero, thereby becoming degenerate quadralateral, or simply a triangle. Since the former can always be divided into two triangles, it is sensible to consider quadrature on a triangle.

In fact, since triangles are the 2D simplex they are able to fill any shape and are typically the go-to for tesselation (space-filling tiling).

Consider dividing a triangle into 3 parts connecting the verticies with a point P,




so as to define areas \(A_{1,2,3}\). The *area coordinates* of P are, $\(\alpha_i = \frac{A_i}{A}\)$

and since \(A_1+A_2+A_3 = A\), $\(\alpha_1 +\alpha_2+\alpha_3 = 1\)$.

Note that \(\alpha_i\) ranges from 0 to 1 when P moves from the opposing side to the corner \(i\).

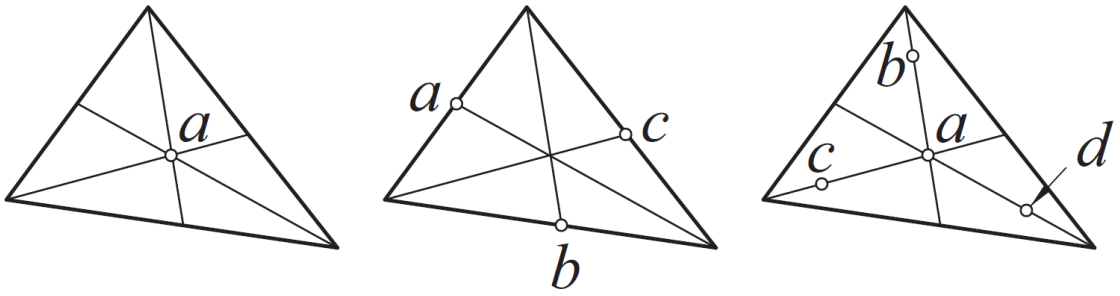
Using these coordinates, $
\(\begin{align} x(\alpha_1, \alpha_2, \alpha_3) = \sum_{i=1}^3 \alpha_i x_i \\ y(\alpha_1, \alpha_2, \alpha_3) = \sum_{i=1}^3 \alpha_i y_i \\ \end{align} \)
$

and the integration becomes, $\(\int \int_A f(x,y) dA = A \sum_k W_k f\big(x(\alpha_k), y(\alpha_k)\big) \)$

with the area, $\(A = \frac{1}{2} \begin{vmatrix} 1 &1 &1 \\ x_1& x_2& x_3 \\ y_1& y_2& y_3 \end{vmatrix}\)$

The weights \(W_k\) are given by,

| Order (n) | Point | Weight (\(W_k\)) | Area Coordinates (\(\alpha_1\), \(\alpha_2\), \(\alpha_3\)) |
|---|---|---|---|
| Linear | a | 1.0 | (1/3, 1/3, 1/3) |
| Quadratic | a | 1/3 | (1/2, 0, 1/2) |
| | b | 1/3 | (1/2, 1/2,0) |
| | c | 1/3 | (0, 1/2, 1/2) |
| Cubic | a | -27/48 | (1/3, 1/3, 1/3) |
| | b | 25/48 | (1/5, 1/5, 3/5) |
| | c | 25/48 | (3/5, 1/5, 1/5) |
| | d | 25/48 | (1/5, 3/5, 1/5) |




(a) Linear (b) Quadratic (c) Cubic

Open in Colab

# Differential equations

> Since Newton, mankind has come to realize that the laws of physics are always expressed in the language of differential equations. - Steven Strogatz

Differential equations relate functions and their derivatives, and are pervasive in modern engineering and science. The study of differential equations is vast with many innovative ideas being explored.

# An incomplete classification of equations

| Classification | Type | Description | Example |
|---|---|---|---|
| **Order of Differential Equations** | First-Order | Involves the first derivative of the function. | $\(\frac{dy}{dx} = y\)$ |
| | Second-Order and Higher | Involves second or higher derivatives. | $ \(\frac{d^2y}{dx^2} + 3\frac{dy}{dx} + 2y = 0\) $ |
| **Linear vs. Nonlinear Differential Equations** | Linear | The dependent variable and its derivatives appear linearly. | $ \(\frac{d^2y}{dx^2} + p(x)\frac{dy}{dx} + q(x)y = g(x)\) $ |
| | Nonlinear | The equation involves nonlinear terms of the dependent variable or its derivatives. | $ \(\left(\frac{dy}{dx}\right)^2 + y = 0\) $ |
| **Homogeneous vs. Non-Homogeneous Differential Equations** | Homogeneous | All terms are a function of the dependent variable and its derivatives. | $\(\frac{d^2y}{dx^2} - y = 0\)$ |
| | Non-Homogeneous | Includes terms that are not a function of the dependent variable or its derivatives. | $\(\frac{d^2y}{dx^2} - y = e^x\) $ |
| **Initial / Boundary Value Problems** | Initial Value Problems | The solution is determined by the value of the function and its derivatives at a single point. | $ \(\frac{dy}{dx} = y, \quad y(0) = 1\) $ |
| | Boundary Value Problems | The solution is determined by the values of the function at multiple points. | $ \(\frac{d^2y}{dx^2} = -y, \quad y(0) = 0, \quad y(\pi) = 0\) $ |
| **Ordinary vs. Partial Differential Equations** | Ordinary Differential Equations (ODEs) | Involve functions of a single variable and their derivatives. | $\(\frac{dy}{dx} + y = 0\)$ |
| | Partial Differential Equations (PDEs) | Involve functions of multiple variables and their partial derivatives. | $ \(\frac{\partial u}{\partial t} = c^2 \frac{\partial^2 u}{\partial x^2}\) $ |
| **Time dependent PDEs** | Elliptic PDEs | Stationary problems in time. | Laplace's equation, $ \(\nabla^2 u = 0\)$ |
| | Parabolic PDEs | First derivative in time. | The heat equation, $ \(\frac{\partial u}{\partial t} = \alpha \nabla^2 u\) |

| Classification | Type | Description | Example |
|---|---|---|---|
| | | | $ |
| | Hyperbolic PDEs | Second derivative in time. | The wave equation, $ \(\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u\) $ |

# Initial value problems

Initial Value Problems are characterised by knowledge of boundary conditions only on one side of the domain. This is typically (but not necessarily) for time-dependent equations, thus the name and following nomenclature.

e.g.: $\(\frac{dy}{dt} = y, \quad y(t=0) = 1\)\( or \)\(\frac{d^2 y}{dt^2} = -g, \quad y(t=0) = 1000, y'(t=0) = 0\)$

In these cases we are forced to start at \(t=0\) and *time-step* forward until we reach our goal time. This has several implications:

- Error in the soultion will accumulate with successive steps.
- Numerical noise may destabilize the solution scheme.
- Time stepping must resolve the time-scale of the physics we are looking at.
- The time-scale may change mid-problem.
- We will build up a *history* of steps as we go.

As with analytical methods, on a high level we obtain a solution by *integrating* the differential equation up until the point of interest. For this reason, solving IVPs are often called *time integrators* and much of the analysis carries over.

## Error analysis

As usual, we will discuss error in the context of truncation error. As with other integration schemes,

- *Local truncation error* occurs over the time-step due to the approximation of \(y_{i+1}\).
- *Global / propogated truncation error* is the accumulation of propogated local truncation error and results in overall error of the approximation to the true solution.

Later we will discuss a new phenomenon, stability and *stiffness*.

We will be using the Taylor expansion to analyse error and help deevelop our numerical schemes. Higher accuracy is achieved through estimation of the higher order derivatives. There are two classes of methods that are applied:

- Single step methods - Uses information about the current time step
- Multistep methods - Uses the history of time steps

## Explicit methods

Explicit solver methods take information up to the current solution in order to predict the next step.

### Runge-Kutta methods

The Runge-Kutta methods are a class of time stepping techniques where the next time step is calculated from the *current time step* and an estimate of its *slope* (rate of change). Increased accuracy is achieved through approximating higher order derivatives to improve our timestep. Since we don't generally have that information, we can sample the function at different steps to approximate them. This is the basis for the Runge-Kutta family of methods.

Consider the Taylor expansion, $
\(\begin{align} y_{i+1} &= y_i + y_i^\prime h + \frac{y_i^{\prime\prime}}{2}h^2 + \frac{y'''}{6}h^3 + \cdots \\ &= y_i + f(x_i,y_i) h + \frac{f(x_i,y_i)^\prime}{2}h^2 + \frac{f(x_i,y_i)^{\prime\prime}}{6}h^3 + \cdots \\ &\approx y_i + h \sum_{n=1}^s a_n k_n \end{align}\)
$

where the last line is the general form of the Runge-Kutta methods. \(a_n\) are a set of constants and \(k_n\) are the function evaluated at different positions in the interval. The goal is clearly to match $
\(\sum_{n=1}^s a_n k_n \approx f(x_i,y_i) + \frac{f(x_i,y_i)^\prime}{2}h + \frac{f(x_i,y_i)^{\prime\prime}}{6}h^2 + \cdots \)$

in so far as possible with a series truncated in \(s\) terms (called stages).

Let's build up to the general form step by step.

**Forward Euler method**

The Forward Euler method (aka: Euler-Cauchy / point-slope method, Explicit Euler) is the simplest Runge-Kutta method. One simply takes the slope at the current point, which is given in the differential equation, and assumes it is constant over the step:

Given, $\( \frac{dy}{dx} = f(x, y) \)\( the next time step is: \)\( y_{i+1} = y_i + f(x_i, y_i) h\)$

which is simply the left Riemann sum, and similarly the error is \(O(h^2)\) over the step, and \(O(h)\) over the full solution.

**Heun's method**

Now that we have a value for \(y_{i+1}\) is there a way we can use the *prediction* to *correct* itself? Heun's method does exactly this and is called a **predictor-corrector** algorithm as a result.

Consider:

\[ \frac{dy}{dx} = f(x, y) \]

make a prediction, $\( y^0_{i+1} = y_i + f(x_i, y_i) h\)\( Now take the average of the slopes: \)
\(\begin{align} \bar{y}' &= \frac{y_i+y_{i+1}}{2} \\ &= \frac{f(x_i,y_i)+f(x_{i+1}, y^0_{i+1})}{2} \end{align} \)
\( which is used in the corrector: \)\( y_{i+1} = y_i + \frac{f(x_i,y_i)+f(x_{i+1}, y^0_{i+1})}{2} h\)$

This is interesting since one could repeat the predictor-corrector cycle with the intention of converging towards the correct answer, but we will see this is a bad idea:

**Example: Heun's method with multiple predictor-corrector cycles**

Integrate $\(y' = 4 e^{0.8 x}-0.5 y\)\( with \)\( y(0)=2\)$ using Heun's method with 1 and 15 predictor-corrector cycles.

```python
import numpy as np

def f(x, y):
  return 4 * (2.71828 ** (0.8 * x)) - 0.5 * y

def heuns_method(x0, y0, h, t, iterations):
  """
  Applies Heun's method to approximate the solution of a differential equation.

  Args:
    x0: The initial x value.
    y0: The initial y value.
    h: The step size.
    t: The target time for the approximation.
    iterations: The number of iterations for the predictor-corrector cycle.

  Returns:
    The approximate y value at time t.
  """
  x = x0
  y = y0
  while x < t:
    y_pred = y + h * f(x, y)
    y_next = y + h * (f(x, y) + f(x + h, y_pred)) / 2
    for _ in range(iterations - 1):
      y_pred = y_next
      y_next = y + h * (f(x, y) + f(x + h, y_pred)) / 2

    y = y_next
    x += h
    print("Time ", x, ", approximation, ", y_next, ', True ,', 4/1.3*(np.exp(.8*x)-np.exp(-.5*x))+2*np.e
  return y


# Initial conditions
x0 = 0
y0 = 2

# Step size and target time
h = 1
t = 4

# Estimate integral with Heun's method, iterating once
print("Heun's method with 1 iteration")
y_approx_once = heuns_method(x0, y0, h, t, 1)

# Estimate integral with Heun's method, iterating 15 times
print("\nHeun's method with 15 iterations")
y_approx_15 = heuns_method(x0, y0, h, t, 15)
```

```
Heun's method with 1 iteration
Time  1 , approximation,  6.701079461759733 , True , 6.194631377209372
Time  2 , approximation,  16.319768581929353 , True , 14.84392190764649
Time  3 , approximation,  37.199199627848756 , True , 33.67717176796817
Time  4 , approximation,  83.33761313495894 , True , 75.33896260915857

Heun's method with 15 iterations
Time  1 , approximation,  6.360863570675189 , True , 6.194631377209372
Time  2 , approximation,  15.302225064771143 , True , 14.84392190764649
Time  3 , approximation,  34.74323213193692 , True , 33.67717176796817
Time  4 , approximation,  77.73495685652544 , True , 75.33896260915857
```

Unfortunatley, Heun's method does converge but not necessarily to the correct answer!

### The Trapezoid rule

In the case that the slope doesn't depend on the function value, $y'(x) = f(x)$ the predictor can be calcualted directly, eliminating the cycle:

$$ y_{i+1} = y_i + \frac{f(x_i)+f(x_{i+1})}{2} h$$

which is mearly the trapezoid rule which carries $O(h^3)$ accuracy locally and $O(h^2)$ globally.

#### The Midpoint method

Recall from the discussion on on differentiation / integration that information from the midpoint was often superior that of either endpoint (in isolation) as over/under estimates tend to cancel.

As an alternative to Heun's predictor-corrector method, let's subdivide the interval and find the slope at the *midpoint*. To do this, take a half step:

$$ y_{i+1/2} = y_i + f(x_i,y_i) \frac{h}{2} $$

then use the slope at the midpoint, $y'_{i+1/2} = f(x_{i+1/2},y_{i+1/2})$ to estimate the full step:

$$ y_{i+1} = y_i + f(x_{i+1/2},y_{i+1/2}) h $$

which has $O(h^3)$ local / $O(h^2)$ global error.

## Explicit Runge-Kutta methods

We are now in a position to generalize the RK methods. Recalling:

$$ y_{i+1} \approx y_i + h \sum_{n=1}^s a_n k_n$$

For the *explicit* set of RK methods,

$$\begin{split}\begin{align} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + p_2 h, y_i + [q_{21}k_1] h) \\ k_3 &= f(x_i + p_3 h, y_i + [q_{31}k_1 + q_{32}k_2] h) \\ \vdots\\ k_s &= f(x_i + p_s h, y_i + [q_{s1}k_1 + q_{s2}k_2 ... q_{s,s-1} k_{s-1}] h) \\ \end{align} \end{split}$$

The $p_n$ and $q_{nm}$, along with $a_n$ are constants and determine the type of RK method.

Note RK-1 is simply the Forward Euler equation.

#### Butcher Tableaus

Noting that $a_n$, $p_n$, and $q_{nm}$ are 2 vectors of dimension $s$ and an $s \times s$ matrix, RK schemes can be written compactly in a *Butcher Tableau*. For explicit methods, this looks like:

$$\begin{split}\begin{array}{c|ccccc} p_1 & & & & &\\ p_2 & q_{21} & & & &\\ p_3 & q_{31} & q_{32} & & &\\ \vdots & \vdots & & \ddots & &\\ p_s & q_{s1} & q_{s2} & \cdots & q_{s,s-1} &\\ \hline & a_1 & a_2 & \cdots & a_{s-1} & a_s \\ \end{array}\end{split}$$

Note that the $q_{nm}$ matrix is lower triangular. We will soon introduce the *implicit* family of RK methods which are upper triangular!

#### RK-2 methods

For a given order, the values of $a_n$, $p_n$, and $q_{nm}$ are derived such that

$$\sum_{n=1}^s a_n k_n \approx f(x_i,y_i) + \frac{f(x_i,y_i)^\prime}{2}h + \frac{f(x_i,y_i)^{\prime\prime}}{6}h^2 + \cdots $$

The second order RK method is:

$$y_{i+1} = y_i + [a_1 k_1 + a_2 k_2] h $$

with $\begin{align} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + p_2 h, y_i+q_{21} k_1 h) \end{align} $

Taylor expanding $k_2$ in $p$ and $q$), $
$ f(x_i + p_2 h, y_i+q_{21} k_1 h) = f(x_i,y_i) + p_2 h \frac{\partial f}{\partial x} + q_{21} k_1 h \frac{\partial f}{\partial y} + O(h^2)$
$

which plugged back in to $y_{i+1}$: $
$y_{i+1} = y_i + [a_1+a_2] f(x_i, y_i) h + \bigg[a_2 p_2 \frac{\partial f}{\partial x} + a_2 q_{21} f(x_i, y_i) \frac{\partial f}{\partial y} \bigg] h^2 + O(h^3) $
$

which we can compare to a second order Taylor expansion: $
$ \begin{align} y_{i+1} &= y_i + f(x_1, y_1) h + \frac{f'(x_1, y_1)}{2} h^2 \\ &= y_i + f(x_1, y_1) h + \bigg[\frac{\partial f}{\partial x} +\frac{\partial f}{\partial x} \frac{dy}{dx} \bigg] \frac{h}{2} \end{align}$
$

Comparing terms we see: $$\begin{align} a_1+a_2 &= 1 \\ a_2 p_2 &= \frac{1}{2} \\ a_2 q_{21}&= \frac{1}{2}\\ \end{align} $$

Here we see that we have a single degree of freedom for the set of constants! Any choice will satisfy 2'nd order equations and therefor ebe exact for constant, linear, or quadratic ODEs. Certaint choices will have better properties in general.

**Heun's method**

With $a_2 = \frac{1}{2}$,

$$y_{i+1} = y_i + \frac{k_1 + k_2}{2} h $$

with $$\begin{align} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + h, y_i + k_1 h) \end{align} $$

which is simply Heun's method.

$$\begin{array}{c|cc} 0 & & \ 1 & 1 & \ \hline & \frac{1}{2} & \frac{1}{2} \ \end{array}$$

**Midpoint method**

With $a_2 = 1$, $y_{i+1} = y_i + k_2 h $ with $
$$\begin{align} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + \frac{1}{2}h, y_i + \frac{1}{2} k_1 h) \end{align} $$ which is the midpoint method.

$$\begin{array}{c|cc} 0 & & \ \frac{1}{2} & \frac{1}{2} & \ \hline & 0 & 1 \ \end{array}$$

**Ralston's method**

The choice $a_2 = \frac{2}{3}$, can be shown to provide a minimum bound on the truncation error, $
$y_{i+1} = y_i + [k_1 + 2 k_2 ] \frac{h}{3} $ with $
$$\begin{align} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + \frac{3}{4}h, y_i + \frac{3}{4} k_1 h) \end{align} $$

$$\begin{array}{c|cc} 0 & & \ \frac{2}{3} & \frac{2}{3} & \ \hline & \frac{1}{4} & \frac{3}{4} \ \end{array}$$

**RK-3 methods**

A similar derivation follows for RK3, again with choices for the missing degrees of freedom. A common choice is:

$$y_{i+1} = y_i + \frac{1}{6}[k_1 + 4 k_2 + k_3] h $$

with $
$$\begin{align} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + \frac{1}{2}h, y_i + \frac{1}{2} k_1 h) \\ k_3 &= f(x_i + h, y_i - k_1 h + 2 k_2 h) \\ \end{align} $$
$

which reduces to Simpson's 1/3 Rule if $f$ is only a function of $x$. As with Simpson's rule, it is $O(h^4)$ local / $O(h^3)$ global error.

The Butcher Tableau is: $$\begin{array}{c|ccc} 0 & & & \ \frac{1}{2} & \frac{1}{2} & & \ 1 & -1 & 2 & \ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \ \end{array}$$

**RK4**

RK4 is the most common implementation. The ‘Classical Runge-Kutta method’ is:

$$y_{i+1} = y_i + \frac{1}{6}[k_1 + 2 k_2 + 2 k_3 + k_4] h$$

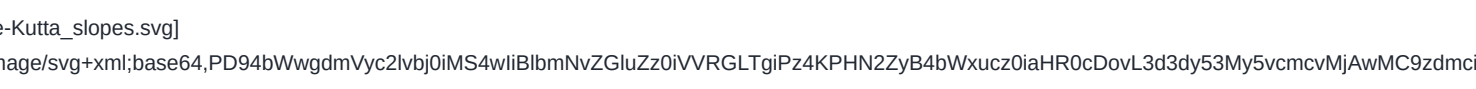
with $
\(\begin{align} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + \frac{1}{2}h, y_i + \frac{1}{2} k_1 h) \\ k_3 &= f(x_i + \frac{1}{2}h, y_i + \frac{1}{2} k_2 h ) \\ k_4 &= f(x_i + h, y_i + k_3 h) \end{align} \)
$

with \(O(h^5)\) local / \(O(h^4)\) global error.

The Butcher Tableau is: $
\( \begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \\ \end{array} \)
$

**Example - RK4 steps**

An example of the RK4 algorithm is below, showing partial steps inform subsequent steps culminating in very good estimate!

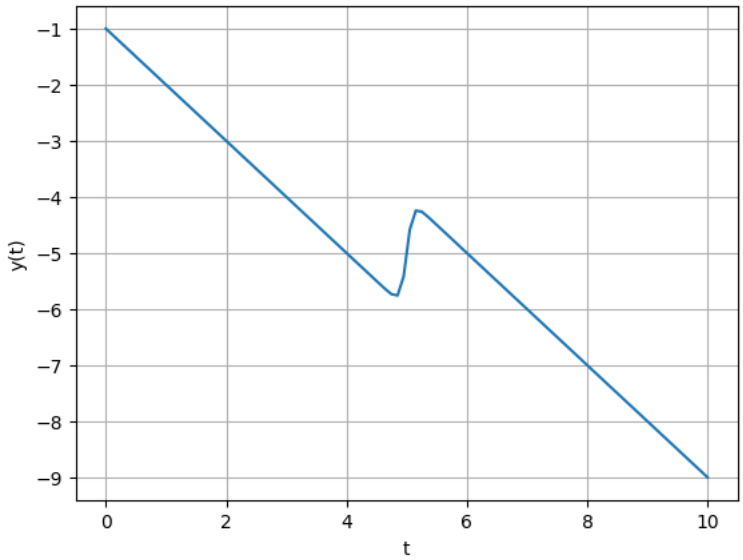



## Adaptive time stepping methods

It is not uncommon for systems to have a combination of fast transients with otherwise slow behaviour. In general, we won’t know when the transient occurs, or even it’s rate of change!

