Numerical Methods II APPM2007

Dr Matthew Woolway

2018-10-09

Contents

List of Tables

List of Figures

Course Outline

WITS UNIVERSITY

Course Structure and Details

- **Office: UG3 - Maths Science Building (MSB)**
- **Consultation: Wednesdays - 12:30 - 14:15** (This is for all three topics)
- **Lecture Venues: P115 - Tuesdays (10:15 - 11:30)**

Course Assessment

- There will be two tests and **no** assignment
- There will be labs. These **may/may not** count for extra marks
- The programming language will be **Python**

Course Topics

We will be covering the following topics throughout the course:

- Numerical differentiation
- Numerical Integration
- Numerical solutions to nonlinear equations
- Numerical solutions to systems of nonlinear equations
- Eigenvalues and eigenvectors
- Interpolation and least squares approximation
- Numerical solutions of ordinary differential equations and boundary value problems

Hardware Requirements

The course will be very computational in nature, however, you do not need your own personal machine. MSL already has **Python** installed. The labs will be running the IDEs for Python (along with Jupyter) while I will be using Jupyter for easier presentation and explanation in lectures. You will at some point need to become familiar with Jupyter as the tests will be conducted in the Maths Science Labs (MSL) utilising this platform for autograding purposes.

If you do have your own machine and would prefer to work from that you are more than welcome. Since all the notes and code will be presented through Jupyter please follow the following steps:

- Install Anaconda from here: https://repo.continuum.io/archive/Anaconda3-5.2.0-Windows-x86_64.exe
  - Make sure when installing Anaconda to set the installation to PATH when prompted (it will be deselected by default)
- To launch a Jupyter notebook, open the command promt (cmd) and type jupyter notebook. This should launch the browser and jupyter. If you see any proxy issues while on campus, then you will need to set the proxy to exclude the localhost.

If you are not running Windows but rather Linux, then you can get Anaconda at: https://repo.continuum.io/archive/Anaconda3-5.2.0-Linux-x86_64.sh

---

Chapter 1

Numerical Differentiation

In certain situations it is difficult to work with the actual derivative of a function. In some cases a derivative may fail to exist at a point. Another situation is when dealing with a function represented only by data and no analytic expression. In such situations it is desirable to be able to approximate the derivative from the available information. Presented below are methods used approximate $f'(x)$.

Numerical differentiation is not a particularly accurate process. It suffers from round-off errors (due to machine precision) and errors through interpolation. Therefore, a derivative of a function can never be computed with the same precision as the function itself.

1.1 Finite Difference Methods

Definition 1.1. The derivative of $y = f(x)$ is:

$$\frac{dy}{dx} = f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{1.1}$$

The above definition uses values of f near the point of differentiation, x. To obtain a formula approximating the first derivative, we typically use equally spaced points in the neighbourhood (e.g. $x-h, x, x+h$, where h is some small positive value) and construct an approximation from these values for the function $f(x)$ at these points.

This obviously leads to an error caused by the discretisation (**truncation error**). This error will decrease as h decrease, provided that the function $f(x)$ is sufficiently smooth.

1.1.1 Approximations to $f'(x)$

Given a smooth function $f : \mathbb{R} \to \mathbb{R}$, we wish to approximate its first and second derivatives at a point x. Consider the Taylor series expansions:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f'''(x)}{6}h^3 + \ldots, \tag{1.2}$$

and

$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2}h^2 - \frac{f'''(x)}{6}h^3 + \ldots. \tag{1.3}$$

Solving for $f'(x)$ in Equation (1.2), we obtain the **Forward Difference Formula**:

$$f'(x) = \frac{f(x+h)-f(x)}{h} - \frac{f''(x)}{2}h + \ldots \approx \frac{f(x+h)-f(x)}{h}, \tag{1.4}$$

which gives an approximation that is first-order accurate since the dominant term in the remainder of the series is $\mathcal{O}(h)$. Here the associated error is $-\frac{h}{2}f''(\epsilon)$.

Similarly, from Equation (1.3) we derive the **Backward Difference Formula**:

$$f'(x) = \frac{f(x)-f(x-h)}{h} + \frac{f''(x)}{2}h + \ldots \approx \frac{f(x)-f(x-h)}{h}, \tag{1.5}$$

which is also $\mathcal{O}(h)$. Here the associated error is $\frac{h}{2}f''(\epsilon)$.

Now, subtracting Equation (1.3) from Equation (1.2) gives the **Central Difference Formula**:

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} - \frac{f'''(x)}{6}h^2 + \ldots \approx \frac{f(x+h)-f(x-h)}{2h}, \tag{1.6}$$

which is second order accurate, i.e. $\mathcal{O}(h^2)$. This approximation is a three-point formula (implying that three points are required to make an approximation). Here the truncation error is $-\frac{h^2}{6}f'''(\epsilon)$, where $x-h \le \epsilon \le x+h$.

1.1.2 Approximations to $f''(x)$

Adding Equation (1.3) to Equation (1.2) gives the **Central Difference Formula** for the **second** derivative:

$$f''(x) = \frac{f(x+h)-2f(x)+f(x-h)}{h^2} - \frac{f^4(x)}{12}h^2 + \ldots \approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2}, \tag{1.7}$$

which is second order accurate $\mathcal{O}(h^2)$. The associated truncation error here is $-\frac{h^2}{12}f^{iv}(\epsilon)$.

Of course we can keep using function values at further addition points, $x \pm 2h, x \pm 3h, \ldots$ etc. This gives us similar difference formulas but at much higher accuracy, or for high-order derivatives. The downside to these however, is that we require more function values. This may add much higher computational cost depending on the situation.

1.1.2.1 Mathematica Demonstration

Show Mathematica Demonstration

1.1.2.2 Example

Compute an approximation to $f'(1)$ for $f(x) = x^2\cos(x)$ using the central difference formula and $h = 0.1, 0.05, 0.025, 0.0125$.

```
from math import *
cfd = lambda f, x, h: (f(x + h) - f(x - h))/(2*h)
x   = 1
h   = [0.1, 0.05, 0.025, 0.0125, 0.00625]
f   = lambda x: (x**2)*cos(x)
for i in h:
    y = cfd(f, x, i)
    print('The derivative at x = 1 with h = {:.4f} is f^1(x) = {:.10f}'.format(i, y))
```