E.g.: The system \(y(t) = \tanh(\frac{t-5}{0.1})-t\) has a slow decrease with a sharp bump. If the time step is small, the bump will be resolved, but it will take a long time for the quasi-linear parts. If the step is too large, the bump may be missed entirely!

```
import numpy as np
import matplotlib.pyplot as plt

# Create a range of t values
t = np.linspace(0, 10, 100)

# Plot the function
plt.plot(t, np.tanh((t-5)/.1) - t)
plt.xlabel('t')
plt.ylabel('y(t)')
plt.grid(True)
plt.show()
```




*Adaptive* time steppers automatically adjust their step size based on an estimate of the error.

If your timestep has an error \(E_{current}\) with stepsize \(h_{current}\), and you are aiming for an error tolerance \(E_{goal}\) then you can update your step size with a formula like:

$$h_{new} = h_{current} \bigg|\frac{E_{goal}}{E_{current}} \bigg|^\alpha$$

where \(\alpha\) is some user value (0.2 is a good choice).

There are remaining quesitons though:

- Would you reject the time step if \(E_{current} \gt E_{tolerance}\) and repeat with a new timestep?
- Would you have a *meta* analysis of the solver method to 'catch' non-convergent cases?

There are a few methods to capture the error. Naturally, you don't want the error calculation to be overly burdomsome.

- Change step-size
- Change integration order
- Directly compare \(y_i\) to \(y_{i+1}\) or other properties of the solution.

**Step-halving methods**

Step-halving methods compare the results of a full step to two half steps. For RK4, the error is calculated as:

$$E = y_{double \ step} - y_{single\ step} \ $$

for this RK4 scheme, one could correct higher accuracy estimate with $\(y_{double \ step}^* = y_{double \ step} + \frac{E}{15}\)
$

Example: Integrate $\(y^\prime = 4 e^{0.8x}-0.5y\)\( with \)y(0) = 2\( from \)x = 0\ to\ 2$. The analytic answer is 14.84392.

```
# prompt: solve above with RK4

import numpy as np
import matplotlib.pyplot as plt

def f(x, y):
  """The differential equation."""
  return 4 * np.exp(0.8 * x) - 0.5 * y

def rk4_step(f, x, y, h):
  """Takes a single RK4 step."""
  k1 = h * f(x, y)
  k2 = h * f(x + h/2, y + k1/2)
  k3 = h * f(x + h/2, y + k2/2)
  k4 = h * f(x + h, y + k3)
  return y + (k1 + 2*k2 + 2*k3 + k4) / 6

# Initial conditions
x0 = 0
y0 = 2
x_end = 2
h_initial = 2


y_onestep = rk4_step(f, x0, y0, 2)
print('One stpe with h=2 ', y_onestep)

y_t = rk4_step(f, x0, y0, 1)
y_twostep = rk4_step(f, x0+1, y_t, 1)
print('Two steps with h=1 ', y_twostep)

print('Approximate error is ', (y_twostep - y_onestep)/15, ' vs true error, ', 14.84392 - y_twostep)
```

```
One stpe with h=2  15.105846327501714
Two steps with h=1  14.8624835881192
Approximate error is  -0.016224182625500915  vs true error,  -0.018563588119199892
```

Note that to calculate this value we had to evaluate $f(x,y)$ 8 times.

#### Runge-Kutta Fehlberg

Another approach is to compare different integration orders over the same step. This is made efficient by *reusing* function calls between the two approximations.

$$\begin{split} \begin{array}{c|cccccc} 0 & & & & & & \\ \frac{1}{5} & \frac{1}{5} & & & & & \\ \frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & \\ \frac{3}{5} & \frac{3}{10} & -\frac{9}{10} & \frac{6}{5} & & & \\ 1 & -\frac{11}{54} & \frac{5}{2} & -\frac{70}{27} & \frac{35}{27} & & \\ \frac{7}{8} & \frac{1631}{55296} & \frac{175}{512} & \frac{575}{13824} & \frac{44275}{110592} & \frac{253}{4096} & \\ \hline & \frac{37}{378} & 0 & \frac{250}{621} & \frac{125}{594} & 0 & \frac{512}{1771} \\ & \frac{2825}{27648} & 0 & \frac{18575}{48384} & \frac{13525}{55296} & \frac{277}{14336} & \frac{1}{4} \\ \end{array} \end{split}$$
where the doubling of the last line means:

$$\begin{split} \begin{align} y_{i+1}^{(4)} &= y_i + h \left( \frac{37}{378} k_1 + \frac{250}{621} k_3 + \frac{125}{594} k_4 + \frac{512}{1771} k_6 \right) \\ y_{i+1}^{(5)} &= y_i + h \left( \frac{2825}{27648} k_1 + \frac{18575}{48384} k_3 + \frac{13525}{55296} k_4 + \frac{277}{14336} k_5 + \frac{1}{4} k_6 \right) \end{align}\end{split}$$
The error is then simply $y_{i+1}^{(5)} - y_{i+1}^{(4)}$

This is the method of choice for packaged tools. E.g.:

```python
# prompt: solve 4 * np.exp(0.8 * x) - 0.5 * y with scipy , step size 2 and output the error

import numpy as np
from scipy.integrate import solve_ivp

def f(x, y):
  """The differential equation."""
  return 4 * np.exp(0.8 * x) - 0.5 * y

# Initial conditions
x0 = 0
y0 = 2
x_end = 2

# Solve the differential equation using solve_ivp
sol = solve_ivp(f, (x0, x_end), [y0], method='RK45')
print(sol)


# Extract the solution
y_numerical = sol.y[0][-1]

# Analytical solution (you might need to calculate this beforehand)
y_analytical = 14.84392

# Calculate the error
error = abs(y_numerical - y_analytical)

print(f"Numerical solution: {y_numerical}")
print(f"Error: {error}")
```

```
  message: The solver successfully reached the end of the integration interval.
  success: True
   status: 0
        t: [ 0.000e+00  9.222e-02  1.014e+00  2.000e+00]
        y: [[ 2.000e+00  2.284e+00  6.279e+00  1.484e+01]]
      sol: None
 t_events: None
 y_events: None
     nfev: 20
     njev: 0
      nlu: 0
Numerical solution: 14.844062517715555
Error: 0.00014251771555429116
```

**Direct comparison**

A less elegant, but sometimes more pragmatic method is use other metrics to control the step size.

Following a time step, the rate of change can be calculated,
$\(\frac{dy}{dt} = \frac{y_{i+1}-y_i}{h}\)\( from which a control on \)|\frac{dy}{dt}|
\( can be placed. E.g.: one could require that the Euclidian or \)\infty$ norm be below a tolerance. This method controls the change in solution, rather than its error.

An even more brute-force method may be to observe the efficiency of the solver. We konw that nonlinear root finding with Newton's method converges quadratically near the root. By monitoring the Newton iterations at each timestep, one can assess if the solver is converging quadratically, or is taking too many iterations which may mean the new time step is too far from the previous one.

Open in Colab

## Implicit methods

Up until this point, we have been discussing *explicit* solver schemes, in which \(y_{i+1}\) is determined using the information \(x_i\) and \(y_i\) with kowledge of \(f(x_i, y_i)\).

*Implicit* methods uses information at the new time step, in order to determine it; i.e.: using \(f(x_{i+1}, y_{i+1})\)! The new value of \(y_{i+1}\) is therefore calcualted implicitly, for which we will generally need to use a root finder. The root finder adds

significant computational expense but is substantially less sensative to numerical instability.

## The Backward Euler method

The simplest implicit scheme is the implicit Euler method (backward Euler). Like the forward Euler method we assumes a constant slope over the timestep but this time it is the slope *at the end of the timestep*:

$$ y_{i+1} = y_i + f(x_{i+1}, y_{i+1}) h$$

which implicitly defines $y_{i+1}$.

## Numerical instability

An algorithm is numerically unstable when small errors that occur during computation grow. Such errors can occur due to user-choices (e.g. - too large a step size) or simply round-off/truncation error.

NB: numerical instability is a property of the *algorithm*, not the equation.

#### Example: Initial value problem

Integrate $\frac{\partial y}{\partial t} = - y$

from 0 to 20 using the forward and backward Euler methods and compare to the exact solution, $ y(t) = e^{-t}$

for varying step sizes.

```
# prompt: Can you make the above plot a slider for the step size ontop of the plot

import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, FloatSlider


def plot_with_slider(h):
  t0 =0
  t_end = 20
  y0 = 2
  # Time points
  t_points = np.arange(t0, t_end + h, h)

  # Analytical solution
  y_analytical = y0 * np.exp(-t_points)

  # Explicit Euler method
  y_explicit = np.zeros_like(t_points)
  y_explicit[0] = y0
  for i in range(1, len(t_points)):
      y_explicit[i] = y_explicit[i-1] * (1 - h)

  # Implicit Euler method
  y_implicit = np.zeros_like(t_points)
  y_implicit[0] = y0
  for i in range(1, len(t_points)):
      y_implicit[i] = y_implicit[i-1] / (1 + h)

  # Plotting the results
  plt.figure(figsize=(10, 6))
  plt.plot(t_points, y_analytical, label='Analytical Solution', color='black', linestyle='--')
  plt.plot(t_points, y_explicit, label='Explicit Euler', color='blue', marker='o')
  plt.plot(t_points, y_implicit, label='Implicit Euler', color='red', marker='x')
  plt.xlabel('Time')
  plt.ylabel('y(t)')
  plt.xlim(0, 20)
  plt.title('Comparison of Explicit and Implicit Euler Methods')
  plt.legend()
  plt.grid(True)
  plt.show()

interact(plot_with_slider, h=FloatSlider(min=.1, max=5, step=.1, value=0.8));
```

Gadzooks!

Let's consider the Forward Euler scheme analytically. For generality, say \(\frac{dy}{dt} = -ay\) for some positive constant \(a\),

\[\begin{split}\begin{align} y_{i+1} &= y_i+h \frac{dy}{dt}(y_{i}) \\ &= y_i - y_i a h \\ &= y_i [1-a h] \end{align} \end{split}\]
In order for the solution to follow the smoothly decreasing behaviour, \(h\lt\frac{1}{a}\). If \(\frac{1}{a}\lt h \lt \frac{2}{a}\), the solution will overshoot but eventually correct itself. But if \(h \gt \frac{2}{a}\) the method will catasrophically overcorrect!

Compare this to the implicit case,

\[\begin{split}\begin{align} y_{i+1} &= y_i+h \frac{dy}{dt}(y_{i+1}) \\ &= y_i - y_{i+1} a h \\ y_{i+1}&= \frac{y_i}{1+ah} \end{align} \end{split}\]
which is nicely decreasing for all \(h>0\) and is therefore *unconditionally stable*!

Numerical stability has therefore placed a limit on the step size for which a method will converge on the correct answer. There are different types of stability, which is outside the scope of this course.

The stability of the explicit vs implicit Euler methods helps show some intuition on what's going on: since implicit schemes focus on the end of the time step and how we *got there*, they are less sensative to errors introduced by large time steps or numerical noise.

##Stiffness

One might think that the adaptive time steppers may dynamically avoid numerical instability but you would be dissappointed…

Consider the equation $\( \frac{dy}{dt} = -1000 y + 3000 - 2000 e^{-t}\)\( with \)y(0) = 0$.

The analytical solution is $\(y(t) = 3-0.998 e^{-1000 t} - 2.0-2 e^{-t}\)\( which features a fasts transient due to \)e^{-1000 t}$ followed by a slow progression:

```
# prompt: plot 3-0.998 e^{-1000 t} - 2.0-2 e^{-t} from 0 to a limit determined by a slider starting at 3

import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, FloatSlider

def plot_function(limit):
  t = np.linspace(0, limit, 500)
  y = 3 - 0.998 * np.exp(-1000 * t) - 2 - 2 * np.exp(-t)
  plt.figure(figsize=(10, 6))
  plt.plot(t, y)
  plt.xlabel('t')
  plt.ylabel('y(t)')
  plt.title('Plot of y(t) = 3 - 0.998e^(-1000t) - 2 - 2e^(-t)')
  plt.grid(True)
  plt.show()

interact(plot_function, limit=FloatSlider(min=.1, max=5, step=0.1, value=3));
```

The stability of the fast term with forward Euler requires \(h\lt \frac{2}{1000}\) to not catastrophically overcorrect, but at least that is only for the first little bit right?

Wrong. Even though the transient only dominates the behaviour for the first little bit, it is still there for the rest, and will still tear shirt up!

```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, FloatSlider


def plot_with_slider(h):
  t0 =0.01
  t_end = .05
  y0 = 0.
  # Time points
  t_points = np.arange(t0, t_end + h, h)

  # Analytical solution
  def y_true(t):
    return 3. - 0.998*np.exp(-1000*t)  - 2.002*np.exp(-t)
  y_analytical = y_true(t_points)

  y0 = y_true(t0)

  def f(y,t):
    return -1000.*y + 3000. - 2000.*np.exp(-t)

  # Explicit Euler method
  y_explicit = np.zeros_like(t_points)
  y_explicit[0] = y0
  for i in range(1, len(t_points)):
      y_explicit[i] = y_explicit[i-1] + f(y_explicit[i-1], t_points[i-1]) * h

  # Implicit Euler method
  y_implicit = np.zeros_like(t_points)
  y_implicit[0] = y0
  def imp(y, t, y0, f, h):
    return y - (y0 + h*f(y, t))

  from scipy.optimize import root
  for i in range(1, len(t_points)):
      sol = root(imp, y_implicit[i-1], args=(t_points[i], y_implicit[i-1], f, h))
      y_implicit[i] = sol.x[0]

  # Plotting the results
  plt.figure(figsize=(10, 6))
  plt.plot(t_points, y_analytical, label='Analytical Solution', color='black', linestyle='--')
  plt.plot(t_points, y_explicit, label='Explicit Euler', color='blue', marker='o')
  plt.plot(t_points, y_implicit, label='Implicit Euler', color='red', marker='x')
  plt.xlabel('Time')
  plt.ylabel('y(t)')
  plt.xlim(t0, t_end)
  plt.title('Comparison of Explicit and Implicit Euler Methods')
  plt.legend()
  plt.grid(True)
  plt.show()

interact(plot_with_slider, h=FloatSlider(min=.0001, max=.003, step=.0001, value=0.0005, readout_format='
```

###Definition of stiffness

A precise mathematical definition of stiffness is difficult, but in general: A stiff equation is one in which accuracy of the solution is determined by numerical stability rather than the behaviour of the solution.

This is expecially problematic in physics where short transients as a system reaches a ‘well behaved state’ are common. If we include the phenomena which describe the transients, we are bound by their timescales weather or not the transient has died off!

One approach is to ignore the transients entirely! Examples include:

- Quasistatic elasticity
- diffusive heat / mass transport
- etc.

Alternately, we can use implicit methods which are more stable, but come with added computational expense.

## Implicit Runge-Kutta methods

We are now in a position to generalize the RK methods. Recalling:

$$ y_{i+1} \approx y_i + h \sum_{n=1}^s a_n k_n$$

with

$$ k_n = f(x_i + p_n, y_i + \sum_{m=1}^s q_{nm}k_m) $$

the Butcher tableau now expanded:

$$\begin{split} \begin{array}{c|cccc} p_1 & q_{11} & q_{12} & \dots & q_{1s} \\ p_2 & q_{21} & q_{22} & \dots & q_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_s & q_{s1} & q_{s2} & \dots & q_{ss} \\ \hline & a_1 & a_2 & \dots & a_s \end{array} \end{split}$$

In explicit methods, the $k_n$s could be built progressively on $k_{n-1...}$, but this is not the case for implicit RK. Rather, a set of $k_n$ may need to be solved *simultaneously* which can dramatically amplify the computaitonal expense.

The balance of computaional expense to increased accuracy has motivated a plethora of RK methods of varying stages, orders, and complexity of which we will only discuss a few:

**Implicit Euler method**

The Implicit Euler, $\( y_{i+1} = y_i + f(x_{i+1}, y_{i+1}) h\)$ has the tableau:

$$\begin{split} \begin{array}{c|c} 1 & 1 \\ \hline & 1 \\ \end{array} \end{split}$$

**RK2 implicit**

**Implicit midpoint method**

$$ \begin{align} y_{i+1} &= y_i + h k_1 \end{align}$$

where

$$k_1 = f(x_{i+\frac{1}{2}}, y_{i+\frac{1}{2}}) $$

which is found through the implicit solution of: $
$y_{i+\frac{1}{2}} = y_n + \frac{h}{2} f\big(x_i+\frac{1}{2}h, y_{i+\frac{1}{2}}\big)\)$

and has the Butcher Tableau, $\( \begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \\ \end{array} \)$

**Implicit trapezoid / Crank-Nicholson method**

$$ y_{n+1} = y_n + \frac{h}{2} \left( f(t_n, y_n) + f(t_{n+1}, y_{n+1}) \right) $$

with Tableau $\( \begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \\ \end{array} \)$

**Gauss-Legendre order 4 (2 stages)**

As an example of the complexity, the Gauss-Legendre order 4 method has a tableau, $
\( \begin{array}{c|cc} \frac{1}{2} - \frac{\sqrt{3}}{6} & \frac{1}{4} & \frac{1}{4} - \frac{\sqrt{3}}{6} \\ \frac{1}{2} + \frac{\sqrt{3}}{6} & \frac{1}{4} + \frac{\sqrt{3}}{6} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \\ \end{array} \)
$

such that

$$\begin{split} \begin{align*} k_1 &= f\left(t_n + \left(\frac{1}{2} - \frac{\sqrt{3}}{6}\right)h, y_n + h\left(\frac{1}{4}k_1 + \left(\frac{1}{4} - \frac{\sqrt{3}}{6}\right)k_2\right)\right) \\ k_2 &= f\left(t_n + \left(\frac{1}{2} + \frac{\sqrt{3}}{6}\right)h, y_n + h\left(\left(\frac{1}{4} + \frac{\sqrt{3}}{6}\right)k_1 + \frac{1}{4}k_2\right)\right) \end{align*} \end{split}$$

with the update for ( y_{n+1} ) is then given by:

$$ y_{n+1} = y_n + h\left(\frac{1}{2}k_1 + \frac{1}{2}k_2\right) $$

and so on.

## Systems of equations

In many engineering problems we will have a system of equations which depend on a single parameter:

\[\begin{split}\begin{align} \frac{dy_1}{dt} &= f_1(x, y_1, y_2, ..., y_j) \\ \frac{dy_2}{dt} &= f_2(x, y_1, y_2, ..., y_j) \\ \vdots \\ \frac{dy_j}{dt} &= f_j(x, y_1, y_2, ..., y_j) \\ \end{align}\end{split}\]
Happily, extension of the Runge-Kutta methods is straightforward! Collecting functions as a vector, we get the vector RK form:

$$ \vec{y}_{i+1} \approx \vec{y}_i + h \sum_{n=1}^s a_n \vec{k}_n $$

with

$$ \vec{k}_n = \vec{f}(x_i + p_n, \vec{y}_i + \sum_{m=1}^s q_{nm}\vec{k}_m) $$

## Reduction of order

Up to this point, we have only discussed first-order differential equations. Higher order differential equations can be reduced to a system of first order equations by defining new unknowns:

$$ \begin{align} y^{\prime\prime} &= f(x, y, y^\prime) \end{align} $$

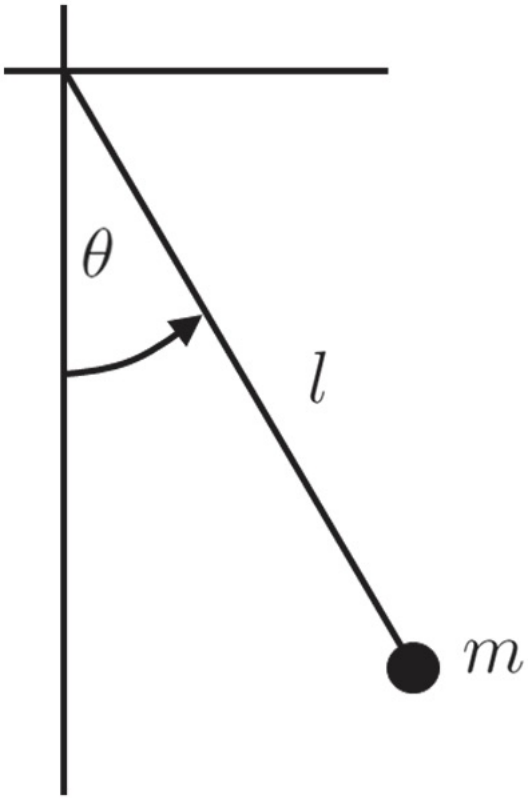
Let $z = y^\prime$, such that

$$\begin{split} \begin{align} z^{\prime} &= f(x, y, z) \\ y^{\prime} &= z \end{align} \end{split}$$

which is solved as a system of equations as above.

**Example: Swinging pendulum**

The equation of motion of a swinging pendulum is, $\( ml \frac{d^2\Theta(t)}{dt^2} = -m g \sin\big( \Theta(t) \big)\)$

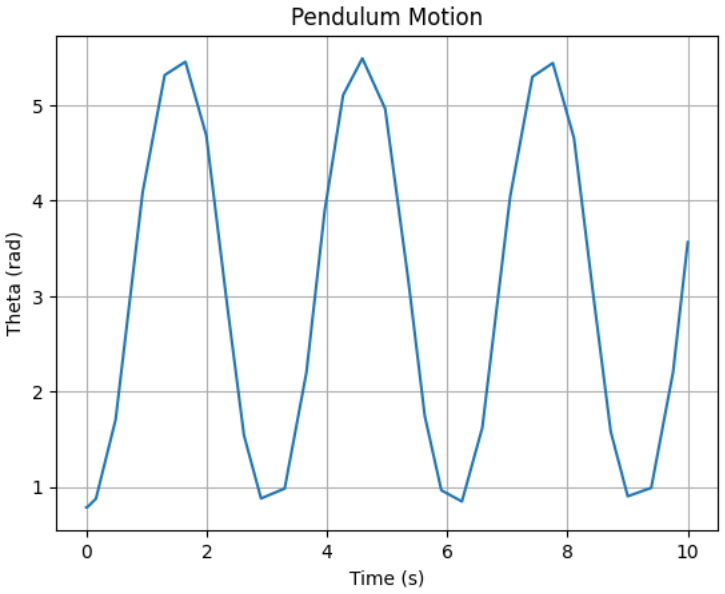



Use the reduction of order to write:

$$\begin{split} \vec{y} = \begin{bmatrix} \Theta(t) \\ \dot{\Theta}(t) \end{bmatrix} \end{split}$$

\[\begin{split}\frac{d\vec{y}}{dt} = \begin{bmatrix} \dot{\Theta}(t) \\ \ddot{\Theta}(t) \end{bmatrix} = \begin{bmatrix} y_2 \\ g \sin(y_1)/l \end{bmatrix} = \vec{f}(t, \vec{y})\end{split}\]

```
# prompt: Sovle the above with RK45

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import root
from scipy.integrate import solve_ivp

# Define the system of differential equations
def f(t, y):
  theta, theta_dot = y
  g = 9.81  # Acceleration due to gravity (m/s^2)
  l = 1.0   # Length of the pendulum (m)
  dydt = [theta_dot, (g/l) * np.sin(theta)]
  return np.array(dydt)

# Initial conditions
theta0 = np.pi/4  # Initial angle (radians)
theta_dot0 = 0.0  # Initial angular velocity (rad/s)

# Time span
t_span = (0, 10)

# Solve using RK45
sol = solve_ivp(f, t_span, [theta0, theta_dot0], method='RK45')

# Plot the results
plt.plot(sol.t, sol.y[0, :])
plt.xlabel('Time (s)')
plt.ylabel('Theta (rad)')
plt.title('Pendulum Motion')
plt.grid(True)
plt.show()
```




#Multistep methods

Multistep methods are an approach to exploit the history of solutions \(y_{\le i}\) in the calculation of the next step \(y_{i+1}\). Note that since they exploit a history, they require *bootstrapping* in order to get started.

In general we know that $
\( \begin{align} \frac{dy}{dx} &= f(x,y) \\ \int dy &= \int f(x,y) dx\\ y_{i+1} &= y_i + \int f(x,y) dx \end{align}\)$

but remember we have formulae for integration which amounts to weighted sums of the function at different points! We can then expand the right hand side and write a General Linear Multistep formula:

\[ \sum_{j=0}^{s} \alpha_j y_{i+j} = h \sum_{j=0}^{s} \beta_j f(t_{i+j}, y_{i+j}) \]

where the $\alpha_j$ and $\beta_j$ coefficients are chosen for accuracy and balance of computation. Note that this indexing is a matter of stype; in general we are interested in the final solution $y_{i+s}$, and $\alpha_s=1$ as a matter of normalization and convenience.

##Explicit linear multistep methods

When $\beta_s = 0$, the final solution, $y_i+s$ depends only on previous solutions and is therefore explicit.

This scheme is often used to generate a *predictor* based solely on what has happened before. While a good guess, it must be *corrected* using some kind of scheme like Heun's method in order to capture anything that happened during the interval.

This scheme does not handle unevenly spaced steps (a significant shortfall!), and doesn't self-start. Moreover, explicit single-step methods generally outperform these, and they are seldom used.

## Backward Difference Formulae: Implicit linear multistep methods

If we say $\beta_{j\ne s}=0$, we arrive at the implicit linear multistep method, better known as the Backward Difference Formulae which is the default for many modern computational tools.

Starting from: $\frac{dy}{dx} = f(x,y)$, the $\alpha_j$ coefficients are found from the derivative of a Lagrange interpolation polynomial fit to the distory: $<x_n, y_n> ... <x_{n+s}, y_{n+s}>$.

The first 5 orders are:

$$\begin{split} \begin{align} y_{n+1}^{(1)} &= y_n + h f(x_{n+1}, y_{n+1})\\ y_{n+1}^{(2)} &= \frac{4}{3} y_n - \frac{1}{3} y_{n-1} + \frac{2}{3} h f(x_{n+1}, y_{n+1})\\ y_{n+1}^{(3)} &= \frac{18}{11} y_n - \frac{9}{11} y_{n-1} + \frac{2}{11} y_{n-2} + \frac{6}{11} h f(x_{n+1}, y_{n+1})\\ y_{n+1}^{(4)} &= \frac{48}{25} y_n - \frac{36}{25} y_{n-1} + \frac{16}{25} y_{n-2} - \frac{3}{25} y_{n-3} + \frac{12}{25} h f(x_{n+1}, y_{n+1})\\ y_{n+1}^{(5)} &= \frac{300}{137} y_n - \frac{300}{137} y_{n-1} + \frac{120}{137} y_{n-2} - \frac{25}{137} y_{n-3} + \frac{12}{137} y_{n-4} + \frac{60}{137} h f(x_{n+1}, y_{n+1}) \end{align} \end{split}$$
Note that the Backward Euler method is BDF1.

The benefits of BDFs are:

- There is no requirement for a constant step size
- It is implicit and therefore good for stiff equations
- One can dynamically change between orders to self-start and restart if the physics change.

#Summary of initial value problems

Explicit methods:

- Easy to calculate.
- Parallelize well.
- Suffer from numerical instability.
- Require small step sizes for stiff equations.

Implicit methods:

- Computationally intensive (general require root finding / linear systems).
- Don't parallelize well.
- Are much more numerically stable.
- Can take *significantly* larger step sizes without diverging.

Systems of equations are natural and ready extensions of the methods.

Reduction of order can be applied to higher order derivatives.

Adaptive time stepping is very important and can be achieved through clever (or brute force) methods without much additional expense.

The explicit Runge-Kutta methods efficiently achieve accurate estimates if numerical instability isn't a factor.

Implicit RK methods require simultaneous solutoin of several equations which can exponenetially increase the computaitonal cost.

Explicit linear multistep methods have conditions which make them impractical.

Implicit linear multistep (Backward Differential Formulas: BDF) methods have excellent properties while exploiting the solution history.

### Dr. Mike's tips:

- Know your physics. If you don't, go with an adaptive BDF method (the default of most software).
- If you know your system is not stiff, RK45 is the go-to and you will substantially benefit in computer time.
- If you know your physics has abrupt changes (e.g.: steps / pulses) consider keeping with Backward Euler - no higher-order accuracy is possible.

## Boundary value problems

Boundary value problems are ordinary differential equations where information is known on the boundaries of the domain. E.g.:

$$\frac{d^2 f}{dt^x} +\frac{df}{dx} = 1, \quad f(x=0) = 1000, f(x=1) = 1$$

One option is to *shoot* a solution from one side using our IVP methods and check if it hits the other boundary, which is called the *shooting method*.

Alternately, we can also exploit the fact that we have information at both sides to solve a discretized representation of the solution simultaneously.

### The Shooting method

The shooting method solves boundary values problems using the algorithms we developed for initial value problems, including all the consideration we made for adaptive stepping, stiffness, and high order approximation.

The concept is simple:

1. Choose an *initial* boundary
2. Solve as an IVP to find the function value at the other boundary.
3. Check if the other boundary condition is met.
4. Wrap 1-3 in a root finding algorithm to find the solution.

### Pros / Cons

**Benefits**

- Conceptually Simple: Reduces a boundary value problem to an initial value problem, which is often simpler to solve.
- Leverages Initial Value Solvers: Allows the use of robust and well-tested initial value problem (IVP) solvers, such as Runge-Kutta methods, which are readily available in many software libraries.
- Good for Linear Problems: Can be particularly effective for linear or mildly nonlinear problems where the solution does not vary drastically with initial conditions.

- Flexible for Adjustments: Easily adaptable to different types of BVPs by adjusting the shooting parameters to meet boundary conditions at the other end of the domain.
- Reduced Complexity in Low Dimensions: For low-dimensional systems, it often involves fewer computations and is easier to set up compared to other methods like finite difference or collocation.

**Drawbacks**

- Nonlinear and Sensitive to Initial Guesses: For nonlinear problems, the solution can be highly sensitive to initial guesses of the shooting parameters, potentially leading to divergence or non-convergence.
- Difficulty with Complex or Oscillatory Solutions: Struggles with problems where the solution exhibits rapid changes, oscillations, or sensitivity to initial conditions, as small errors can propagate.
- Limited Effectiveness in High Dimensions: Becomes computationally expensive and less effective for high-dimensional systems or systems with multiple boundary conditions.
- Requires Numerical Root-Finding: Often necessitates a root-finding algorithm (e.g., Newton's method) to adjust initial guesses to match boundary conditions, adding an additional layer of complexity.
- Potential for Numerical Instability: Errors can accumulate over the integration interval, leading to instability, especially in stiff ODEs or systems with sensitive boundary conditions.

## Example: Ballistics

NB: Ballistic targetting was likely one of the the original motivations for these tools!

We are launching a rocket, and need it to be 50m altitude after 5 seconds. Ignoring aerodynamic drag, what should the initial speed be?

Answer: This is a 1D problem for altitude as a function of time, $y(t)$. Given gravity is $-9.8 m/s^2$ the equation of motion is:

$$ \frac{\partial^2 y}{\partial t^2} = -g, \quad y(0) = 0, y(5) = 50 \]

Rewrite this using reduction of order:

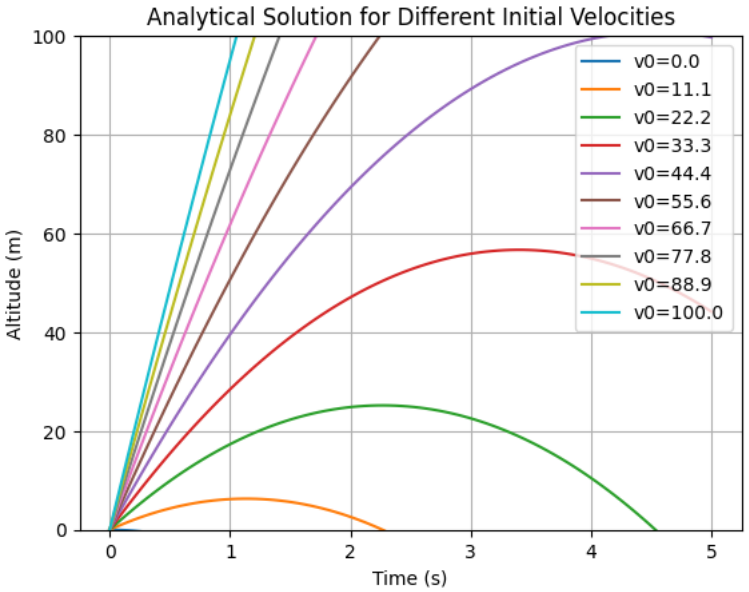
$$\begin{split}\begin{align} \frac{\partial y}{\partial t} &= v \\ \frac{\partial v}{\partial t} &= -g \\ \end{align}\end{split}\]

```
# prompt: solve the above system with the solve_ivp method

import numpy as np
from scipy.integrate import solve_ivp, solve_bvp
from scipy.optimize import root

def model(t, y):
  y, v = y
  g = -9.8  # Acceleration due to gravity
  dydt = v
  dvdt = g
  return [dydt, dvdt]

# Initial condition for altitude (y)
y0 = 0

# Define the time span
t_span = [0, 5]

# Define the boundary condition for altitude at the end time
y_end = 50

# Implement the shooting method
def shooting_method(v0):
  y_initial = [y0, v0[0]]
  sol = solve_ivp(model, t_span, y_initial, method='RK45')
  return sol.y[0][-1]


# Find the root for the shooting method
result = root(lambda v0: shooting_method(v0) - y_end, 1) # Initial guess for v0
print(result)
v0 = result.x[0]  # The calculated initial velocity

# Solve the IVP with the found initial velocity
y_initial = [y0, v0]
sol = solve_ivp(model, t_span, y_initial, method='RK45')

# Print the solution
print(f"Initial velocity (v0): {v0:.2f} m/s")
print(f"Altitude at t=5s: {sol.y[0][-1]:.2f} m")
```

```
 message: The solution converged.
 success: True
  status: 1
     fun: 2.1316282072803006e-14
       x: [ 3.450e+01]
  method: hybr
    nfev: 4
    fjac: [[-1.000e+00]]
       r: [-5.000e+00]
     qtf: [-3.652e-10]
Initial velocity (v0): 34.50 m/s
Altitude at t=5s: 50.00 m
```

Look at the analytical solution: $\(y(t) = v0 t - (g t^2)/+2\)$

```python
# prompt: plot the analytical asolution with v0 from 0 to 100

import matplotlib.pyplot as plt
import numpy as np

# Analytical solution
g = -9.8
t = np.linspace(0, 5, 100)
v0_values = np.linspace(0, 100, 10)

for v0 in v0_values:
  y_analytical = v0 * t + (g * t**2) / 2
  plt.plot(t, y_analytical, label=f'v0={v0:.1f}')


plt.xlabel('Time (s)')
plt.ylabel('Altitude (m)')
plt.title('Analytical Solution for Different Initial Velocities')
plt.legend()
plt.ylim(0, 100)
plt.grid(True)
plt.show()
```




## Collocation algorithms

Yet another method involves invoking our interpolator methods to model the solution directly. The process is:

1. Determine a set of (possibly unevenly spaced) points, \(x_i\).
2. Define an interpolator function with *unknown* parameters \(w_i\).
3. Apply the ODE at each point and find a residual, \(R(w_i) = y^\prime(w_i) - f\big(x_i, y_i(w_i)\big)\)
4. Solve (root find or in least squares) for the interpolator parameters.

We can apply a number of interpolators to this scheme, but some common ones include:

- Splines (1D / 2D)
- Radial Basis Functions (ND)
- Neural Networks
- Spectral methods

Spectral methods in this context refers to the use of special basis functions (Sin / cos, Chebychev, Legenre polynomials, etc) with associated optimal collocation points.

## Pros / Cons

**Benefits:**

- Flexibility: Can handle complex geometries and irregular domains.
- High Accuracy: Often provides high accuracy with fewer collocation points, especially with higher-order polynomials.
- Versatility: Suitable for a wide range of problems, including ODEs and PDEs.
- Spectral Methods: Achieve exponential convergence for smooth problems when using spectral collocation (e.g., Chebyshev or Legendre polynomials).

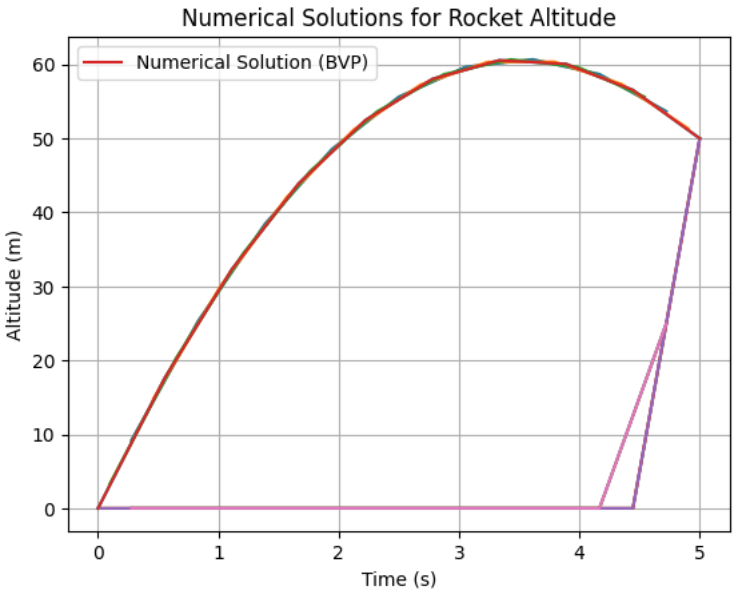
**Drawbacks:**

- Complexity: More complex to implement compared to FDM, especially for higher-order methods.
- Computational Cost: Can be computationally expensive due to the need to solve large systems of equations.
- Stability: Requires careful selection of collocation points to ensure stability and convergence.

**Example: Splines**

$$ \frac{\partial^2 y}{\partial t^2} = -g, \quad y(0) = 0, y(5) = 50 $$

```
def fun(x, y):
  plt.plot(x,y[0])
  return np.vstack((y[1], -9.8*np.ones_like(y[1])))

def bc(ya, yb):
  return np.array([ya[0], yb[0] - 50])

x = np.linspace(0, 5, 10)
y = np.zeros((2, x.size))
y[0, 0] = 0
y[0, -1] = 50
sol = solve_bvp(fun, bc, x, y)

# Plot the solution from solve_bvp
plt.plot(sol.x, sol.y[0], label='Numerical Solution (BVP)')
plt.xlabel('Time (s)')
plt.ylabel('Altitude (m)')
plt.title('Numerical Solutions for Rocket Altitude')
plt.legend()
plt.grid(True)
plt.show()
```



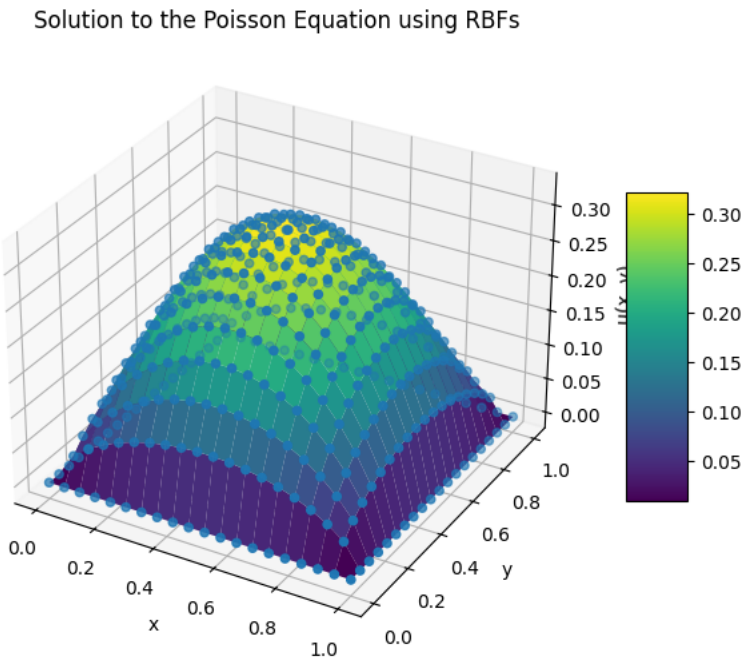
**Example: RBFs**

Use RBFs to solve

$$\nabla^2 u = 4$$

on the unit square where $u(x=0,1,y) = u(x, y=0,1) = 0$.

```
import numpy as np
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt
from matplotlib import cm

# Parameters
N = 20  # Number of nodes in each direction (NxN grid)
epsilon = N/np.sqrt(2)  # Shape parameter for RBFs

# Define the RBF (we use a Gaussian RBF in this example)
def rbf(r, epsilon):
    return np.exp(-(epsilon * r) ** 2)

# Define the Laplacian of the RBF (for the Poisson equation)
def laplacian_rbf(r, epsilon):
    return (4 * epsilon**2 - 4 * epsilon**4 * r**2) * np.exp(-(epsilon * r) ** 2)

# Generate nodes in a square domain [0,1]x[0,1]
x = np.linspace(0, 1, N)
y = np.linspace(0, 1, N)
X, Y = np.meshgrid(x, y)
points = np.vstack([X.ravel(), Y.ravel()]).T

#points = np.random.uniform(0, 1, (N**2, 2))

# Compute pairwise distance matrix
r = cdist(points, points)

# Build the RBF matrix and Laplacian matrix
A = rbf(r, epsilon)
L = laplacian_rbf(r, epsilon)

# Define right-hand side (RHS) for Poisson equation
rhs = np.full(points.shape[0], 4.0)

# Apply boundary conditions (u=0 on the boundary)
boundary_indices = np.where((points[:, 0] == 0) | (points[:, 0] == 1) |
                            (points[:, 1] == 0) | (points[:, 1] == 1))[0]
interior_indices = np.setdiff1d(np.arange(points.shape[0]), boundary_indices)

# Modify RHS and matrices to incorporate Dirichlet BCs
rhs[boundary_indices] = 0
A[boundary_indices, :] = 0
A[boundary_indices, boundary_indices] = 1
L[boundary_indices, :] = 0
L[boundary_indices, boundary_indices] = 1

# Solve for the weights
weights = np.linalg.solve(L, rhs)

# Calculate solution u as a weighted sum of RBFs
u = A @ weights

# Reshape solution to 2D grid
U = u.reshape(N, N)

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, U, cmap=cm.viridis)
ax.scatter(points[:,0], points[:,1], u)
fig.colorbar(surf, shrink=0.5, aspect=5)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("u(x, y)")
ax.set_title("Solution to the Poisson Equation using RBFs")
plt.show()
```




Open in Colab

Goal

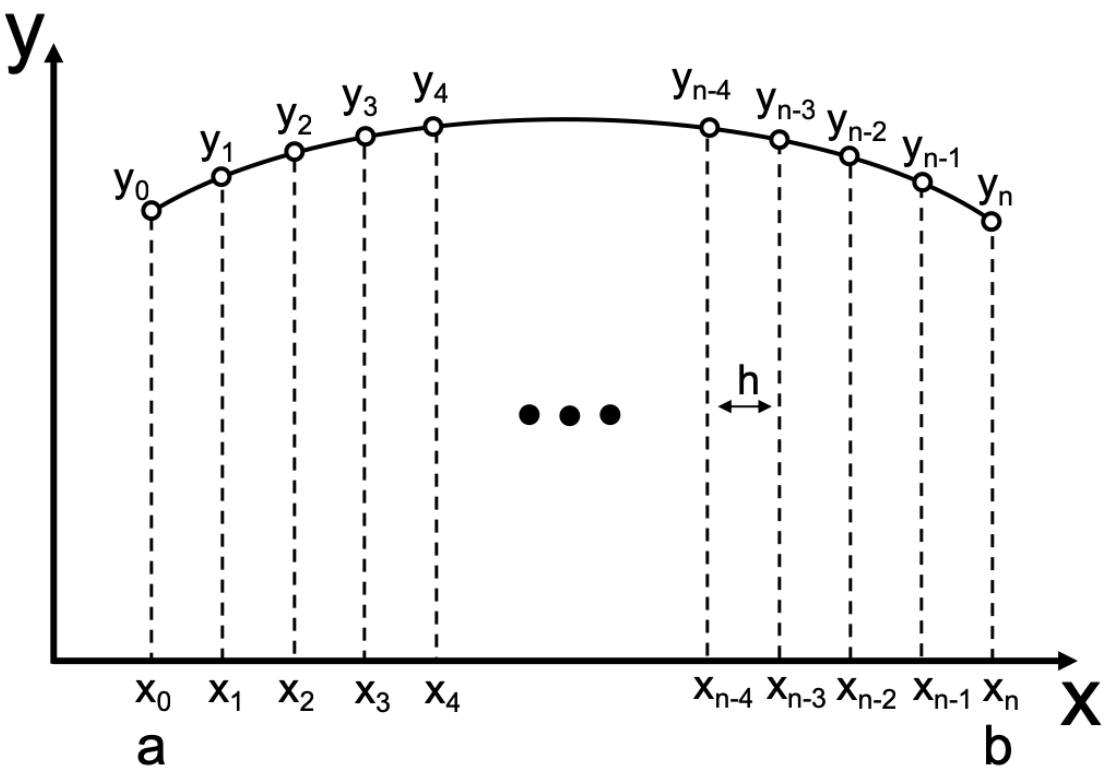
- Apply the finite difference method to solve steady state differential equations
- Apply linear system solution methods to real applications
- Be able to implement Dichilet and Neumann boundary conditions

## The Finite Difference method

In the past lecture, we saw how to write derivatives in terms of the function evaluated at discrete points. Let's use it and see what we've been building up to.