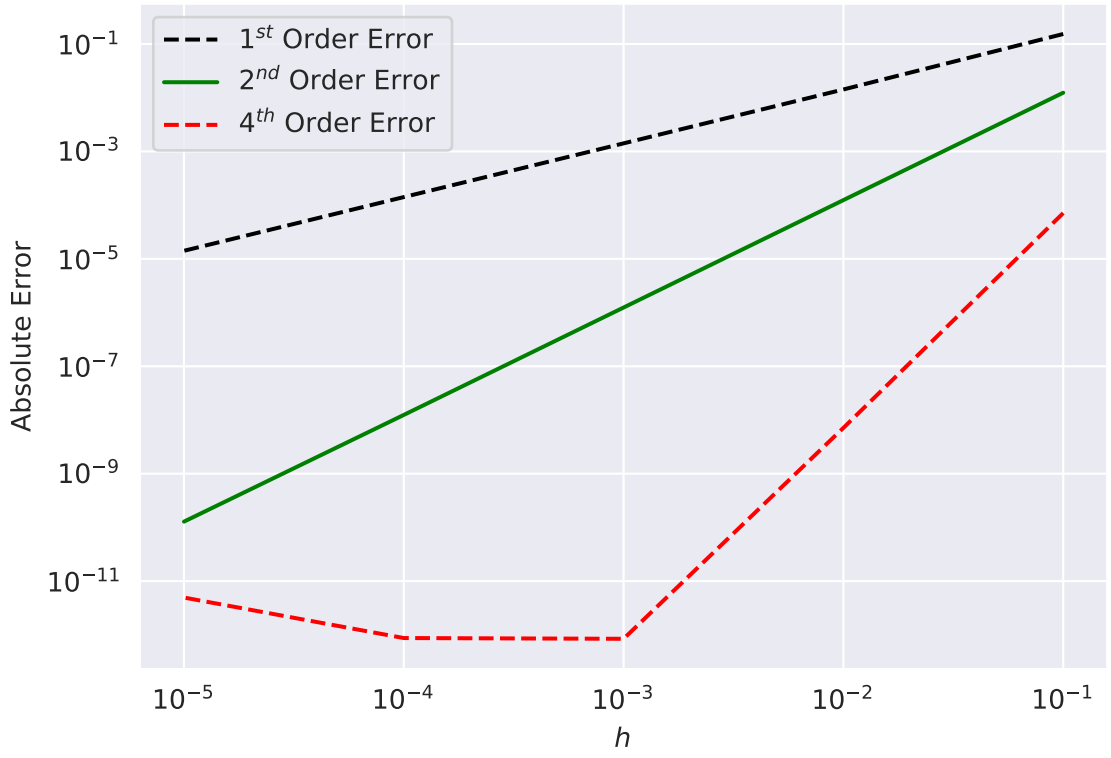
```
## The derivative at x = 1 with h = 0.1000 is f^1(x) = 0.2267361631
## The derivative at x = 1 with h = 0.0500 is f^1(x) = 0.2360309206
## The derivative at x = 1 with h = 0.0250 is f^1(x) = 0.2383577415
## The derivative at x = 1 with h = 0.0125 is f^1(x) = 0.2389396425
## The derivative at x = 1 with h = 0.0063 is f^1(x) = 0.2390851300
```

```
tans = 2*cos(1) - sin(1)
print('The true solution at x = 1 is: {:.10}'.format(tans))
```

```
## The true solution at x = 1 is: 0.2391336269
```

1.1.3 Errors in First and Second Order

How do the relevant errors look with regard to the example above? We can plot the absolute error of the approximations using both approaches compared to the true value. This is illustrated below:




```
## Error Values -  Tabulated
```

```
##          First Order  Second Order  Fourth Order
## 0.10000     0.153644  1.239746e-02  7.113648e-05
## 0.01000     0.014252  1.241510e-04  7.130152e-09
## 0.00100     0.001414  1.241528e-06  8.428813e-13
## 0.00010     0.000141  1.241594e-08  8.706647e-13
## 0.00001     0.000014  1.279912e-10  4.941492e-12
```

Notice the observed behaviour of the fourth order errors in the plot above. Does this seem expected? While intuitively we expect only the behaviour seem in the first and second orders errors something is causing a loss of

accuracy as h decreases with our fourth order approximation. The reason for this is inherent in the approximation formula. At small h the formula has instances of subtracting nearly equal numbers, and along with the loss of significant digits, this is exacerbated by the division of small numbers. This illustrates the effect that machine precision can have on our computations!

Optimal Stepsize:

If f is a function with continuous derivatives up to order 2, then the approximate stepsize h which minimises the total error(truncation + round-off error) of the derivative of f at x is:

$$h^* = 2\frac{\sqrt{\epsilon^*|f(x)|}}{\sqrt{|f''(x)|}}. \tag{1.8}$$

Here ϵ^* is the maximum relative error that occurs when real numbers are represented by floating-point numbers and there is no underflow or overflow. I useful rule-of-thumb estimate for ϵ^* is 7×10^{-17}.

1.2 Exercises

1. Consider the function $f(x) = \sin(x)$. Using the forward derivative approximation, compute an approximation for $f'(x)$ at $x = 0.5$. Investigate a gradual decrease in stepsize and determine a rough estimate for an optimal stepsize h, i.e. one which avoids round-off error. Plot the behaviour of these errors at the varying stepsizes. Does your estimation seem reasonable?
2. If we increased the number of bits for storing floating-point numbers, i.e. 128-bit precision, can we obtain better numerical approximations to derivatives?
3. The approximation $f'(x) \approx \dfrac{f(x+h)-f(x)}{h}$ will give the exact answer (assuming no round-off error) if the function f is linear?
4. Use the forward difference formula for $f'(x)$ to find an approximation for $f'(1)$ where $f(x) = \ln(x)$ and $h = 0.1, 0.01, 0.001$.
5. Use the central difference formula for $f'(x)$ to find an approximation for $f'(0)$ where $f(x) = \exp(x)$ and $h = 0.1, 0.01, 0.001$.