Lets take an unkown function \(f(x)\) and *discretize it* by sampling it at a set of coordinates. We saw that finite difference benefits from equal spacing, so lets sample it with an even step size \(h\).

For a function \(y(x)\) we would get:




Finite difference allows us to express the derivatives of \(y(x)\) in terms of the vector elements. For step size \(h\), we have:

| Finite Difference Approximation | Formula in terms of \(y_i\) | Order of Accuracy |
|---|---|---|
| Forward difference (1st derivative) | \(y'(x_i) \approx (y_{i+1} - y_i) / h\) | O(h) |
| Backward difference (1st derivative) | \(y'(x_i) ≈ (y_i - y_{i-1}) / h\) | O(h) |
| Central difference (1st derivative) | \(y'(x_i) ≈ (y_{i+1} - y_{i-1}) / (2h)\) | $O(h^2)$ |
| Forward difference (2nd derivative) | \(y''(x_i) ≈ \frac{y_{i+2} - 2y_{i+1} + y_i} {h^2}\) | O(h) |
| Backward difference (2nd derivative) | \(y''(x_i) ≈ (y_i - 2y_{i-1} + y_{i-2}) / h^2\) | O(h) |
| Central difference (2nd derivative) | \(y''(x_i) ≈ (y_{i+1} - 2y_i + y_{i-1}) / h^2\) | $O(h^2)$ |

The finite difference method applied to the discretized function lets us write the differential equation in terms neighbouring values (or the boundary conditions)!

### Example: Rocket tragectory

We are going out to launch a rocket, and let y(t) be the altitude (meters from the surface) of the rocket at time t. We know the gravity \(g=9.8m/s^2\).

If we want to have the rocket at 50 m off the ground 5 seconds after launching, what should be the velocity at launching? (we ignore the drag of the air resistance).

The ODE is

\[\begin{split} \begin{align} \frac{d^2y}{dt^2} &= -g \\ \frac{y_{i+1} - 2y_{i} + y_{i-1}} {h^2} &= -g \end{align} \end{split}\]

This expression only applies in *interior* points. We must use the boundary conditions \(y(0) = 0\) and \(y(5) = 50\).

Let's take $n=10$. Since the time interval is $[0, 5]$ and we have $n=10$, therefore, $h=0.5$. Using the finite difference approximated derivatives, we have

$$y_0 = 0$$
$$y_{i-1} - 2y_i + y_{i+1} = -gh^2, \;i = 1, 2, ..., n-1$$
$$y_{10} = 50$$

This looks awfully familiar…

It's just our familiar linear system!

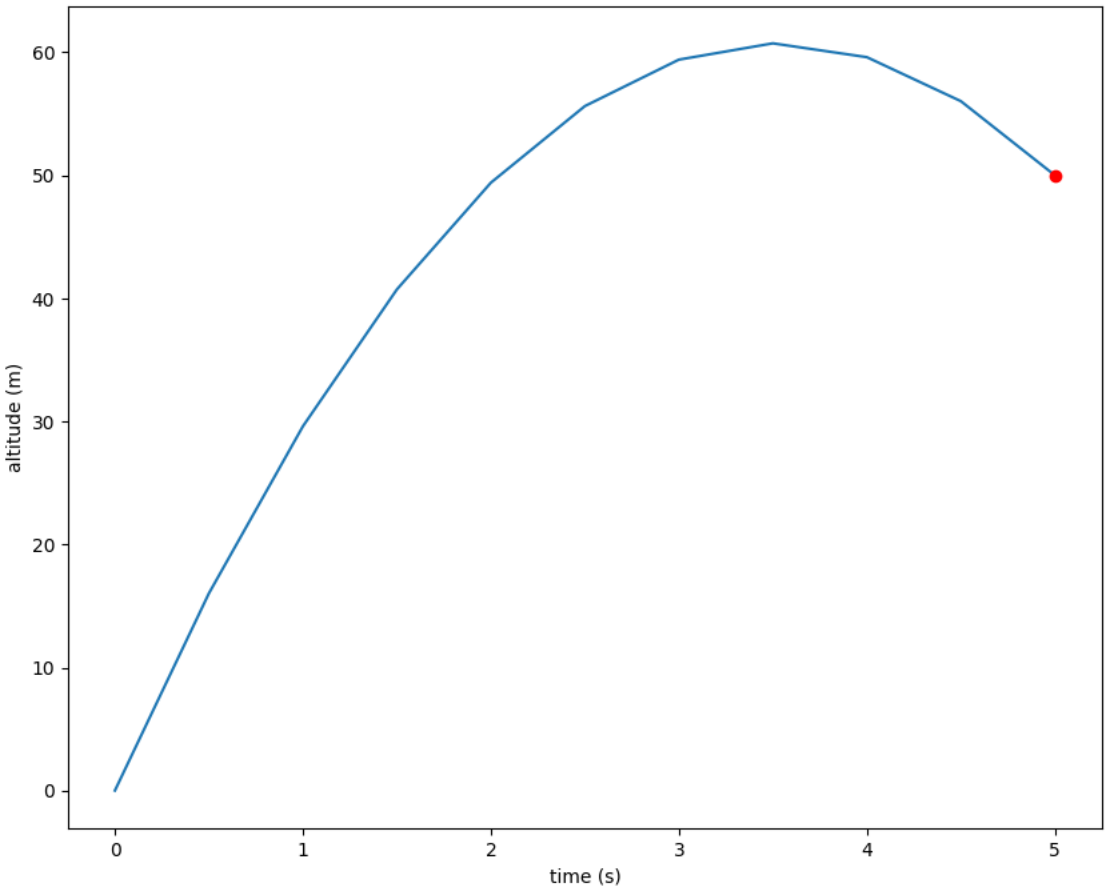
$$\begin{split}\begin{bmatrix} 1 & 0 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1& -2& 1 \\ & & & &1 \end{bmatrix}\left[\begin{array}{c} y_0 \\y_1 \\ ... \\ y_{n-1}\\y_n \end{array}\right] = \left[\begin{array}{c} 0 \\-gh^2 \\ ... \\ -gh^2 \\50\end{array}\right]\end{split}$$

and how do we solve linear systems?!?

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

n = 10

def get_A_and_b(n):
  h = (5.-0) / n

  # Get A
  A = np.zeros((n+1, n+1))
  A[0, 0] = 1
  A[n, n] = 1
  for i in range(1, n):
      A[i, i-1] = 1
      A[i, i] = -2
      A[i, i+1] = 1

  # Get b
  b = np.zeros(n+1)
  b[1:-1] = -9.8*h**2
  b[-1] = 50

  return A,b
```

```
n = 10
A,b = get_A_and_b(n=10)

print(A)
print(b)

# solve the linear equations
y = sp.linalg.solve(A, b)

print()

t = np.linspace(0, 5, n+1)
plt.figure(figsize=(10,8))
plt.plot(t, y)
plt.plot(5, 50, 'ro')
plt.xlabel('time (s)')
plt.ylabel('altitude (m)')
plt.show()
```

```
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 1. -2.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1. -2.  1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1. -2.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1. -2.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1. -2.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1. -2.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  1. -2.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1. -2.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1. -2.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]]
[ 0.   -2.45 -2.45 -2.45 -2.45 -2.45 -2.45 -2.45 -2.45 -2.45 50.  ]
```




For the velocity, \(y'(0)\) we can calculate using the forward difference:

```
(y[1] - y[0])/(5/n)
```

```
32.05
```

which is pretty good compared to the analytic answer 34.5 .

### Example: Neuman boundary condition

Using finite difference method to solve the following linear boundary value problem:

\[ y''(x) = -4y(x) +4x \]

with the boundary conditions: \(y(0) = 0\) and \(y'(\pi/2)=0\).

The Dirichlet boundary condition is simple: $\( y_0 = 0\)$

The interior points are a little more complex. We need to move all the terms with the dependant quantity (\(y\)) to the left and everything else (including \(x\) terms) to the right.

\[\begin{split}\begin{align} y''(x) &= -4y(x) +4x \\ \frac{y_{i-1} - 2 y_i + y_{i+1}}{h^2} +4y_i &= 4x_i \\ y_{i-1} - 2y_i + y_{i+1} + 4 h^2 y_i&= -4 h^2 x_i \end{align} \end{split}\]
for \(i = 1, 2, ..., n-1\)

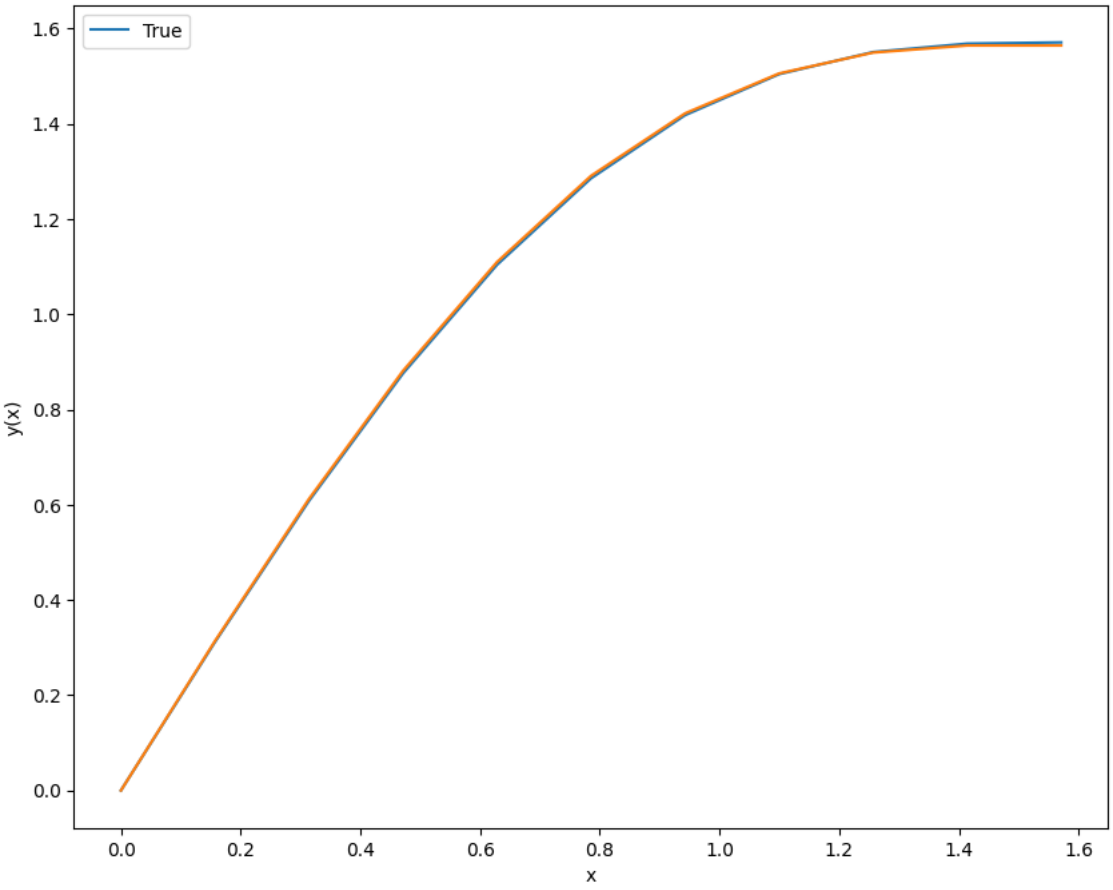
But what should we do for the boundary \(y'(\pi/2) = 0\)?

We can actually just write it out exactly in finite difference:

\[\begin{split}\begin{align} \frac{y_i - y_{i-1}}{h} &= 0 \\ y_i - y_{i-1} = 0 \end{align}\end{split}\]
Using matrix notation, we have:

\[\begin{split}\begin{bmatrix} 1 & 0 & & & \\ 1 & -2+4h^2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1& -2+4h^2 & 1 \\ & & & -1 & 1 \end{bmatrix}\left[\begin{array}{c} y_0 \\y_1 \\ ... \\ y_{n-1}\\y_n \end{array}\right] = \left[\begin{array}{c} 0 \\4h^2x_1 \\ ... \\ 4h^2x_{n-1} \\4h^2x_{n}\end{array}\right]\end{split}\]
Lets see what we get compared to the analytic answer \(y=x+0.5 \sin2x\)

```
# prompt: Solve the above linear system and plot y vs x

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt

n = 10

def get_A_and_b(n):
  h = (np.pi/2 - 0) / n

  # Get A
  A = np.zeros((n+1, n+1))
  A[0, 0] = 1
  A[n, n-1] = -1
  A[n, n] = 1
  for i in range(1, n):
      A[i, i-1] = 1
      A[i, i] = -2 + 4*h**2
      A[i, i+1] = 1

  # Get b
  b = np.zeros(n+1)
  for i in range(1, n):
      b[i] = 4*h**2*(i*h)

  return A,b

A,b = get_A_and_b(n=10)

# solve the linear equations
y = sp.linalg.solve(A, b)

y_true = lambda x: x + .5*np.sin(2*x)

x = np.linspace(0, np.pi/2, n+1)
plt.figure(figsize=(10,8))
plt.plot(x, y_true(x), label="True")
plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y(x)')
plt.legend()
plt.show()
```




**Higher order accuracy boundary**

Alternately, as a bit of a trick to gain accuracy, the Neuman condition at the boundary can be considered with the central difference for \(y_n\) (despite \(y_{n+1}\) not existing!):

\[\begin{split} \begin{align} \frac{y_{n+1}-y_{n-1}}{2h} &= 0\\ y_{n+1} &= y_{n-1} \end{align} \end{split}\]

We substitute this into the equation for the interior points to get:

\[ 2y_{n-1}-2y_n -h^2(-4y_n+4x_n)=0\]

which should have second order accuracy.

Now we have

\[\begin{split}\begin{bmatrix} 1 & 0 & & & \\ 1 & -2+4h^2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1& -2+4h^2 & 1 \\ & & & 2 &-2+4h^2 \end{bmatrix}\left[\begin{array}{c} y_0 \\y_1 \\ ... \\ y_{n-1}\\y_n \end{array}\right] = \left[\begin{array}{c} 0 \\4h^2x_1 \\ ... \\ 4h^2x_{n-1} \\4h^2x_{n}\end{array}\right]\end{split}\]

Lets compare this solution vs the previous formula in terms of \(y'(\pi/2)\)

```
def get_A_and_b_2(n):
  h = (np.pi/2 - 0) / n

  # Get A
  A = np.zeros((n+1, n+1))
  A[0, 0] = 1
  A[n, n] = -2+4*h**2
  A[n, n-1] = 2
  for i in range(1, n):
      A[i, i-1] = 1
      A[i, i] = -2 + 4*h**2
      A[i, i+1] = 1

  # Get b
  b = np.zeros(n+1)
  for i in range(1, n+1):
      b[i] = 4*h**2*(i*h)

  return A,b

print('Old')
for n in range(10,30,5):
  # Original implementation:
  A,b = get_A_and_b(n)
  y = sp.linalg.solve(A, b)
  print(y[-1])

print('New')
# New boundary condition
for n in range(10,30,5):
  A,b = get_A_and_b_2(n)
  y = sp.linalg.solve(A, b)
  print(y[-1])

print('True')
print(y_true(np.pi/2))
```

```
Old
1.5641951194959582
1.5678968928038604
1.5691723081328415
1.569759027706592
New
1.5641814012620596
1.5678951310352263
1.5691718937395551
1.569758892476149
True
1.5707963267948966
```

The higher order implementation is more accurate (for the same computational complexity), but the step size still domainates the quality of the answer.

### Example 3: Nonlinear systems

Solve the boundary value problem,

$$ \begin{align} y'' &= -3 y y' \end{align} $$

with $y(0) = 0$ and $y(2) = 1$.

The Dirichlet boundary conditions are easy, and the interior points are given by: $
$ \begin{align} \frac{y_{i-1} -2 y_i +y_{i+1}}{h^2} &= -3 y_i \bigg[ \frac{y_{i+1} - y_{i-1}}{2h} \bigg] \end{align} $$

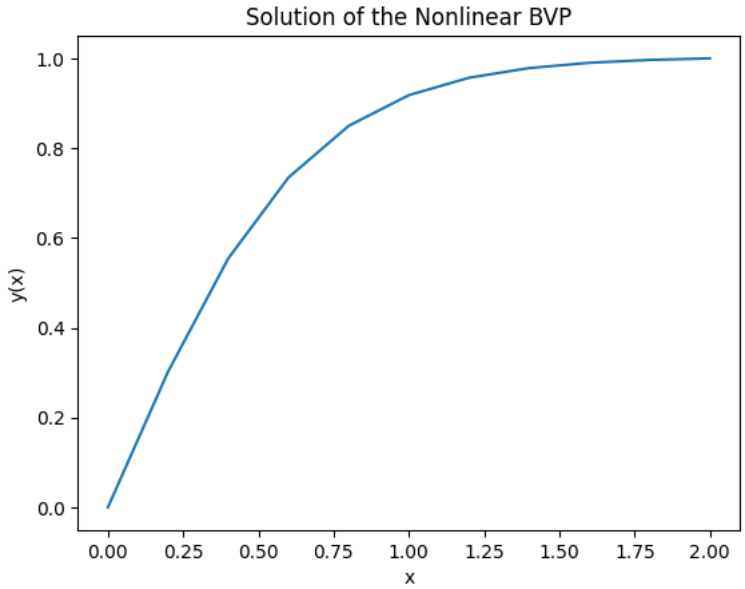
How do we solve this?

Since this system is *nonlinear* we will need to use a *nonlinear solver* which is a root finder! In particular, we need to find the root of the residual,

$$\begin{split} \begin{align} y_{i-1} -2 y_i +y_{i+1} + 3 h y_i [y_{i+1} - y_{i-1}] &=0 \\ \vec{R}(\vec{y}) &= \vec{0} \end{align} \end{split}$$

```
from scipy.optimize import root

def residual(y, n, h):
  r = np.zeros(n + 1)
  r[0] = y[0]  # y(0) = 0
  r[-1] = y[-1] - 1  # y(2) = 1
  for i in range(1, n):
    r[i] = y[i - 1] - 2 * y[i] + y[i + 1] + 3 * h * y[i] * (y[i + 1] - y[i - 1]) / 2
  return r

n = 10  # Number of grid points
y0 = np.linspace(0, 1, n + 1)  # Initial guess

# Solve the nonlinear system using scipy.optimize.root
sol = root(residual, y0, args=(n,2. / n, ))
print(sol)
y = sol.x

# Plot the solution
x = np.linspace(0, 2, n + 1)
plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y(x)')
plt.title('Solution of the Nonlinear BVP')
plt.show()
```

```
 message: The solution converged.
 success: True
  status: 1
     fun: [-3.711e-26  2.691e-11 -9.336e-11  1.270e-10 -4.548e-12
           -1.767e-10  1.645e-10 -2.525e-11 -5.023e-11  3.169e-11
            0.000e+00]
       x: [-3.711e-26  3.024e-01  5.545e-01  7.347e-01  8.498e-01
            9.181e-01  9.570e-01  9.785e-01  9.902e-01  9.966e-01
            1.000e+00]
  method: hybr
    nfev: 21
    fjac: [[-7.178e-01 -6.963e-01 ...  2.602e-17  0.000e+00]
           [ 5.736e-01 -5.914e-01 ...  2.989e-03  0.000e+00]
           ...
           [ 1.560e-02 -1.609e-02 ... -8.190e-01  0.000e+00]
           [ 0.000e+00 -0.000e+00 ...  0.000e+00 -1.000e+00]]
       r: [-1.393e+00  1.342e+00 ... -1.040e+00 -1.000e+00]
     qtf: [-2.429e-10  5.987e-10 -2.227e-09  3.264e-10 -1.230e-09
            2.292e-09  2.220e-09 -3.106e-09 -8.640e-10  5.212e-10
            0.000e+00]
```




### Example 4: Multidimensional

Solve the discrete Poisson equation on the unit square,

\(-\Delta u = 1\) on \(\Omega=[0,1]^2\)

with

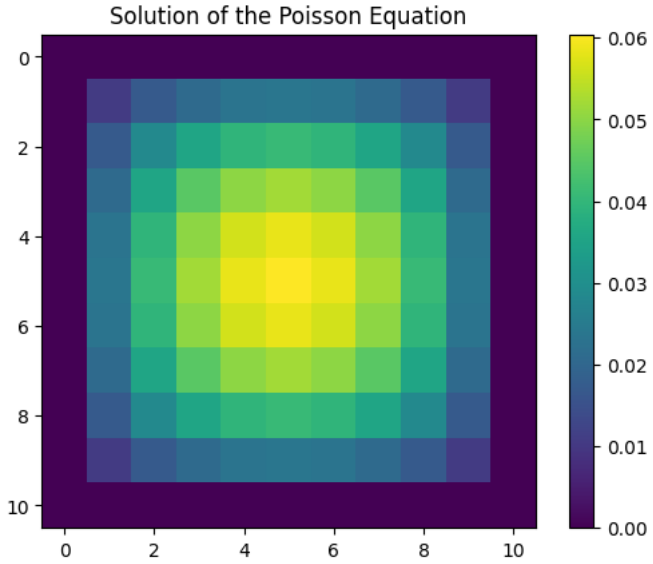
\(u=0\) on \(\partial \Omega\)

The interior points are given by \(-\Delta u = 1\) on \(\Omega=[0,1]^2\)

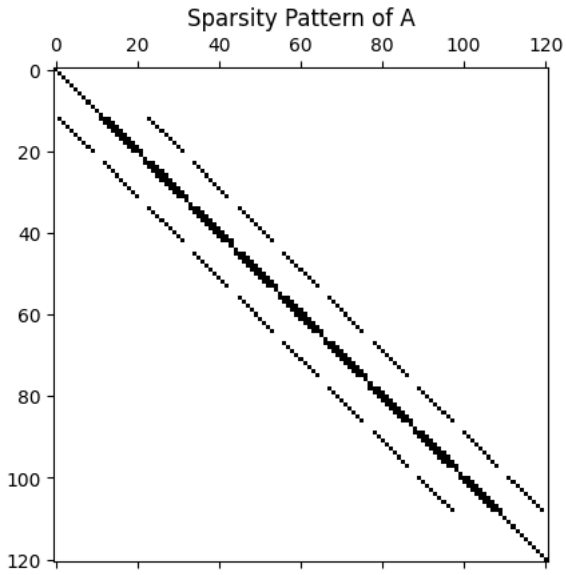
For central difference on a 2D grid with spacing h in both x and y directions, the discrete Laplacian at a point (i,j) is:

\[(u(i+1,j) + u(i-1,j) + u(i,j+1) + u(i,j-1) - 4u(i,j)) / h^2=1\]

```
# prompt: write a code to generate hte matrix u from the equation above with the boundary

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from scipy.optimize import root

def get_A_and_b_for_poisson(n):
  h = 1. / (n + 1)
  A = np.zeros(((n + 1)**2, (n + 1)**2))
  b = np.zeros((n + 1)**2)

  for i in range(n + 1):
    for j in range(n + 1):
      if i == 0 or i == n or j == 0 or j == n:  # Boundary condition
        row_index = i * (n + 1) + j
        A[row_index, row_index] = 1
      else:  # Interior point
        row_index = i * (n + 1) + j
        A[row_index, row_index] = -4
        A[row_index, (i - 1) * (n + 1) + j] = 1
        A[row_index, (i + 1) * (n + 1) + j] = 1
        A[row_index, i * (n + 1) + (j - 1)] = 1
        A[row_index, i * (n + 1) + (j + 1)] = 1
        b[row_index] = -h**2

  return A, b


n = 10  # Number of grid points in each direction
A, b = get_A_and_b_for_poisson(n)


u = np.linalg.solve(A, b)

# Reshape the solution into a 2D array
u_matrix = u.reshape((n + 1, n + 1))

# Plot the solution
plt.imshow(u_matrix, cmap='viridis')
plt.colorbar()
plt.title('Solution of the Poisson Equation')
plt.show()
```




```
# prompt: Plot the sparsity pattern of A

plt.spy(A)
plt.title('Sparsity Pattern of A')
plt.show()
```




#Summary

**Benefits**

1. **Simplicity**: The finite difference method is relatively easy to understand and implement, making it accessible for many users.
2. **Efficiency**: It can be computationally efficient, especially for problems with simple geometries and boundary conditions.
3. **Structured Grids**: Works well with structured grids, which can simplify the discretization process.
4. **High-Order Approximations**: It is possible to obtain high-order approximations, which can improve the accuracy of the solution.

**Drawbacks**

1. **Limited Flexibility**: The method is less flexible compared to other methods like finite element or finite volume methods, particularly for complex geometries.
2. **Stability Issues**: Finite difference methods can suffer from stability issues, especially for certain types of partial differential equations.
3. **Boundary Conditions**: Handling complex boundary conditions can be challenging.
4. **Accuracy**: The accuracy of the method can be limited by the discretization error and round-off error.

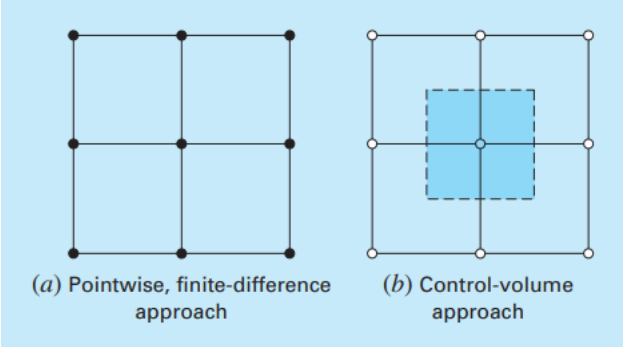

## Finite volume methods

The Finite Volume Method (aka the control-volume / volume-integral approach) is a technique that focusses on the *integral* of the quantity in a *control volume* surrounding a point.

This approach has advantages for conservation equations and different convergence properties for advective systems, e.g.: \(\frac{\partial u}{\partial t} + c \frac{\partial u}{\partial x} = 0\) and is common in computational fluid dynamics codes.

> Contrast this with the Finite Difference Method which focusses on writing derivatives in terms of relations to the surrounding points.

The control volume is constructed algorithmically through, e.g.: bisection of the lines joining nodes.




Many physics equations contain conservative term (divergence of a vector field \(\vec{J}\)). The integral of these terms can be transformed through Green's law (divergence theorem) into surface term:

\[\int_V \nabla \cdot \vec{J} dV = \oint\vec{J}\cdot \hat{n} dS\]

Convervation equations are very common in physics. When advection is present, the system can feature sharp gradients leading to numerical instabilities. Since Finite Volume focusses on the integral quantity (rather approximating the derivatives) it is more robust to these instabilities.

Some examples of conservation equations are:

1. **Conservation of Mass**
   - **Equation**: \( \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \)
   - **Conserved Term**: Mass density ( \rho )
   - **Description**: Mass is conserved in a closed system, meaning it cannot be created or destroyed.
2. **Conservation of Momentum**
   - **Equation**:
     \( \frac{\partial (\rho \mathbf{v})}{\partial t} + \nabla \cdot (\rho \mathbf{v} \otimes \mathbf{v} - \mathbf{\tau}) = \mathbf{F} \)
   - **Conserved Term**: Linear momentum \( \rho \mathbf{v} \)
   - **Description**: The total momentum of a system remains constant unless acted on by external forces.
3. **Conservation of Energy**

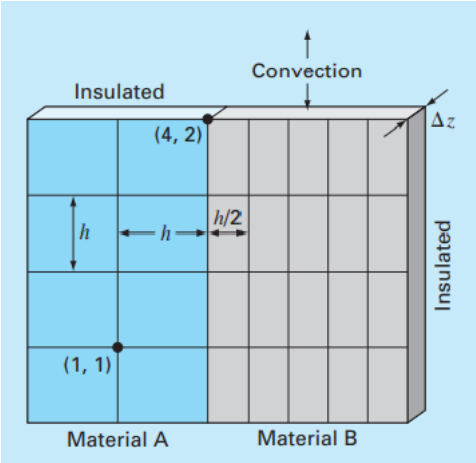
- **Equation**: $\frac{\partial E}{\partial t} + \nabla \cdot (E \mathbf{v} + \lambda \nabla T) = \mathbf{v} \cdot \mathbf{F}$
- **Conserved Term**: Total energy ( E ) (including kinetic, potential, and internal energy)
- **Description**: Energy within a closed system remains constant over time.

4. **Conservation of Electric Charge**
   - **Equation**: $\frac{\partial \rho_e}{\partial t} + \nabla \cdot \mathbf{J} = 0$
   - **Conserved Term**: Electric charge ( \rho_e )
   - **Description**: Electric charge is conserved, meaning it cannot be created or destroyed.
5. **Conservation of dilute species**
   - **Equation**: $\frac{\partial n_i}{\partial t} + \nabla \cdot (n_i \mathbf{v} + D_i \nabla \mu_i) = \dot{Q}$
   - **Conserved Term**: Abundance of species $i$ $n_i$
   - **Description**: Species are conserved through convective and diffusive transport.

The Finite volume method is also more flexible accounting for irregular / unstructured grids and differing materials again due to the focus on the change in integrated quantity through its surface.

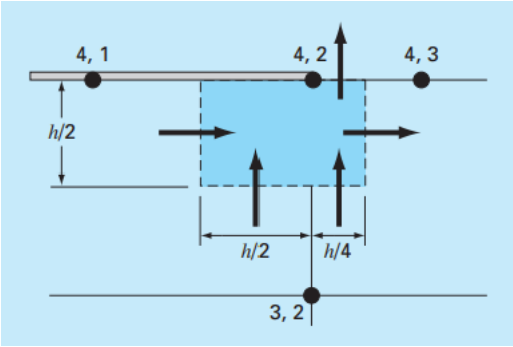
## Example: Finite volume expression

Write the finite volume expression for node (4,2) for a plate with uneven mesh spacing in differing materials.




Consider the steady state heat equation: $$ -\nabla \cdot \vec{J} = 0$$ where $\vec{J} = -\lambda \nabla T$ is the heat flux.

The control volume is around (4,2) is depicted:




The sum of fluxes is:

$$0 = J_{left} \frac{h}{2}\Delta z - J_{right} \frac{h}{2} \Delta z + J_{lower\ a} \frac{h}{2} \Delta z + J_{lower\ b} \frac{h}{4} \Delta z - J_{convection} \frac{h}{4} \Delta z$$

with

\[\begin{split}\begin{align} J_{left} &= - k_a \frac{T_{42} - T_{41}}{h} \\ J_{right}&= - k_b \frac{T_{43} - T_{42}}{\frac{h}{2}} \\ J_{lower\ a} &= - k_a \frac{T_{42} - T_{32}}{h} \\ J_{lower\ b} &= - k_b \frac{T_{42} - T_{32}}{h} \\ J_{conv} & = h \left(T_{amb} - T_{42} \right) \end{align} \end{split}\]
Substituting and simplifying we get:

\[ 0 = \Delta z \left( -\frac{k_a}{2} (T_{42} - T_{41}) - k_b (T_{43} - T_{42}) - \frac{k_a}{2} (T_{42} - T_{32}) - \frac{k_b}{4} (T_{42} - T_{32}) - \frac{h^2}{4} (T_{amb} - T_{42}) \right). \]
which is an algabreic expression for \(T_{42}\), \(T_{32}\), \(T_{41}\). Once constructed over all the nodes we build a linear system!

Note that this look suspitiously familiar to finite difference! It is exactly finite difference for certain cases (regular square grids).

**Benefits**

- **Flexibility**: Can handle complex geometries and irregular domains.
- **Conservation**: Inherently conserves quantities like mass, momentum, and energy.
- **Unstructured Meshes**: Works well with both structured and unstructured meshes.
- **Boundary Conditions**: Can apply boundary conditions non-invasively.
- **Robustness**: Provides numerical robustness through discrete maximum (minimum) principles.

**Drawbacks**

- **Complexity**: Implementation can be more complex compared to simpler methods like finite difference.
- **Computational Cost**: Can be computationally expensive due to the need to solve large systems of equations.
- **Accuracy**: May require fine meshes to achieve high accuracy, increasing computational cost.
- **Numerical Diffusion**: Can introduce numerical diffusion, which may smear out sharp gradients.
- **Stability**: Requires careful consideration of stability criteria, such as the CFL condition.

# Partial differential equations

Partial differential equations expand ODEs to include multiple independant variables (coordinates).

- Other *spatial* coordinates are typically extensions of the ODE case and are usually posed and treated as boundary value problems.
- The *time* coordinate is almost always posed as an initial value problem and tends to introduces complexity in the form of numerical stability issues.

## Classification of PDEs

Introduction of time as a coordinate in PDEs raises the question of *how information propogates*, from the initial condition or other events that occur during solution.

## Elliptic equations

Equations of the form, $
\( A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} \)$ are termed *elliptic*. The lack of a rate of change term implies that information is *not being propogated*, but rather is either instantaneously available everywere in space, or all propagations have finished and you are left with a steady state.

Note this does not imply the system isn't changing with time (imagine the coefficients could be functions of time, \(A(t)\)), just that the impact of the change has reach an equilibrium.

Examples include steady-state, or quasistatic equations:

Laplace's and and Poisson's equation: $\nabla^2 u = 0$, $\nabla^2 u = f(x)$

- Electrostatics: Electric potential with a charge
- Heat transport: Steady state heat distribution with a source
- Fluid flow: Velocity potential of an incompressible, irrotational fluid with a sink

Helmholtz equation, $\nabla^2 u +k^2 u = 0$

- Accoustics: vibrations of a membrane of a cavity
- Quantum mechanics: wave functions in a potential well

> Inclusion of Helmholtz's equation may be surprising since it describes wave phenomena! Recall that wave propogation can be solved by the method of separation of variables to separate the time and space dependance, the latter of which is of the Helmholtz's form (and defines the eigenfunctions!)

### Solution approaches

Elliptic equations generally amount to solving a system of (nonlinear) equations simultaneously, which we can do with root finders. Let's recall the shortfalls of rootfinders in this context:

- A good initial guess: This can be a substantial problem in common systems. A mitigation strategy might be to parametrically ramp-up a parameter from an easily solvable condition to your desired condition. E.g.: gradually 'turn up the heat' on a natural convection problem.
- Convergence: You may need to discretize the problem with a large mesh which is a high dimensional problem. The root finder may *wander* easily in this space and fall into local minima or step completely out of the feasible region. Trust-region methods can help mitigate this but consume additional resources.

There is another subtle issue with parallelization on modern high performance platforms. A common approach is to perform *domain decomposition* which partitions the domain geometrically and passes pieces to different nodes. Elliptic systems imply each node instantaneously *talks* to all the others, which can be an overhead / node synchronization nightmare! For this reason *multigrid methods / preconditioners* are tremendously effective.

## Parabolic equations

Equations of the form, $$\( A \frac{\partial u}{\partial t} - C \frac{\partial^2 u}{\partial x^2} \)$$

propogate information in one direction (forward in time) and the process is *diffusive*.

Diffusive processes tend to *smooth* out discontinuities and any small oscillations, which is a favourable feature but doesn't gaurantee stability as we will see.

This still implies an infinite rate of propogation of information, but now in the flux which instantaneously *knows* about a force everywhere at once.

##Solution schemes

Solving diffusion problems generally amounts to a time-marching scheme in which time is discretized with the spactial equation solved at each time step. We still have to be wary of stiffness!

It is common to want to solve the *inverse* problem, in which one uses a result to attempt to back-calculate a parameter of the model (e.g.: Use the rate of cooling to infer the thermal conductivity). It is here that the tendancy of diffusion to smooth the features (information only flows forward in time) works against you. Many paths can lead to the same result (consider trying to sharpen a blurry image!) this can be very problematic to arrive at anything more than a statistical answer.

## Hyperbolic equations

Equations of the form, $\( A \frac{\partial^2 u}{\partial t^2} - C \frac{\partial^2 u}{\partial x^2} \)
\( propogate information forward *and* backward in time. This implies time-reversal symmetry, wherein you could replace \)t
\( with \)-t
\( and the physics would remain unchanged. This describes waves (second order equations) and advective terms (first order equations) e.g.: \)
A \frac{\partial u}{\partial t} - C \frac{\partial u}{\partial x}$

### Solution schemes

Second time derivatives can generally be decomposed into a series of parabolic equations, which is a standard approach. Systems with periodic boundary conditions are treatable with fourier spectral analysis which can be much more efficient.

Time reversal symmetry implies the model may be *reversible* which is interesting for the inverse problem and reconstruction of earlier times before what is measured / simulated.

#Solving time dependent PDEs

The general approach to solving time dependent PDEs is to transform them into a BVP through a time-marching scheme:

1. Discretize the time derivative as an IVP into current and previous solution(s) with the initial value known
2. Discretize space as a BVP and add the term from the time descritization.
3. Step forward in time, updating the BVP accordingly.

Consider the parabolic case, since we can usually use reduction of order to transform a hyperbolic equation into parabolic equations.

$$ \frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} $$

where \(\alpha\) is the thermal diffusivity, which is necessarily postive.

The finite difference scheme for the spatial component is: $
\(\alpha \frac{\partial^2 T}{\partial x^2} = \alpha \frac{T_{i-1} - 2 T_i + T_{i+1}}{\Delta x^2}\)$

### Notation

- The spatial index will be noted as a subscript and indexed with \(i\), e.g.: \(T_i\).
- The temporal index will be index with superscript \(t\).
- The known timestep will always be \(T^t\) and we will be calculating \(T^{t+1}\).

# Explicit Euler timestepping

As in the IVP section, the explicit Euler scheme calculates the next solution based on the current one, $
\( \frac{\partial T}{\partial t} = \frac{T^{t+1} -T^{t}}{\Delta t}\)$

and the full equation becomes,

\[\begin{split} \begin{align} \frac{\partial T}{\partial t} &= \alpha \frac{\partial^2 T}{{\partial x}^2} \\ \frac{T^{t+1}_i -T^{t}_i}{\Delta t} &= \alpha \frac{T^t_{i-1} - 2 T^t_i + T^t_{i+1}}{\Delta x^2} \\ T^{t+1}_i &= T_i^t + r [T^t_{i-1} - 2 T^t_i + T^t_{i+1}] \end{align}\end{split}\]
with $\( r = \frac{\alpha \Delta t} {\Delta x^2}\)$ for compactness.

As a stencil for forward Euler time stepping is:


This implies that the value at a point at the next timestep depends *only* on the values at the previous time-step.

## Example of Forward Euler time stepping

```
# prompt: Give me an example of forward euler timestepping for a 1D heat balance equation with a slider

import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, FloatSlider

def forward_euler_heat(T, alpha, dx, dt, num_steps):
  """
  Solves the 1D heat equation using forward Euler timestepping.

  Args:
    T: Initial temperature distribution (numpy array).
    alpha: Thermal diffusivity.
    dx: Spatial step size.
    dt: Time step size.
    num_steps: Number of time steps.

  Returns:
    A list of temperature distributions at each time step.
  """

  T_history = [T.copy()]
  for _ in range(num_steps):
    T_new = T.copy()
    for i in range(1, len(T) - 1):
      T_new[i] = T[i] + alpha * dt / (dx ** 2) * (T[i - 1] - 2 * T[i] + T[i + 1])
    T = T_new
    T_history.append(T.copy())
  return T_history


def plot_heat_equation(dt):
  """
  Plots the solution of the heat equation for a given time step size.

  Args:
    dt: Time step size.
  """

  # Parameters
  alpha = 0.1  # Thermal diffusivity
  dx = 0.1  # Spatial step size
  num_steps = 50
  x = np.arange(0, 1, 1/num_steps)
  T_initial = np.zeros_like(x)
  T_initial[int(len(x) / 2)] = 1  # Initial heat source in the middle

  # Solve the heat equation
  T_history = forward_euler_heat(T_initial, alpha, dx, dt, num_steps)

  # Plotting
  plt.figure(figsize=(8, 6))
  for i in range(0, len(T_history), 5):
    plt.plot(x, T_history[i], label=f"Time step {i}")
  plt.xlabel("Position (x)")
  plt.ylabel("Temperature (T)")
  plt.title("Forward Euler Solution of 1D Heat Equation")
  plt.legend()
  plt.grid(True)
  plt.show()

# Interactive widget
interact(plot_heat_equation, dt=FloatSlider(min=0.001, max=0.05, step=0.001, value=0.01));
```

## Error analysis

Analysing the error using the Explicit Euler timestepper is more sophisticated than for the ODE since we now need to consider the behaviouir of a (spatial) function at a given time.

We can do this through *von Neumann stability analysis* which checks the stability of a solution to oscillations. Mathematically, we express the soultion as a Fourier series and see which waves damp and which waves grow.

Consider Fourier modes of the solution: $^kT_i^t = e^{j k i \Delta x}$ where $k$ is the wave number and $j$ is the imaginary number. For a particular wave with number $k$, we check how the future solution, $^kT_i^{t+1}$ depends on the current, $^kT_i^n$, and define the growth factor $G$, $^kT_i^{t+1} = G ^kT_i^{t}.$ A method is deemed stable if the amplitude of the oscillations don't grow over time; i.e.: $|G| \le 1$.

Substitute this into the update equation:

$$\begin{split} \begin{align} ^k T^{t+1}_i &= e^{j k i \Delta x} + \big[e^{j k [i-1] \Delta x} - 2 e^{j k i \Delta x} + e^{j k [i+1] \Delta x}\big] r \\ &= e^{j k i \Delta x} \big[1+ r [e^{j k \Delta x} +e^{-j k \Delta x} -2] \big] \\ ^k T^{t+1}_i &= \ ^kT^{t}_i \big[1+ r [e^{j k \Delta x} +e^{-j k \Delta x} -2] \big] \\ G &= 1+2 r [\cos(\Delta x)-1] \end{align}\end{split}$$

Looking at the cosign, we can see $G$ is in the range $\bigg[1-4\frac{\alpha \Delta t}{\Delta x^2}, 1\bigg]$. The maximum already satisfies the stability, but the minimum must satisfy, $\( \frac{\alpha \Delta t}{\Delta x^2} \le \frac{1}{2}\)$

or more simply $\( \Delta t \le \frac{\Delta x^2}{2 \alpha}\)$

For higher dimensions, additional waves add to the instabilities. For a square grid with step size $h$,

2D: $\( \Delta t \le \frac{h}{4 \alpha}\)$ 3D: $\( \Delta t \le \frac{h}{6 \alpha}\)$

Once again we have come up against a stiffness issue wherein our timestep is limited by numerical stability rather than accuracy. Unsurprisingly and unfortunatley, this is true of all our explicit time stepping schemes.

# Implicit Euler timestepping

As with the ODE case, we can attempt an Implicit Euler timestepping scheme: $
\( \frac{\partial T}{\partial t} = \frac{T^{t+1} -T^{t}}{\Delta t}\)$

and now discretize space at the *new timestep*:

$$\[ T^{t+1}_i = T_i^{t} + [T^{t+1}_{i-1} - 2 T^{t+1}_i + T^{t+1}_{i+1}] r\]$$

The stencil is:




A quick glance at the error analysis will see that it is now $\frac{1}{G}$ which is in the range $\bigg[1-4\frac{\alpha \Delta t}{\Delta x^2}, 1\bigg]$ and which always satisfies the stability condition.

Once again the Forward Euler method, and implicit method more generally, are unconditional stability but at the cost of solving a system of equations.

### Example of Backward Euler time stepping

```
# prompt: Repeat solving hte heat transport equation but thistime with a backward Euler method

import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, FloatSlider

def backward_euler_heat(T, alpha, dx, dt, num_steps):
  """
  Solves the 1D heat equation using backward Euler timestepping.