1.3 Richardson's Extrapolation

In numerical differentiation and soon to be seen integration, we are computing approximate values according to some stepsize. Clearly we would have an ideal case where the stepsize approaches zero as seen in our demo. However, due to rounding error this is simply not possible. Using nonzero stepsizes however, we may be able to estimate the what the value would be for a stepsize approaching zero. If we compute some value F from some stepsizes h_i and know something of its behaviour of F as $h \to 0$, then it may be possible to extrapolate from the known values an approximation of F at $h = 0$. This extrapolation will be of higher order accuracy than any of the originally used values.

In summary:

> **Richardson extrapolation method is a procedure which combines several approximations of a certain quantity to yield a more accurate approximation of that quantity.**

Suppose we are computing some quantity F and assume that the result depends on some stepsize h. Denoting the approximation by $f(h)$, we have $F = f(h) + E(h)$, where $E(h)$ represents an error. Richardson's extrapolation can

remove the error provided $E(h) = ch^p$, where c and p are constants. We start by computing $f(h)$ at some value of h, say h_1 giving:

$$F = f(h_1) + ch_1^p,$$

and another value $h = h_2$:

$$F = f(h_2) + ch_2^p.$$

Then solving the above equations for F we get:

$$F = \frac{(h_1/h_2)^p f(h_2) - f(h_1)}{(h_1/h_2)^p - 1},$$

which is the **Richardson's Extrapolation Formula**. In this course we will only consider half-steps, thus $h_2 = h_1/2$. This allows us to rewrite our formula as:

$$F = \frac{2^p f(h_1/2) - f(h_1)}{2^p - 1}.$$

Or more specifically, assuming we have some initial approximation $F_n(h)$, then we can obtain a general $n+1$th order formula $F_{n+1}(h)$:

$$F_{n+1}(h) = \frac{2^n F_n(h/2) - F_n(h)}{2^n - 1}.$$

To illustrate why this yields a higher order approximation, consider the the second order central difference formula:

$$F_2(h) = f'(x) = \frac{f(x+h) - f(x-h)}{2h},$$

using this extrapolation formula we can get a new formula for $f'(x)$ as:

$$\begin{aligned} F_4(h) &= \frac{2^2 F_2(h/2) - F_2(h)}{2^2 - 1}, \\ &= \left[4\frac{f(x+h/2) - f(x-h/2)}{h} - \frac{f(x+h) - f(x-h)}{2h}\right]/3, \\ &= \frac{f(x-h) - 8f(x-h/2) + 8f(x+h/2) - f(x+h)}{6h}, \end{aligned}$$

which is the five-point central difference formula which is of order four $\mathcal{O}(h^4)$.

In order to apply this strategy, we need only apply a cheap but high order approximation for an array for halving stepsizes in order to extrapolate a much high order accurate approximation. We can do this by building the **Richardson's Extrapolation Table**. To do this, let us rewrite our formula one last time:

$$F_j^i = \frac{1}{4^j - 1}\left(4^j F_{j-1}^i - F_{j-1}^{i-1}\right), \quad j = 1, 2, \ldots m,\ i = 1, 2, \ldots, n. \tag{1.9}$$

Here j denotes iteration of the extrapolation and i the particular stepsize.

So if we use our difference formulae to compute our initial approximations $F_1^1, F_1^2, \ldots, T_1^n$ (which we should try to use as higher an order as possible), then we use the above formula to build up the table.

1.3.1 Example

Build a Richardson's extrapolation table for $f(x) = x^2\cos(x)$ to evaluate $f'(1)$ for $h = 0.1, 0.05, 0.025, 0.0125$.