  Args:
    T: Initial temperature distribution (numpy array).
    alpha: Thermal diffusivity.
    dx: Spatial step size.
    dt: Time step size.
    num_steps: Number of time steps.

  Returns:
    A list of temperature distributions at each time step.
  """

  T_history = [T.copy()]

  for _ in range(num_steps):
    A = np.diag(-alpha * dt / dx**2 * np.ones(len(T) - 1), -1) + \
        np.diag(1 + 2 * alpha * dt / dx**2 * np.ones(len(T)), 0) + \
        np.diag(-alpha * dt / dx**2 * np.ones(len(T) - 1), 1)

    # Apply boundary conditions (Dirichlet in this example, T=0 at edges)
    b = T.copy()
    b[1:-1] = T[1:-1]

    # plt.spy(A, markersize=1)  # Creates a sparsity plot
    # plt.title('Sparsity Pattern of Matrix A')
    # plt.show()

    # Solve the linear system for T at the next time step
    T_new = np.linalg.solve(A, b)

    # Update T and store the solution
    T = T_new.copy()
    T_history.append(T.copy())

  return T_history

def plot_heat_equation(dt):
  """
  Plots the solution of the heat equation for a given time step size.

  Args:
    dt: Time step size.
  """

  # Parameters
  alpha = 0.1  # Thermal diffusivity
  dx = 0.1  # Spatial step size
  num_steps = 50
  x = np.arange(0, 1, 1/num_steps)
  T_initial = np.zeros_like(x)
  T_initial[int(len(x) / 2)] = 1  # Initial heat source in the middle

  # Solve the heat equation
  T_history = backward_euler_heat(T_initial, alpha, dx, dt, num_steps)

  # Plotting
  plt.figure(figsize=(8, 6))
  for i in range(0, len(T_history), 5):
    plt.plot(x, T_history[i], label=f"Time step {i}")
  plt.xlabel("Position (x)")
  plt.ylabel("Temperature (T)")
  plt.title("Backward Euler Solution of 1D Heat Equation")
  plt.legend()
  plt.grid(True)
  plt.show()


# Interactive widget
interact(plot_heat_equation, dt=FloatSlider(min=0.01, max=0.1, step=0.01, value=0.05));
```

# Crank-Nicholson method

One shortcoming of the backward Euler method is that it is only 1st order accurate in time whereas the central difference spatial discretization is second order. This means that one would need smaller time steps compared to spatial steps to ensure accuracy is optimized.

The Crank-Nicholson method is an alternative implicit method that is second order in both time and space. As with the central difference derivation (and many others!) the key is to average the explicit and implicit methods!

Take, $\( \frac{\partial T}{\partial t} = \frac{T^{t+1} -T^{t}}{\Delta t}\)$

and now average the spatial distretizations at \(t+1\) and \(t\):

\[ \frac{\partial^2 T}{\partial x^2} = \frac{1}{2} [T^{t+1}_{i-1} - 2 T^{t+1}_i + T^{t+1}_{i+1} + T^{t}_{i-1} - 2 T^{t}_i + T^{t}_{i+1}]
{\Delta x^2}\]
Such that the increment scheme is: $
\( (1 + 2r) T_i^{t+1} - r T_{i+1}^{t+1} - r T_{i-1}^{t+1} = (1 - 2r) T_i^{t} + r T_{i+1}^{t} + r T_{i-1}^{t}\)$

which seems ugly at first but is actually the exact same matrix sparsity as backward Euler, so achieves better accuracy for comparable computational cost.

The stencil is,

![Crank Nicholson stencil.svg]
(

## Example of Crank Nicholson accuracy vs. Euler

```
# prompt: compare the backward euler and crank-nicholson method for 5 time steps using the  previous exa

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from ipywidgets import interact, IntSlider

# ... (Your existing code for forward_euler_heat, backward_euler_heat, etc.) ...


def crank_nicholson_heat(T, alpha, dx, dt, num_steps):
  """
  Solves the 1D heat equation using the Crank-Nicholson method.

  Args:
    T: Initial temperature distribution (numpy array).
    alpha: Thermal diffusivity.
    dx: Spatial step size.
    dt: Time step size.
    num_steps: Number of time steps.

  Returns:
    A list of temperature distributions at each time step.
  """

  T_history = [T.copy()]
  r = alpha * dt / (2 * dx**2)

  for _ in range(num_steps):
    A = np.diag(1 + 2 * r * np.ones(len(T)), 0) + \
        np.diag(-r * np.ones(len(T) - 1), -1) + \
        np.diag(-r * np.ones(len(T) - 1), 1)

    b = np.diag(1 - 2 * r * np.ones(len(T)), 0) @ T + \
        np.diag(r * np.ones(len(T) - 1), -1) @ T + \
        np.diag(r * np.ones(len(T) - 1), 1) @ T

    plt.spy(A, markersize=1)  # Creates a sparsity plot
    plt.title('Sparsity Pattern of Matrix A')
    plt.show()

    # Apply boundary conditions (Dirichlet in this example, T=0 at edges)
    b[0] = 0
    b[-1] = 0

    T_new = np.linalg.solve(A, b)
    T = T_new.copy()
    T_history.append(T.copy())

  return T_history


def plot_comparison(time_index):
  """
  Plots the solution of the heat equation using different methods for 5 timesteps.

  Args:
    time_index: Index of the time step to plot.
  """

  # Parameters
  alpha = 0.1
  dx = 0.1
  dt = 0.01
  num_steps = 5
  x = np.arange(0, 1, 1/10)
  T_initial = np.zeros_like(x)
  T_initial[int(len(x) / 2)] = 1

  # Solve the heat equation using different methods
  forward_euler_solution = forward_euler_heat(T_initial, alpha, dx, dt, num_steps)
  backward_euler_solution = backward_euler_heat(T_initial, alpha, dx, dt, num_steps)
  crank_nicholson_solution = crank_nicholson_heat(T_initial, alpha, dx, dt, num_steps)

  # Plotting
  plt.figure(figsize=(8, 6))
  plt.plot(x, forward_euler_solution[time_index], label="Forward Euler")
  plt.plot(x, backward_euler_solution[time_index], label="Backward Euler")
  plt.plot(x, crank_nicholson_solution[time_index], label="Crank-Nicholson")
  plt.xlabel("Position (x)")
  plt.ylabel("Temperature (T)")
```

```
    plt.ylim(-.1, 1.1)
    plt.title(f"Comparison of Methods at Time Step {time_index}")
    plt.legend()
    plt.grid(True)
    plt.show()


# Interactive widget
interact(plot_comparison, time_index=IntSlider(min=0, max=4, step=1, value=0));
```

#Summary

Once again, we have seen the value of implicit timesteppers for solving stiff problems. However, as discussed in the Initial Value Problem section, they require a root finder to converge on a system, which generally relies on linear solvers. For large systems, direct solution (LU) is intractable, and so iterative methods are required.

Explicit timesteppers are conditionally stable which limits the timestep, but they don't require simultaneous solutions of equations which avoids the pitfalls of a root finder:

- no question of initial guesses
- always converges
- parallelizes well on HPCs
- much smaller memory usage
- Does not need an iterative linear solver routine

At what point does the expense of interations in an implicit time stepper outweigh the moderate step size? -> It depends on the problem!

General multiphysics solvers rely on implicit timestepping since modern systems can use LU to root find. Specialized software (build for large systems) may prefer explicit methods which *just work*:

- LS-Dyna - impact tests, crashes, drop tests, etc.
- Several CFD packages (Flow 3D)

# The Finite Element Method

The Finite Element Method (FEM) is a very popular method with a wide range of applications. It is an *integral* technique, similar to the Finite Volume Method which provide it with a number of benefits.

- Complex geometries
- Irregular meshing
- Irregular material properties
- Options to tweak the method according to the physics
- (quasi)dynamic mesh refinement
- Highly flexible boundary conditions
- Ability to solve *boundary physics*
- Excellent accuracy

## Motivating example

Consider an ODE \( y^\prime = -y \quad y(0) = 1\), which has an analytical answer \(y = e^{-t}\).

We wish to approximate this function numerically between \([0,1]\) with a line defined by the boundary points, \(f(0) = a = 1\) and \(f(1) = b\), $\(f(x) = [1-x] + x b\)$

How do we find \(b\)?

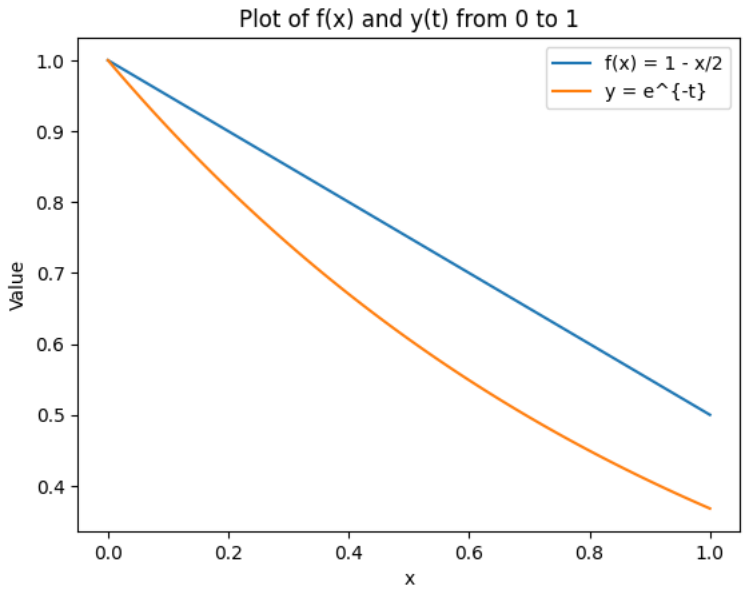
## Option 1: Direct substitution

Following the Finite Difference method, we can subsitute $f(1)$ into the ODE and solve for b:

$$\begin{split}\begin{align} f'(1) &= f(1) \\ b-1 &= -b \\ b &= 1/2 \end{align}\end{split}$$

and the approximate solution is $f = 1 -\frac{x}{2}$

```
import numpy as np
import matplotlib.pyplot as plt

# Define the functions
f = lambda x: 1 - x / 2
y = lambda t: np.exp(-t)

# Generate values
x = np.linspace(0, 1, 100)
f_values = f(x)
y_values = y(x)

# Plot the functions
plt.plot(x, f_values, label='f(x) = 1 - x/2')
plt.plot(x, y_values, label='y = e^{-t}')
plt.xlabel('x')
plt.ylabel('Value')
plt.legend()
plt.title('Plot of f(x) and y(t) from 0 to 1')
plt.show()
```



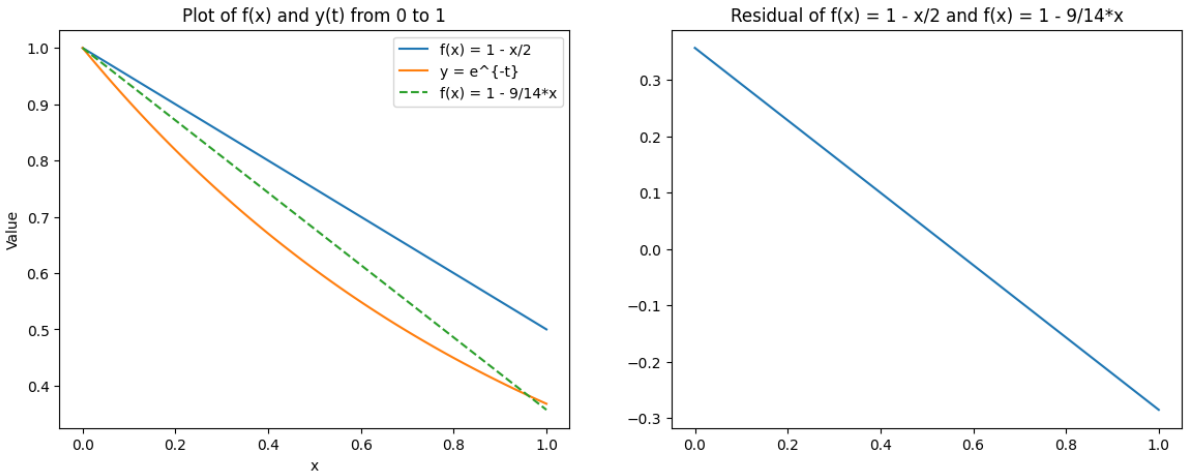

## Option 2: Least square minimization of the residual

Let's integrate the squared residual of f(x) plugged into the ODE, $y^\prime + y$

$$\begin{split}\begin{align} R &= f^\prime(x) +f(x) \\ &= [b-1] + 1-x + x b \\ &= b+x[b-1] \\ R^2 &= b^2 + 2 b x[b - 1] + [x[b - 1]]^2 \\ F = \int_0^1 R^2 \ dx &= \frac{1}{3} [7 b^2 - 5 b + 1] \\ \end{align} \end{split}$$

Note that in the end the integral of the residual, $F$ is a parabolic function of the parameter $b$. The minimum is found at: $\frac{\partial F}{\partial b}(b) = 0$, for $b = \frac{5}{14}$. The approximate solution is $f(x) = 1-\frac{9}{14}x$.

```
# Define the new function
fig, axs = plt.subplots(1, 2, figsize=(14, 5))
f_new = lambda x: 1 - 9./14*x
f_new_values = f_new(x)


# Plot the functions
axs[0].plot(x, f_values, label='f(x) = 1 - x/2')
axs[0].plot(x, y_values, label='y = e^{-t}')
axs[0].plot(x, f_new_values, label='f(x) = 1 - 9/14*x', linestyle='--')
axs[0].set_xlabel('x')
axs[0].set_ylabel('Value')
axs[0].legend()
axs[0].set_title('Plot of f(x) and y(t) from 0 to 1')

# Plot the residual
residual = lambda x: -9/14 + 1 - 9./14*x
axs[1].plot(x, residual(x))
axs[1].set_title('Residual of f(x) = 1 - x/2 and f(x) = 1 - 9/14*x')

plt.show()
```




We see that option 2 does a better job of approximating the function over the full domain since it focusses on the integral of the residual.

You may have noticed that we did something funny here: we integrated $R^2$ and then took its derivative with respect to $b$ in order to find the minimum of the integral! I.e.:

$$\begin{split}\begin{align} \frac{\partial F}{\partial b} = 0 &= \frac{\partial }{\partial b} \int_0^1 R^2 \ dx \\ &= \int_0^1 2R \frac{\partial R}{\partial b} \ dx \\ &= \int_0^1 R v \ dx \end{align} \end{split}$$

where the residual has now been weighted by the function $v$. This is called the Method of Weighted Residual [MWR]. Let's try a different function and see what happens.

## Option 3: Integrate the MWR to find the parameters

Integrate the weighted residual function directly with a convenient choice of $v$.

lets take $v = x$ (to be motivated later) and integrate:

$$\begin{split}\begin{align} \frac{\partial F}{\partial b} =0 &= \int_0^1 R x \ dx \\ &= \int_0^1 \big[b+[b-1]x\big]x \ dx \\ &= \int_0^1 b x+[b-1]x^2 \ dx \\ &= \bigg[ \frac{1}{6} x^2\big[3b+2[b-1]x\big]\bigg]_0^1 \\ &= b-\frac{2}{5} \\ \end{align} \end{split}$$

which is very close to our previous solution $b = \frac{5}{14}$ but with a simpler integration. BUT, we are still let with analytically integrating! If only there were a way to express integrals as the sum of the integrand evaluated at certain points…

## Option 4: Express the MWR integral as the sum of the integrand evaluated at

## certain points.

Integrate the weighted residual function directly with a convenient choice of $v$ as the sum of the integrand evaluated at certain points:

Recall Gaussian Quadrature allows us to evaluate an integral by summing the integrand at the Gauss Points. The Gauss Points for the domain $[-1,1]$ are $\pm\frac{1}{\sqrt{3}}$, scaled to this integral domain become,

$$\begin{split} \begin{align} x_{[0,1]} &= 0.5 (x_{[-1,1]} + 1) \\ &= \frac{1}{2} \bigg[1 \pm\frac{1}{\sqrt{3}} \bigg] \\ &\approx [0.21132487, 0.78867513] \end{align} \end{split}$$

and

$$\begin{split}\begin{align} 0 &= \int_0^1 R x \ dx \\ &= Rx (x=.21132487) + Rx(x = .78867513) \\ &= -0.0446582 + 0.255983 b -0.622008 + 1.41068 b \\ b &= \frac{2}{5} \end{align} \end{split}$$

## Recap of the approach

Let's recap what we've done:

1. parameterized a function as a weighted sum of simpler functions (a linear basis)
2. found a (simple) integral expression that minimizes the error in the approxiation (Minimized Weighted Residual)
3. performed the integration *exactly* using only function evaluations into a linear system (Gaussian quadrature)
4. Solved the linear system (I **told** you everything boiled down to linear systems!)

Practical solution of linear systems requires sparsity! Let's formalize our proceedure and see how we can ensure sparsity.

# Theoretical background

## Derivation of the finite element method BVPs

Consider a differential equation, $\( \mathcal{L}(u) = f \quad \text{in } \Omega, \)$

in one dimension for convenience.

** Boundary conditions in $\partial \Omega$?

where:

- $\mathcal{L}$ is a differential operator
- $u(x)$ is the unknown *exact* solution
- $f(x)$ is a given source term dependant only on space.
- $\Omega$ is the ND domain of the problem, with boundary $\partial \Omega$.

Define the residual, $\( R(u) = \mathcal{L}(u) - f\)\( For the exact solution, \)R(u) = 0\( everywhere in \)\Omega$.

### Approximate the solution with *shape functions*

In liue of an exact solution, we will have to settle for an approximate solution, $u_h$, for which $R(u_h) \ne 0$ but will be made as small as possible. As discussed in the Interpolation sections, a function can be approximated as a weighted sum of basis functions:

$$\begin{split} \begin{align} u_h(x) &= \sum_{i=1}^N w_i \phi_i(x) \\ \end{align}\end{split}$$

Where:

- $w_i$ are the weights / coefficients of the approximation

- $\phi_i(x)$ are a set of suitable *shape functions*.

It is trivial to see that, $\( \nabla u_h = \sum_i w_i \nabla \phi_i \)$

### Minimizing the residual with the *Method of Weighted Residuals*

The best approximation will minimize the residual everywhere in $\Omega$. In previous work (e.g.: curve fit) we minimzed the square of the residual, but here we will use the Method of Weighted Residuals:

$$ \int_\Omega R v \ dx = 0 \]$$

The MWR is a *weaker* statement of minimization. $v(x)$ is a *test* function for which we have options.

#### $v = 2\frac{\partial R}{\partial w_i}$ : Least Squares

If we take $v = 2\frac{\partial R}{\partial w_i}$,

$$\begin{split} \begin{align} \int_\Omega R \frac{\partial R}{\partial w_i} d\Omega &= 0 \\ \frac{\partial }{\partial w_i} \int_\Omega R^2 d\Omega &= 0 \end{align}\end{split}\]$$

which is exactly the condition for finding the parameters for Least Squares minimization.

#### $v = \delta (x) $: Collocation methods

If $v = \delta (x)$ then for a discrete set of points $x_i$, the integral turns into a sum:

$$\begin{split} \begin{align} \int_\Omega R \delta(x_i) d\Omega &= 0 \\ \sum_i R(x_i) = 0 \end{align}\end{split}\]$$

which requires that the residual be zero at all discretization points (the collocation method).

#### $v = \sum_i \phi_i$): The Galerkin method

The Galerkin method chooses the test functions to be in the same basis as the solution $u_h = \sum_i w_i \phi_i$.

$$\begin{split} \begin{align} \int_\Omega R v \ d\Omega &= 0 \\ \sum_i \int_\Omega R \phi_i \ d\Omega &= 0 \end{align}\end{split}\]$$

Why does this work? Note that the MWR is an expression of orthogonality between $R$ and $v$. What this means is that the residual has no component in the space defined by the basis functions, or more simply, it can't be improved by changing the weights $w_i$. This is a more general expression for the minimum which is more in line with our discrete / numerical approach. It also opens avenues to choose $\phi_i$ carefully for computational efficiency!

## Meshing the domain

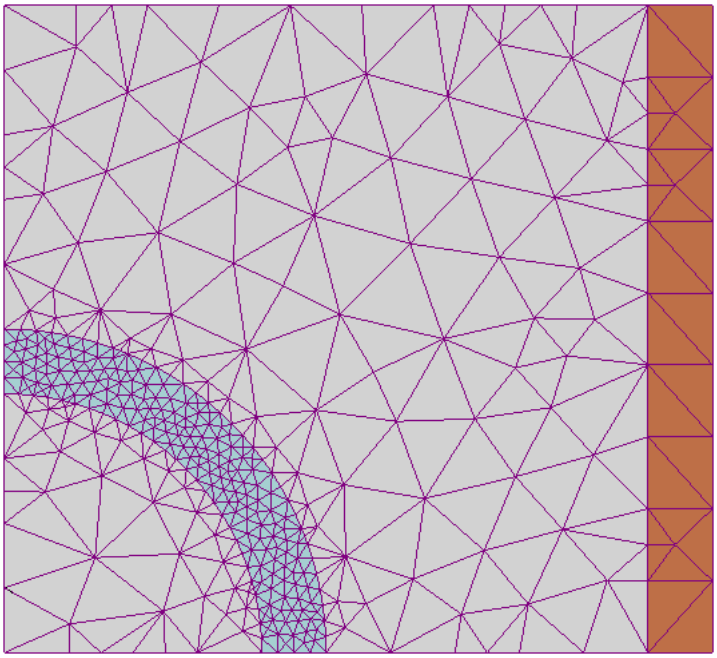
The geometry of the problem can generally be complex, with curved boundaries separating adjacent domains. Additionally, we may want different spatial resolution at different points in our model to capture gradients or differeing material properties.

The Finite Element Method partitions the domain into a *Finite* set of *Elements* through tesselation (space filling tiling). This meshing proceedure is actually very complex and there are sophisticated, dedicated software packages e.g., Cubit (commercial), GMSH (open source), etc.

The most flexible element in 2D is triangular, but quadralateral elements are also popular. In 3D, tetrahedra are common, triangular prisms or rectangular prisms are also admissable.

Elements may be mixed to better capture features. E.g.: in Computational Fluid Dynamics one may use square boundary layers along a surface and then triangles in the stream.

One does have to be wary of degeneracy where one side is small compared to the others, which can lead to poor numerical performance.



Notice how each mesh cell is a scaled, translated version of the same reference cell.

# Elements

Shape functions are generally chosen to approximate the (unknown) function *on each element* in terms of the weights discussed above, which are generally called the *degrees of freedom* (DOFs).

The choice of *element type* determines:

- the function which approximates the solution on the element.
- the *nodes* - the points within the element at which the degrees of freedom are specified.
- the meaning of the DOFs which might be:
    - function value
    - gradient
    - divergence of a vector field
- the continuity of the approximate solution between elements.
- integration scheme (typically in terms of the Gauss Points).

One might consider placing DOFs at the Gauss Points to expedite integration, but this is not common. It is more important for the the shape functions should reflect the behaviour of the solution (e.g.: continuity, conservation of divergence, high order variations.

## Lagrange elements

A general purpose basis is our good old Lagrange interpolation functions! These functions have nodes at the verticies of the mesh (at least), at which the value of the function is specified. Higher order elements (i.e. higher order polynomials) include additional nodes *inside* the element.

LaGrange shape functions are 1 at the associated node and 0 at *every* other node.

$$\phi_i(x_j) = \delta_{ij}$$

The first three orders (in 1D) are:

| Order | Node …….. | Basis Function…….. |
|---|---|---|
| 0 | $x_0 = 0$ | $\phi_0(x) = 1$ |
| 1 | $x_0 = 0$ | $\phi_0(x) = 1 - x$ |
| | $x_1 = 1$ | $\phi_1(x) = x$ |
| 2 | $x_0 = 0$ | $\phi_0(x) = 2(1 - x)(0.5 - x)$ |
| | $x_1 = 0.5$ | $\phi_1(x) = 4x(1 - x)$ |
| | $x_2 = 1$ | $\phi_2(x) = 2x(x - 0.5)$ |
| 3 | $x_0 = 0$ | $\phi_0(x) = -9/2 (x - 1/3)(x - 2/3)(x - 1)$ |
| | $x_1 = 1/3$ | $\phi_1(x) = 27/2 x(x - 2/3)(x - 1)$ |
| | $x_2 = 2/3$ | $\phi_2(x) = -27/2 x(x - 1/3)(x - 1)$ |
| | $x_3 = 1$ | $\phi_3(x) = 9/2 x(x - 1/3)(x - 2/3)$ |

as plotted below.