Solution:

We have:

$$
\begin{aligned}
F_1^1 &= F_0^2 + \frac{1}{3}(F_0^2 - F_0^1) = \frac{1}{3}(4F_0^2 - F_0^1) \\
F_1^2 &= F_0^3 + \frac{1}{3}(F_0^3 - F_0^2) = \frac{1}{3}(4F_0^3 - F_0^2) \\
F_1^3 &= F_0^4 + \frac{1}{3}(F_0^4 - F_0^3) = \frac{1}{3}(4F_0^4 - F_0^3) \\
F_2^1 &= F_1^2 + \frac{1}{15}(F_1^2 - F_1^1) = \frac{1}{15}(16F_1^2 - F_1^1) \\
F_2^2 &= F_1^3 + \frac{1}{15}(F_1^3 - F_1^2) = \frac{1}{15}(16F_1^3 - F_1^2) \\
F_3^1 &= F_2^2 + \frac{1}{63}(F_2^2 - F_2^1) = \frac{1}{63}(64F_2^2 - F_2^1)
\end{aligned}
$$

In Tabular form:

i	h_i	F_0^i	F_1^i	F_2^i	F_3^i
1	0.1	0.226736			
2	0.05	0.236031	0.239129		
3	0.025	0.238358	0.239133	0.239134	
4	0.0125	0.238938	0.239132	0.239132	0.239132

Note: The F_1^i values are computed from whatever was used for the inital approximation. In this case, it was the central difference approximation.

1.3.2 Exercises

1. Develop a two point backward difference formula for approximating $f'(x)$ including the error term.
2. Develop a second order method for approximating $f'(x)$ that only uses the data $f(x-h)$, $f(x)$ and $f(x+3h)$.
3. Extrapolate the formula obtained in exercise (2). Then demonstrate the order of this new formula by approximating $f'(\pi/3)$ where $f(x) = \sin(x)$ and $h = 0.1, 0.01$.
4. Use the forward difference formulas and backward difference formulas to determine the missing entries in the following table:

x	$f(x)$	$f'(x)$
0.5	0.4794	Compute
0.6	0.5646	Compute
0.7	0.6442	Compute

5. Using the times and positions recorded for a moving car below, compute the velocity of the car at all times listed:

Time (s)	0	3	5	8	10	13
Distance (m)	0	225	383	623	742	993

6. Apply Richard's extrapolation to determine $F_3(h)$, an approximation to $f'(x)$ for the the following function:

$$f(x) = 2^x \sin(x),\ x_0 = 1.05,\ h = 0.4$$

7. Use the centred difference formula to approximate the derivative of each of the following functions at the specified location and for the specified size:
   - $y = \tan x$ at $x = 4, h = 0.1$
   - $y = \sin(0.5\sqrt{x})$ at $x = 1, h = 0.125$

8. A jet fighter's position on an aircraft carrier's runway was timed during landing: where x is the distance from the end of the carrier, measured in metres and t is the time in seconds. Estimate the velocity and acceleration for each time point and plot these values accordingly.

t	0	0.51	1.03	1.74	2.36	3.24	3.82
x	154	186	209	250	262	272	274

9. The following data was collected when a large oil tanker was loading. Calculate the flow rate $Q = \dfrac{dV}{dt}$ for each time point.

t, min	0	15	30	45	60	90	120
V, 10^6 barrels	0.5	0.65	0.73	0.88	1.03	1.14	1.30

Chapter 2

Numerical Integration

A common problem is to evaluate the definite integral:

$$I = \int_a^b f(x)\,dx. \tag{2.1}$$

Here we wish to compute the area under a the curve $f(x)$ over an interval $[a,b]$ on the real line. The numerical approximation of definite integrals is known as **numerical quadrature**. We will consider the interval of integration to be finite and assume the integrand f is smooth and continuous.

Since integration is an infinite summation we will need to approximate this infinite sum by a finite sum. This finite sum involves sampling the integrand a some number of finite points within the interval, this is known as the **quadrature rule**. Thus, our goal is to determine which sample points to take and how to weight these in contribution to the quadrature formula. We can design these to a desired accuracy at which we are satisfied with the computational cost required. Generally, this computational cost is measured through the number of integrand function requirements undertaken. Importantly, Numerical Integration is *insensitive* to round-off error.