```python
import numpy as np
import ipywidgets as widgets
from ipywidgets import interact

import matplotlib.pyplot as plt

# Define Lagrange basis functions for each order
def lagrange_basis(order, x, node):
    """
    Compute the Lagrange basis function for a given node and order.

    Args:
    - order: Order of the Lagrange polynomial (0, 1, 2, 3).
    - x: The points where the function is evaluated (numpy array).
    - node: The index of the current basis function (0, 1, ..., order).

    Returns:
    - The values of the Lagrange basis function at points `x`.
    """
    if order ==0:
      nodes = .5
    else:
      nodes = np.linspace(0, 1, order + 1)
    basis = np.ones_like(x)
    for j in range(order + 1):
        if j != node:
            basis *= (x - nodes[j]) / (nodes[node] - nodes[j])
    return basis

# Function to plot the Lagrange basis functions
def plot_lagrange_basis(order):
    x = np.linspace(0, 1, 500)  # Points to evaluate the basis functions
    colors = ['blue', 'green', 'orange', 'red']  # Colors for each order
    plt.figure(figsize=(10, 8))
    if order ==0:
      nodes = np.array([.5])
    else:
      nodes = np.linspace(0, 1, order + 1)
    for node in range(order + 1):
        y = lagrange_basis(order, x, node)
        plt.plot(x, y, label=f"Node {node}")
        plt.scatter(nodes[node], 1, color=colors[order % len(colors)], zorder=5)
    plt.title(f"Lagrange Basis Functions of Order {order}")
    plt.xlabel("x")
    plt.ylim([-.4, 1.2])
    plt.ylabel("Basis Function Value")
    plt.legend(loc='best', fontsize=8)
    plt.grid(True)
    plt.show()

# Create an interactive slider for the order
interact(plot_lagrange_basis, order=widgets.IntSlider(min=0, max=3, step=1, value=0))
```

```
plot_lagrange_basis
def plot_lagrange_basis(order)

<no docstring>
```

## Continuity

Since the LaGrange Shape functions are defined *at least* on the mesh verticies, those DOFs may be shared between adjacent elements. This enforces continuity in the solution at the nodes (at least the verticies) and, if both adjacent elements are off the same type/order, along \(\partial \Omega\) too.

The flux / first derivative however is not generally continuous. However, the weak form does not require this continuity to approximate a second order (in space) PDE!

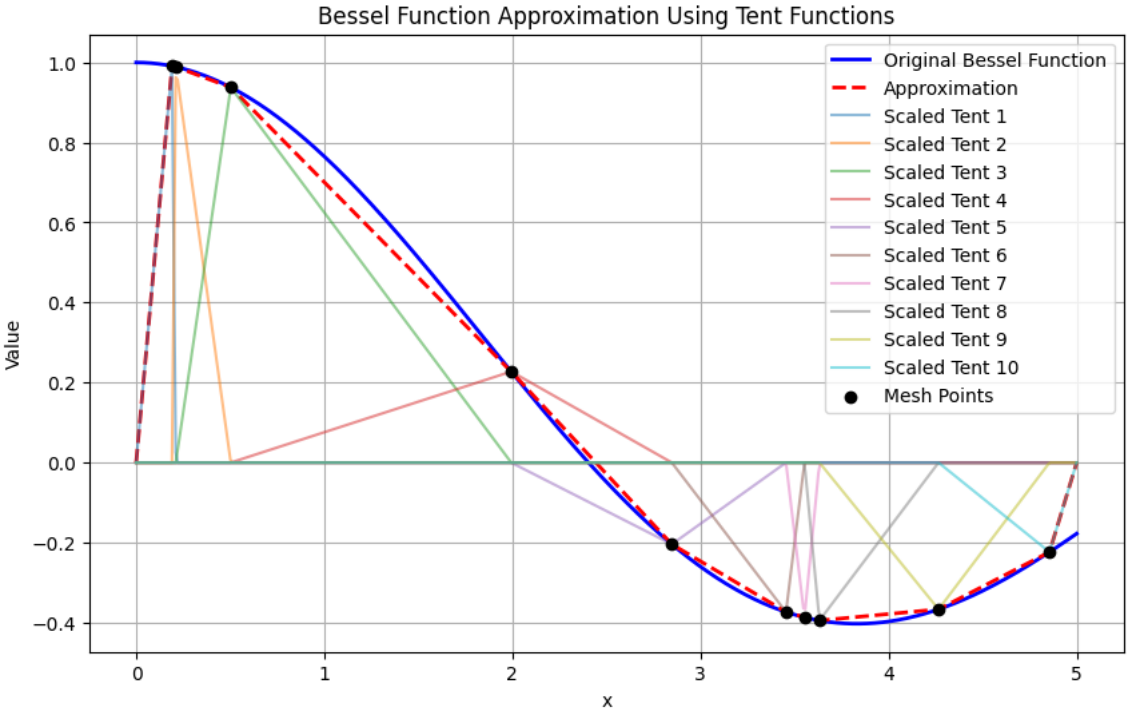
Take another look at the Lagrange element order 0 above. It doesn't have DOFs at the end points and so this is actually a *discontinuous lagrange element*.

### Example: Tent functions

An illustrative example is the 1D Linear LaGrange elements which are lines inside the element (interval). Continuity between elements implies sharing the vertex values and therefore we see *tent / triangle* functions.

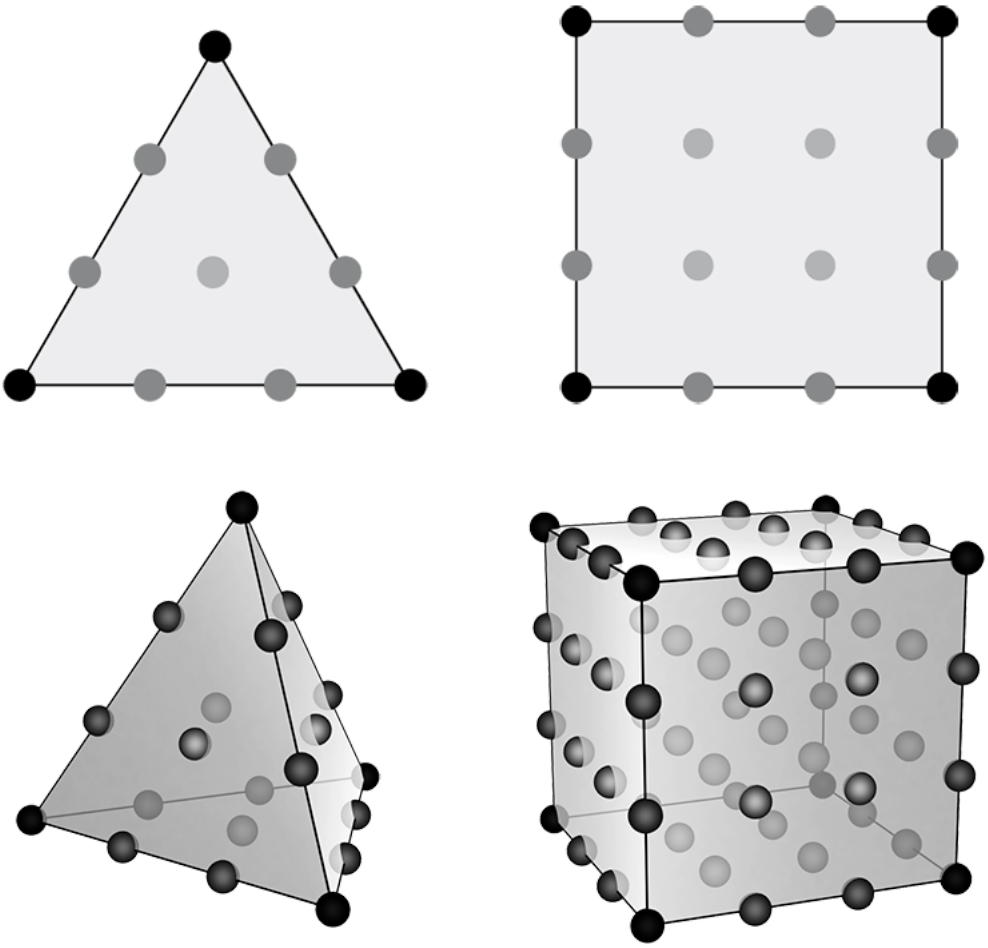
Note that since the shape functions are \(1\) only at the mesh verticies, calculating the weights is as easy as evaluating the function at those points!

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import jv

# Define the Bessel function to approximate
def bessel_function(x):
    return jv(0, x)

# Define the mesh (unevenly spaced points)
mesh = np.sort(np.random.uniform(0, 5, 10))

# Define the tent (piecewise linear) basis functions
def tent_function(x, xi, x_prev, x_next):
    if x_prev <= x <= xi:
        return (x - x_prev) / (xi - x_prev)
    elif xi < x <= x_next:
        return (x_next - x) / (x_next - xi)
    else:
        return 0.0

# Evaluate the coefficients for the tent functions (sample the Bessel function at the mesh points)
coefficients = bessel_function(mesh)

# Define the range for plotting
x_vals = np.linspace(0, 5, 500)

# Compute the approximation
approximation = np.zeros_like(x_vals)
tent_functions = []

for i, xi in enumerate(mesh):
    x_prev = mesh[i - 1] if i > 0 else 0
    x_next = mesh[i + 1] if i < len(mesh) - 1 else 5
    tent_func = np.array([tent_function(x, xi, x_prev, x_next) for x in x_vals])
    tent_functions.append(tent_func)
    approximation += coefficients[i] * tent_func

# Plot the original function, the approximation, and the tent functions
plt.figure(figsize=(10, 6))
plt.plot(x_vals, bessel_function(x_vals), label='Original Bessel Function', color='blue', linewidth=2)
plt.plot(x_vals, approximation, label='Approximation', color='red', linestyle='--', linewidth=2)
for i, tent_func in enumerate(tent_functions):
    plt.plot(x_vals, coefficients[i] * tent_func, label=f'Scaled Tent {i+1}', alpha=0.5)
plt.scatter(mesh, coefficients, color='black', label='Mesh Points', zorder=5)
plt.title('Bessel Function Approximation Using Tent Functions')
plt.xlabel('x')
plt.ylabel('Value')
plt.legend()
plt.grid()
plt.show()
```




## 2D and 3D, and other shapes

Extension to 2D and 3D is fairly straight-forward, beginning with the verticies and then refining by subdivision.



## Advanced elements

The field of Finite Elements is very rich, with elements designed specifically for certain types of physics e.g.:

- Curl of a vector field
- Divergence of a vector field
- Discontinuous elements
- Spectral Elements (very high order based on orthogonal shape functions).

A first step may be the Periodic Table of Finite Elements

# Discretization and assembly

## The weak form

Our usual formulation of BVPs is called the *strong* form because it assumes a lot of continuity in a solution. E.g.: if the equation contains $\nabla^2 u$, the solution assumes that that derivative exists!

The MWR leads to the *weak form* which has less requirements on the solution, amoungst other benefits. Essentially, all we have to do is multiply through by our test function $v$ and integrate. Something *very* useful happens to our divergence-of-flux terms, which are prevelant in conservation equations.

As an example, consider a time dependant heat balance (temperature = \(u\)), using the backward Euler time marching scheme. $
\( \begin{align} \frac{\partial u}{\partial t} &= -\nabla \cdot \vec{J} \\ \frac{u - u^{t-1}}{\Delta t} &= -\nabla \cdot \vec{J} \\ u + \Delta t [\nabla \cdot \vec{J}] &= u^{t-1} \end{align}\)
$

To get to the weak form, multiply through by the test funciton \(v\) and integrate:

$$ 0 = \int_{\Omega} u v - \int_{\Omega} u^{t-1}v \ + \Delta t \int_{\Omega}v \nabla \cdot \vec{J} \ $$

Examining the divergence-of-flux term, we see that we can apply integration by parts followed by the Divergence Theorem: $
\( \begin{align} \int_\Omega v \nabla \cdot \vec{J} &= -\int_\Omega \nabla v \cdot \vec{J} + \int_{\Omega} \nabla \cdot \big[v\vec{J}\big] \\ &=-\int_\Omega \nabla v \cdot \vec{J} + \int_{\partial \Omega} v \ \vec{J} \cdot \hat{n} \\ &=\int_\Omega \lambda \nabla v \nabla u + \int_{\partial \Omega} v \ J_n \end{align}\)
$

Where we have defined the outward normal boundary flux, \(J_n = \vec{J} \cdot \hat{n}\)

The weak form is remarkable since we:

- replaced a second derivative with the product of two first derivatives.
- no longer require our solution to *have* a second derivative (it can be less smooth and satisfy the equation in an average sense which is *weaker* than the *strong* form)!
- incorporated boundary fluxes directly into the residual calculation (through the surface integral)
- do not require anything from \(\lambda\), which means this is still valid for \(\lambda(u,x,t...)\), however it is now outside of the gradient!
- We also provided an avenue for directly integrating surface *phenomena* into our model through the surface integral!)

### Boundary conditions / terms

The term \(\int_{\partial \Omega} v \ \vec{J} \cdot \hat{n}\) is a *boundary integral* of the *normal flux*, \(\vec{J}_n = \vec{J} \cdot \hat{n}\).

Typically, we will have some knowledge of the boundary conditions either:

- Neumann (flux) boundary condition in which case we have the quantity \(\vec{J}_n\) to insert directly.
- Insulation (zero flux) for which \(\vec{J}_n= 0\) and the term vanishes.
- Dirichlet (value) boundary condition which we would just insert directly into the applicable unknown, e.g.: \(u_0 = 0\) for a boundary \(u(0) = 0\). In this case, there is still a flux at that boundary but it is determined by the dirichlet condition.

Some software will opt to implement Dirichlet conditions as a *weak constraint* which is a Lagrange multiplier formulation in which the lagrange multiple assume the value of the *reaction force* necesssary to affect that constraint.

## Discretization

We are now in a position to discretize the weak equation ,

$$\begin{split} \begin{align} \int_{\Omega} u v + \Delta t \int_\Omega \lambda \nabla v \nabla u &= \int_{\Omega} u^{t-1}v - \Delta t \int_{\partial \Omega} v \ J_n\\ \end{align}\end{split}$$
into a form we are able to solve, which will (finally) give us an idea of what the shape functions are.

### Small support for shape functions

The shape functions are generally designed to be non-zero only inside their element. This has the effect of limiting products, \(\int \phi_i \phi_j\) and \(\int \nabla \phi_i \nabla \phi_j\) to be zero for almost all combination of \(i\) and \(j\) except for cases where the nodes \(i\) and \(j\) are within the same element!

## Assembly

The matrix assembly stage involves transforming the weak form into a matrix equation for the unknowns, $u_i$ such that the approximate solution can be assembled as $u_h = \sum u_i \phi_i$.

Consider the term, $\int_{\Omega} uv$, recalling $u_h = \sum u_i \phi_i$ and $v = \sum_j \phi_j$. Since integration is a linear operator, we have,

$$\begin{split}\begin{align} \int_{\Omega} uv &= \sum_i u_i \int_{\Omega} \phi_i \phi_j \\ &= \sum_i u_i M_{ij} \\ \end{align}\end{split}$$
where $M_{ij}$ is called the *Mass* matrix.

Similarly,

$$\begin{split}\begin{align} \int_\Omega \lambda \nabla v \nabla u &= \sum_i \lambda(u_i) u_i \int_{\Omega} \nabla \phi_i \nabla \phi_j \\ &= \sum_i \lambda(u_i) u_i K_{ij} \end{align}\end{split}$$
which is called the *stiffness matrix*.

The term $
$$\begin{align} \int_{\Omega} u^{t-1} v &= \sum_i u^{t-1}_i \int_{\Omega} \phi_i \phi_j \\ &= \sum_i u_i^{t-1} M_{ij} = b_j \end{align}$$
$ but can also be calculated directly since $u_i^{t-1}$ is known.

Finally we can assemble our equation:

$$\begin{split} \begin{align} \int_{\Omega} u v + \Delta t \int_\Omega \lambda \nabla v \nabla u &= \int_{\Omega} u^{t-1}v - \Delta t \int_{\partial \Omega} v \ J_n\\ \sum_i u_i M_{ij} + \Delta t \lambda u_i K_{ij} &= b_j - {J_n}_j \\ \vec{u}_h M + \Delta t \lambda \vec{u}_h K &= \vec{b} - \vec{J_n} \end{align}\end{split}$$
The elements of the matrix are can be calculated for a reference element and then simply scaled according to the transformation of the template to the mesh. The integration itself can be analytic, or Gaussian Quadrature (for which Gauss Points would be obtained on the reference element).

### Summary of FEM Matricies

| Weak Form Term …… | Matrix Form (FEM) | Description |
|---|---|---|
| $\int_\Omega \phi_i \phi_j \, d\Omega$ | $\mathbf{M}_{ij} \$ | Mass matrix. Often used in time-dependent problems. |
| $\int_\Omega \nabla \phi_i \cdot \nabla \phi_j \, d\Omega$ | $\mathbf{K}_{ij} \$ | Stiffness matrix. Used in problems like Poisson or elasticity equations. |
| $\int_\Omega \phi_i f \, d\Omega$ | $\mathbf{F}_i \$ | Load vector. Represents source terms in the governing equation. |
| $\int_{\partial \Omega} \phi_i g \, d\partial \Omega$ | $\mathbf{F}_i^{\text{bc}} \$ | Boundary load vector (from Neumann boundary conditions). |

## Solution

We now arrive at a matrix equation for the unknows $u_i$, which is mearly a system of equations as we've spoken about beforehand.

### Linear PDE

If the PDE is linear, we can solve it with a linear solver which is sparse due to the small support requirement of the elements. The matrix will not in general be banded however, due to the irregular conectivity of the mesh verticies.

**Example: 1D heat equation**

Solve the 1D heat balance equation with \(\lambda = 1\) for \(x\) from 0 to 10 subject to \(J(0)\) = 1 and \(T(10) = 0\). Initial condition is \(T(x,t=0) = 0\).