2.1 Quadrature Rules

An n-point quadrature formula has the form:

$$I = \int_a^b f(x)\,dx = \sum_{i=1}^{n} w_i f(x_i) + R_n. \tag{2.2}$$

The points x_i are the values at which f is evaluated (called nodes), the multipliers w_i (called weights) and the remainder R_n. To approximate the value of the integral we compute:

$$I = \sum_{i=1}^{n} w_i f(x_i), \tag{2.3}$$

giving the quadrature rule.

Methods of numerical integration are divided into two groups; (i) Newton-Cotes formulas and (ii) Gaussian Quadrature. Newton-Cotes formulas deal with evenly spaced nodes. They are generally used when $f(x)$ can be computed cheaply. With Gaussian Quadrature nodes are chosen to deliver the best possible accuracy. It requires less evaluations of the integrand and is often used when $f(x)$ is expensive to compute. It is also used when dealing with integrals containing singularities or infinite limits. In this course we will only be working with Newton-Cotes.

2.2 Newton-Cotes Quadrature

If the nodes x_i are equally spaced on the interval $[a,b]$, then the resultant quadrature rule is known as a **Newton-Cotes Quadrature rule**. A **closed Newton-Cotes rule** includes the endpoints a and b, if not, the rule is **closed**.

Consider the definite integral:

$$I = \int_a^b f(x)dx. \tag{2.4}$$

Dividing the interval of integration (a,b) into n equal intervals, each of length $h = (b-a)/n$, then we obtain our nodes $x_0, x_1, \ldots, x_n$. We then approximate $f(x)$ with an interpolant of degree n which intersects all the nodes. Thus:

$$I = \int_a^b f(x)dx \approx \int_a^b P_n(x)dx. \tag{2.5}$$

2.2.1 Trapezoidal Rule

This is the first and simplest of Newton–Cotes closed integration formulae. It corresponds to the case when the polynomial is of first degree. We partition the interval $[a,b]$ of integration into n subintervals of equal width, and with $n+1$ points $x_0,\ x_1,\ \cdots,\ x_n$, where $x_0 = a$ and $x_n = b$. Let

$$x_{i+1} - x_i = h = \frac{b-a}{n}, \quad i = 0,1,2,\cdots,n-1.$$

On each subinterval $[x_i, x_{i+1}]$, we approximate $f(x)$ with a first degree polynomial,

$$\begin{aligned} P_1(x) &= f_i + \frac{f_{i+1}-f_i}{x_{i+1}-x_i}(x-x_i) \\ &= f_i + \frac{f_{i+1}-f_i}{h}(x-x_i). \end{aligned}$$

Then we have:

$$\begin{aligned} \int_{x_i}^{x_{i+1}} f(x)dx &\approx \int_{x_i}^{x_{i+1}} P_1(x)dx \\ &= \int_{x_i}^{x_{i+1}} f_i + \frac{f_{i+1}-f_i}{h}(x-x_i)dx \\ &= hf_i + f_{i+1} - f_i\ over h\frac{h^2}{2} \\ &= \frac{h}{2}\left(f_i + f_{i+1}\right) \end{aligned}$$

Geometrically, the trapezoidal rule is equivalent to approximating the area of the trapezoid under the straight line connecting $f(x_i)$ and $f(x_{i+1})$. Summing over all subintervals and simplifying gives:

$$I = \int_a^b f(x)dx = \sum_{i=1}^{n}\int_{x_{i-1}}^{x_i} f(x)dx \approx \sum_{i}^{n} \frac{f(x_{i-1}) + f(x_i)}{2}h, \tag{2.6}$$

or:

$$I \approx \frac{h}{2}\left[f_0 + 2(f_1 + f_2 + \cdots + f_{n-1}) + f_n\right], \tag{2.7}$$

which is known as the **Composite Trapezoidal rule**. In practice we would always used composite trapezoidal rule since it is simply trapezoidal rule applied in a piecewise fashion. The error of the composite trapezoidal rule is the difference between the value of the integral and the computed numerical result:

$$E = \int_a^b f(x)dx - I, \tag{2.8}$$

So:

$$E_T = -\frac{(b-a)h^2}{12} f''(\epsilon), \quad \epsilon \in [a,b], \tag{2.9}$$

where ϵ is a point which exists between a and b. We can also see that the error is of order $\mathcal{O}(h^2)$. Therefore, if the integrand is concave then the error is negative and the trapezoidal rule overestimates the true value. Should the integrand be concave then the error is positive and we have underestimated the true value.