The mass matrix is: $\(M_{ij} = \int_{x_1}^{x_2} \phi_i(x) \phi_j(x) dx\)$

with

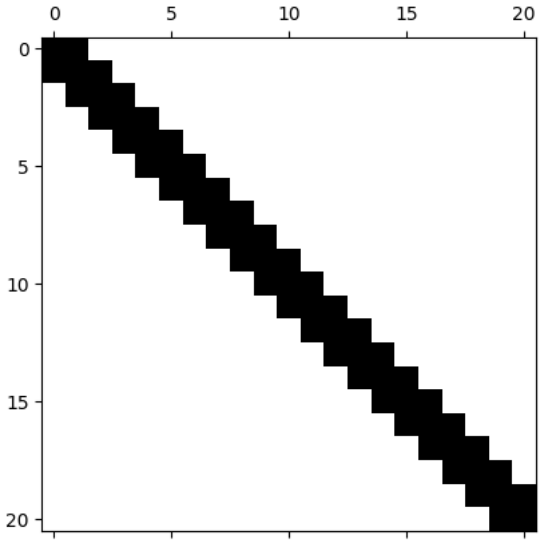
\[\begin{split}\begin{align} \phi_1(x) &= \frac{x_2-x}{x_2-x_1} \\ \phi_2(x) &= \frac{x-x_1}{x_2-x_1} \\ \end{align}\end{split}\]
we (Mathematica) get: $\( M = \frac{[x_2-x_1]}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \)$

The stiffness matrix needs $
\(\begin{align} \nabla \phi_1(x) &= \frac{-1}{x_2-x_1} \\ \nabla \phi_2(x) &= \frac{1}{x_2-x_1} \\ \end{align}\)$

and $
\( \begin{align} K_{ij} &= \lambda \int_{x_1}^{x_2} \nabla \phi_i(x) \nabla \phi_j(x) dx\\ &= \frac{\lambda}{h} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \end{align}\)
$

These *blocks* belong to each element, so we have to build the full matrix by *summing them*. Note this implies an overlap at common verticies between elements!

```python
# prompt: Assemble a a linear system for the heat equation above using the finite element method with La

import numpy as np
import matplotlib.pyplot as plt
from scipy import sparse
from ipywidgets import interact, FloatSlider

# Define the domain and parameters
L = 10.0  # Length of the domain
n_elements = 20  # Number of elements
dt = 0.1  # Time step
t_final = 1.0  # Final time

# Create a mesh with concentration near the left boundary
x_nodes = np.concatenate((np.linspace(0, 2, int(n_elements / 2) + 1),
                          np.linspace(2 + (L - 2) / (n_elements / 2 - 1), L, int(n_elements / 2))))

# Define the element stiffness and mass matrices for linear elements
def element_stiffness(x1, x2):
    return np.array([[1, -1], [-1, 1]]) / (x2 - x1)

def element_mass(x1, x2):
    return np.array([[2, 1], [1, 2]]) * (x2 - x1) / 6

# Assemble the global stiffness and mass matrices
K = np.zeros((len(x_nodes), len(x_nodes)))
M = np.zeros((len(x_nodes), len(x_nodes)))

for i in range(len(x_nodes) - 1):
    x1 = x_nodes[i]
    x2 = x_nodes[i + 1]
    Ke = element_stiffness(x1, x2)
    Me = element_mass(x1, x2)
    K[i:i + 2, i:i + 2] += Ke
    M[i:i + 2, i:i + 2] += Me

plt.spy(M)
plt.show()

# Apply boundary conditions
# Dirichlet BC at x = L (T(L) = 0)
K[-1, :] = 0
K[-1, -1] = 1
M[-1, :] = 0
M[-1, -1] = 1

# Neumann BC at x = 0 (J(0) = 1)
# We'll implement this in the right-hand side vector later.

# Function to solve and plot for a given time
def solve_and_plot(t_final):
    T = np.zeros(len(x_nodes))
    t = 0.0
    while t < t_final:
        b = M @ T
        b[0] += dt * 1
        T = np.linalg.solve(M + dt * K, b)
        t += dt

    plt.plot(x_nodes, T)
    plt.xlabel('x')
    plt.ylim(-.5, 10)  # Set the y-axis limits to [0, 1]
    plt.ylabel('Temperature')
    plt.title('Temperature Profile at t = {}'.format(t_final))
    plt.show()


# Create a slider for the final time
time_slider = FloatSlider(value=0.0, min=0.0, max=100.0, step=1, description='Final Time:')

# Use interact to link the slider to the function
interact(solve_and_plot, t_final=time_slider);
```




Note that we built the \(M\) and \(K\) matricies once, outside the time loop and reuse them for each time step! Even if we had adaptive time stepping, the matrix wouldn't change!

## Nonlinear case

If the PDE is nonlinear, we simply have to use a root finder. Note that in this case we are looking for the parameters \(u_i\) directly - the shape functions are unaffected and therefore only the coefficients of the matrix *blocks* are altered. This brings up the concept of *sparsity patterns* which can help reduce the overhead of sparse matrix representaitons.

Since modern finite element software needs to construct the weak form, it is usually equiped with symbolic logic capabilities and therefore able to calculate the Jacobian for use in the root finding algorithm.

#### Example: 1D nonlinear heat equation

Repeat the above exercise with \(\lambda = 1 + u/10\)

```python
# prompt: Solve the heat transport problem again but this time with \lambda = 1+T/10

import numpy as np
import matplotlib.pyplot as plt
from scipy import sparse
from ipywidgets import interact, FloatSlider

# Define the domain and parameters
L = 10.0  # Length of the domain
n_elements = 20  # Number of elements
dt = 0.1  # Time step
t_final = 1.0  # Final time

# Create a mesh with concentration near the left boundary
x_nodes = np.concatenate((np.linspace(0, 2, int(n_elements / 2) + 1),
                          np.linspace(2 + (L - 2) / (n_elements / 2 - 1), L, int(n_elements / 2))))

# Define the element stiffness and mass matrices for linear elements
def element_stiffness(x1, x2, T):
    lambda_val = 1 + T / 10
    return np.array([[1, -1], [-1, 1]]) * lambda_val / (x2 - x1)


def element_mass(x1, x2):
    return np.array([[2, 1], [1, 2]]) * (x2 - x1) / 6

# Assemble the global stiffness and mass matrices
def assemble_matrices(T):
    K = np.zeros((len(x_nodes), len(x_nodes)))
    M = np.zeros((len(x_nodes), len(x_nodes)))

    for i in range(len(x_nodes) - 1):
        x1 = x_nodes[i]
        x2 = x_nodes[i + 1]
        Ke = element_stiffness(x1, x2, T[i:i+2].mean())
        Me = element_mass(x1, x2)
        K[i:i + 2, i:i + 2] += Ke
        M[i:i + 2, i:i + 2] += Me

    return K, M


# Apply boundary conditions
def apply_boundary_conditions(K, M):
    # Dirichlet BC at x = L (T(L) = 0)
    K[-1, :] = 0
    K[-1, -1] = 1
    M[-1, :] = 0
    M[-1, -1] = 1
    return K, M

# Function to solve and plot for a given time
def solve_and_plot(t_final):
    T = np.zeros(len(x_nodes))
    t = 0.0
    while t < t_final:
        K, M = assemble_matrices(T)
        K, M = apply_boundary_conditions(K, M)
        b = M @ T
        b[0] += dt * 1
        T = np.linalg.solve(M + dt * K, b)
        t += dt

    plt.plot(x_nodes, T)
    plt.xlabel('x')
    plt.ylim(-.5, 10)  # Set the y-axis limits to [0, 1]
    plt.ylabel('Temperature')
    plt.title('Temperature Profile at t = {}'.format(t_final))
    plt.show()


# Create a slider for the final time
time_slider = FloatSlider(value=0.0, min=0.0, max=100.0, step=1, description='Final Time:')

# Use interact to link the slider to the function
interact(solve_and_plot, t_final=time_slider);
```

Hmmmm… something is funny in that code!

```python
# prompt: Solve the heat transport problem again but this time with \lambda = 1+T/10 using a root finder

import numpy as np
import matplotlib.pyplot as plt
from scipy import sparse
from scipy.optimize import root
from ipywidgets import interact, FloatSlider


# Define the domain and parameters
L = 10.0  # Length of the domain
n_elements = 20  # Number of elements
dt = 0.1  # Time step
t_final = 1.0  # Final time

# Create a mesh with concentration near the left boundary
x_nodes = np.concatenate((np.linspace(0, 2, int(n_elements / 2) + 1),
                          np.linspace(2 + (L - 2) / (n_elements / 2 - 1), L, int(n_elements / 2))))

# Define the element stiffness and mass matrices for linear elements
def element_stiffness(x1, x2, lambda_val):
    return np.array([[1, -1], [-1, 1]]) * lambda_val / (x2 - x1)

def element_mass(x1, x2):
    return np.array([[2, 1], [1, 2]]) * (x2 - x1) / 6

# Assemble the global stiffness and mass matrices
M = np.zeros((len(x_nodes), len(x_nodes)))

for i in range(len(x_nodes) - 1):
    x1 = x_nodes[i]
    x2 = x_nodes[i + 1]
    Me = element_mass(x1, x2)
    M[i:i + 2, i:i + 2] += Me

# Apply boundary conditions
# Dirichlet BC at x = L (T(L) = 0)
M[-1, :] = 0
M[-1, -1] = 1

# Neumann BC at x = 0 (J(0) = 1)
# We'll implement this in the right-hand side vector later.

# Function to solve for T at the next time step using a root finder
def solve_for_next_T(T_prev):
    def residual(T):
        K = np.zeros((len(x_nodes), len(x_nodes)))
        for i in range(len(x_nodes) - 1):
            x1 = x_nodes[i]
            x2 = x_nodes[i + 1]
            lambda_val = 1 + T[i] / 10  # Update lambda based on T
            Ke = element_stiffness(x1, x2, lambda_val)
            K[i:i + 2, i:i + 2] += Ke

        b = M @ T_prev
        b[0] += dt * 1
        b[-1] = 0
        return (M @ T) + dt * (K @ T) - b

    sol = root(residual, T_prev)
    return sol.x

# Function to solve and plot for a given time
def solve_and_plot(t_final):
    T = np.zeros(len(x_nodes))
    t = 0.0
    while t < t_final:
        T = solve_for_next_T(T)
        t += dt

    plt.plot(x_nodes, T)
    plt.xlabel('x')
    plt.ylim(-.5, 10)
    plt.ylabel('Temperature')
    plt.title('Temperature Profile at t = {}'.format(t_final))
    plt.show()


# Create a slider for the final time
time_slider = FloatSlider(value=0.0, min=0.0, max=100.0, step=1, description='Final Time:')
```

```
# Use interact to link the slider to the function
interact(solve_and_plot, t_final=time_slider);
```

# Solution

We now arrive at a matrix equation for the unknows \(u_i\), which is mearly a system of equations as we've spoken about beforehand.

## Linear PDE

If the PDE is linear, we can solve it with a linear solver which is sparse due to the small support requirement of the elements. The matrix will not in general be banded however, due to the irregular conectivity of the mesh verticies.

### Example: 1D heat equation

Solve the 1D heat balance equation with \(\lambda = 1\) for \(x\) from 0 to 10 subject to \(J(0)\) = 1 and \(T(10) = 0\). Initial condition is \(T(x,t=0) = 0\).

The mass matrix is: $\(M_{ij} = \int_{x_1}^{x_2} \phi_i(x) \phi_j(x) dx\)$

with

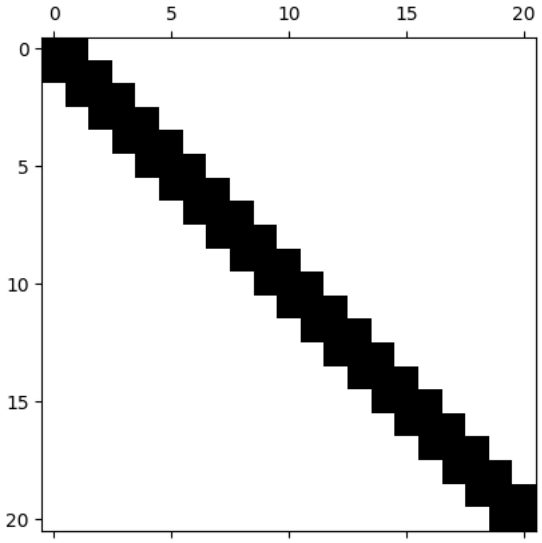
\[\begin{split}\begin{align} \phi_1(x) &= \frac{x_2-x}{x_2-x_1} \\ \phi_2(x) &= \frac{x-x_1}{x_2-x_1} \\ \end{align}\end{split}\]
we (Mathematica) get: $\( M = \frac{[x_2-x_1]}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \)$

The stiffness matrix needs $
\(\begin{align} \nabla \phi_1(x) &= \frac{-1}{x_2-x_1} \\ \nabla \phi_2(x) &= \frac{1}{x_2-x_1} \\ \end{align}\)$

and $
\( \begin{align} K_{ij} &= \lambda \int_{x_1}^{x_2} \nabla \phi_i(x) \nabla \phi_j(x) dx\\ &= \frac{\lambda}{h} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \end{align}\)
$

These *blocks* belong to each element, so we have to build the full matrix by *summing them*. Note this implies an overlap at common verticies between elements!

```python
# prompt: Assemble a a linear system for the heat equation above using the finite element method with La

import numpy as np
import matplotlib.pyplot as plt
from scipy import sparse
from ipywidgets import interact, FloatSlider

# Define the domain and parameters
L = 10.0  # Length of the domain
n_elements = 20  # Number of elements
dt = 0.1  # Time step
t_final = 1.0  # Final time

# Create a mesh with concentration near the left boundary
x_nodes = np.concatenate((np.linspace(0, 2, int(n_elements / 2) + 1),
                          np.linspace(2 + (L - 2) / (n_elements / 2 - 1), L, int(n_elements / 2))))

# Define the element stiffness and mass matrices for linear elements
def element_stiffness(x1, x2):
    return np.array([[1, -1], [-1, 1]]) / (x2 - x1)

def element_mass(x1, x2):
    return np.array([[2, 1], [1, 2]]) * (x2 - x1) / 6

# Assemble the global stiffness and mass matrices
K = np.zeros((len(x_nodes), len(x_nodes)))
M = np.zeros((len(x_nodes), len(x_nodes)))

for i in range(len(x_nodes) - 1):
    x1 = x_nodes[i]
    x2 = x_nodes[i + 1]
    Ke = element_stiffness(x1, x2)
    Me = element_mass(x1, x2)
    K[i:i + 2, i:i + 2] += Ke
    M[i:i + 2, i:i + 2] += Me

plt.spy(M)
plt.show()

# Apply boundary conditions
# Dirichlet BC at x = L (T(L) = 0)
K[-1, :] = 0
K[-1, -1] = 1
M[-1, :] = 0
M[-1, -1] = 1

# Neumann BC at x = 0 (J(0) = 1)
# We'll implement this in the right-hand side vector later.

# Function to solve and plot for a given time
def solve_and_plot(t_final):
    T = np.zeros(len(x_nodes))
    t = 0.0
    while t < t_final:
        b = M @ T
        b[0] += dt * 1
        T = np.linalg.solve(M + dt * K, b)
        t += dt

    plt.plot(x_nodes, T)
    plt.xlabel('x')
    plt.ylim(-.5, 10)  # Set the y-axis limits to [0, 1]
    plt.ylabel('Temperature')
    plt.title('Temperature Profile at t = {}'.format(t_final))
    plt.show()


# Create a slider for the final time
time_slider = FloatSlider(value=0.0, min=0.0, max=100.0, step=1, description='Final Time:')

# Use interact to link the slider to the function
interact(solve_and_plot, t_final=time_slider);
```




Note that we built the \(M\) and \(K\) matricies once, outside the time loop and reuse them for each time step! Even if we had adaptive time stepping, the matrix wouldn't change!

## Nonlinear case

If the PDE is nonlinear, we simply have to use a root finder. Note that in this case we are looking for the parameters \(u_i\) directly - the shape functions are unaffected and therefore only the coefficients of the matrix *blocks* are altered. This brings up the concept of *sparsity patterns* which can help reduce the overhead of sparse matrix representaitons.

Since modern finite element software needs to construct the weak form, it is usually equiped with symbolic logic capabilities and therefore able to calculate the Jacobian for use in the root finding algorithm.

### Example: 1D nonlinear heat equation

Repeat the above exercise with \(\lambda = 1 + u/10\)

```
# prompt: Solve the heat transport problem again but this time with \lambda = 1+T/10

import numpy as np
import matplotlib.pyplot as plt
from scipy import sparse
from ipywidgets import interact, FloatSlider

# Define the domain and parameters
L = 10.0  # Length of the domain
n_elements = 20  # Number of elements
dt = 0.1  # Time step
t_final = 1.0  # Final time

# Create a mesh with concentration near the left boundary
x_nodes = np.concatenate((np.linspace(0, 2, int(n_elements / 2) + 1),
                          np.linspace(2 + (L - 2) / (n_elements / 2 - 1), L, int(n_elements / 2))))

# Define the element stiffness and mass matrices for linear elements
def element_stiffness(x1, x2, T):
    lambda_val = 1 + T / 10
    return np.array([[1, -1], [-1, 1]]) * lambda_val / (x2 - x1)


def element_mass(x1, x2):
    return np.array([[2, 1], [1, 2]]) * (x2 - x1) / 6

# Assemble the global stiffness and mass matrices
def assemble_matrices(T):
    K = np.zeros((len(x_nodes), len(x_nodes)))
    M = np.zeros((len(x_nodes), len(x_nodes)))

    for i in range(len(x_nodes) - 1):
        x1 = x_nodes[i]
        x2 = x_nodes[i + 1]
        Ke = element_stiffness(x1, x2, T[i:i+2].mean())
        Me = element_mass(x1, x2)
        K[i:i + 2, i:i + 2] += Ke
        M[i:i + 2, i:i + 2] += Me

    return K, M


# Apply boundary conditions
def apply_boundary_conditions(K, M):
    # Dirichlet BC at x = L (T(L) = 0)
    K[-1, :] = 0
    K[-1, -1] = 1
    M[-1, :] = 0
    M[-1, -1] = 1
    return K, M

# Function to solve and plot for a given time
def solve_and_plot(t_final):
    T = np.zeros(len(x_nodes))
    t = 0.0
    while t < t_final:
        K, M = assemble_matrices(T)
        K, M = apply_boundary_conditions(K, M)
        b = M @ T
        b[0] += dt * 1
        T = np.linalg.solve(M + dt * K, b)
        t += dt

    plt.plot(x_nodes, T)
    plt.xlabel('x')
    plt.ylim(-.5, 10)  # Set the y-axis limits to [0, 1]
    plt.ylabel('Temperature')
    plt.title('Temperature Profile at t = {}'.format(t_final))
    plt.show()


# Create a slider for the final time
time_slider = FloatSlider(value=0.0, min=0.0, max=100.0, step=1, description='Final Time:')

# Use interact to link the slider to the function
interact(solve_and_plot, t_final=time_slider);
```

Hmmmm… something is funny in that code!

```python
# prompt: Solve the heat transport problem again but this time with \lambda = 1+T/10 using a root finder

import numpy as np
import matplotlib.pyplot as plt
from scipy import sparse
from scipy.optimize import root
from ipywidgets import interact, FloatSlider


# Define the domain and parameters
L = 10.0  # Length of the domain
n_elements = 20  # Number of elements
dt = 0.1  # Time step
t_final = 1.0  # Final time

# Create a mesh with concentration near the left boundary
x_nodes = np.concatenate((np.linspace(0, 2, int(n_elements / 2) + 1),
                          np.linspace(2 + (L - 2) / (n_elements / 2 - 1), L, int(n_elements / 2))))

# Define the element stiffness and mass matrices for linear elements
def element_stiffness(x1, x2, lambda_val):
    return np.array([[1, -1], [-1, 1]]) * lambda_val / (x2 - x1)

def element_mass(x1, x2):
    return np.array([[2, 1], [1, 2]]) * (x2 - x1) / 6

# Assemble the global stiffness and mass matrices
M = np.zeros((len(x_nodes), len(x_nodes)))

for i in range(len(x_nodes) - 1):
    x1 = x_nodes[i]
    x2 = x_nodes[i + 1]
    Me = element_mass(x1, x2)
    M[i:i + 2, i:i + 2] += Me

# Apply boundary conditions
# Dirichlet BC at x = L (T(L) = 0)
M[-1, :] = 0
M[-1, -1] = 1

# Neumann BC at x = 0 (J(0) = 1)
# We'll implement this in the right-hand side vector later.

# Function to solve for T at the next time step using a root finder
def solve_for_next_T(T_prev):
    def residual(T):
        K = np.zeros((len(x_nodes), len(x_nodes)))
        for i in range(len(x_nodes) - 1):
            x1 = x_nodes[i]
            x2 = x_nodes[i + 1]
            lambda_val = 1 + T[i] / 10  # Update lambda based on T
            Ke = element_stiffness(x1, x2, lambda_val)
            K[i:i + 2, i:i + 2] += Ke

        b = M @ T_prev
        b[0] += dt * 1
        b[-1] = 0
        return (M @ T) + dt * (K @ T) - b

    sol = root(residual, T_prev)
    return sol.x

# Function to solve and plot for a given time
def solve_and_plot(t_final):
    T = np.zeros(len(x_nodes))
    t = 0.0
    while t < t_final:
        T = solve_for_next_T(T)
        t += dt

    plt.plot(x_nodes, T)
    plt.xlabel('x')
    plt.ylim(-.5, 10)
    plt.ylabel('Temperature')
    plt.title('Temperature Profile at t = {}'.format(t_final))
    plt.show()


# Create a slider for the final time
time_slider = FloatSlider(value=0.0, min=0.0, max=100.0, step=1, description='Final Time:')
```

```
# Use interact to link the slider to the function
interact(solve_and_plot, t_final=time_slider);
```