2.2.2 Example

Using the trapezoidal rule, evaluate:

$$\int_0^1 \frac{1}{1+x^2} dx = \frac{\pi}{4},$$

use $n = 6$, i.e. we need 7 nodes.

Solution:

Since $n = 6$ then $h = (1-0)/6 = 1/6$, therefore:

$$I \approx \frac{1}{12}\left[f_0 + 2(f_1 + f_2 + f_3 + f_4 + f_5) + f_6\right]$$

```
import numpy as np
from math import *
import warnings
warnings.filterwarnings("ignore")
trap = lambda f, x, h: (h/2)*(f(x[0]) + sum(2*f(x[1:-1])) +f(x[-1]))
f    = lambda x: (1 + x**2)**(-1)
print('Computed Inputs:')
```

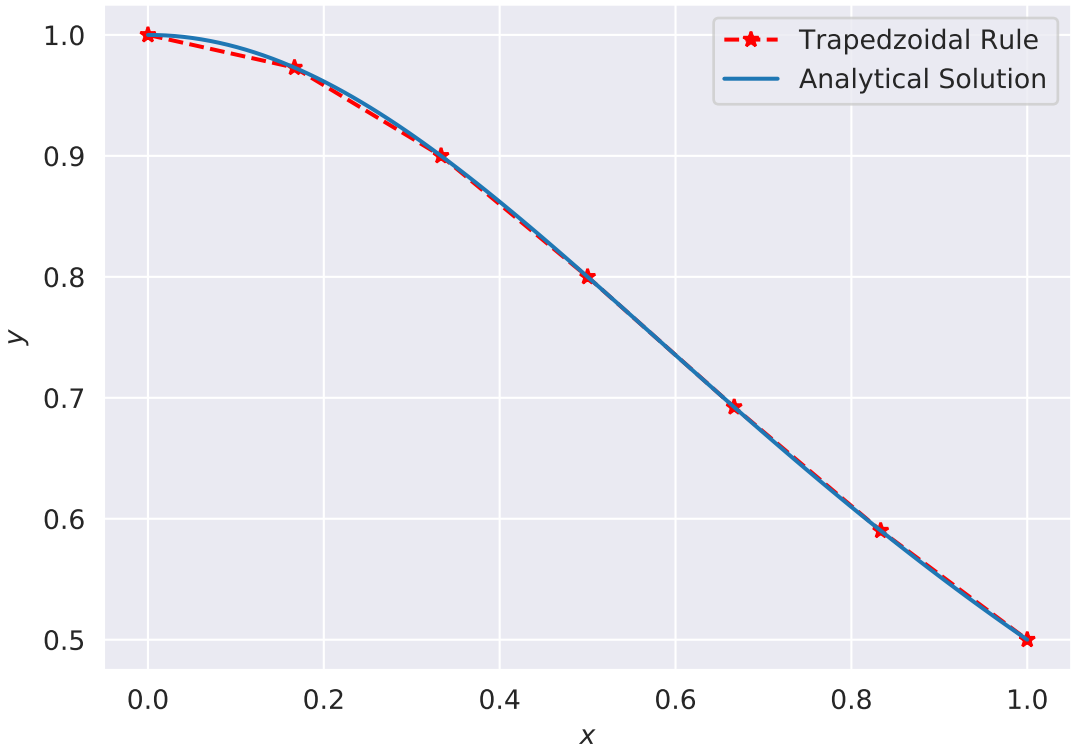
```
## Computed Inputs:
```

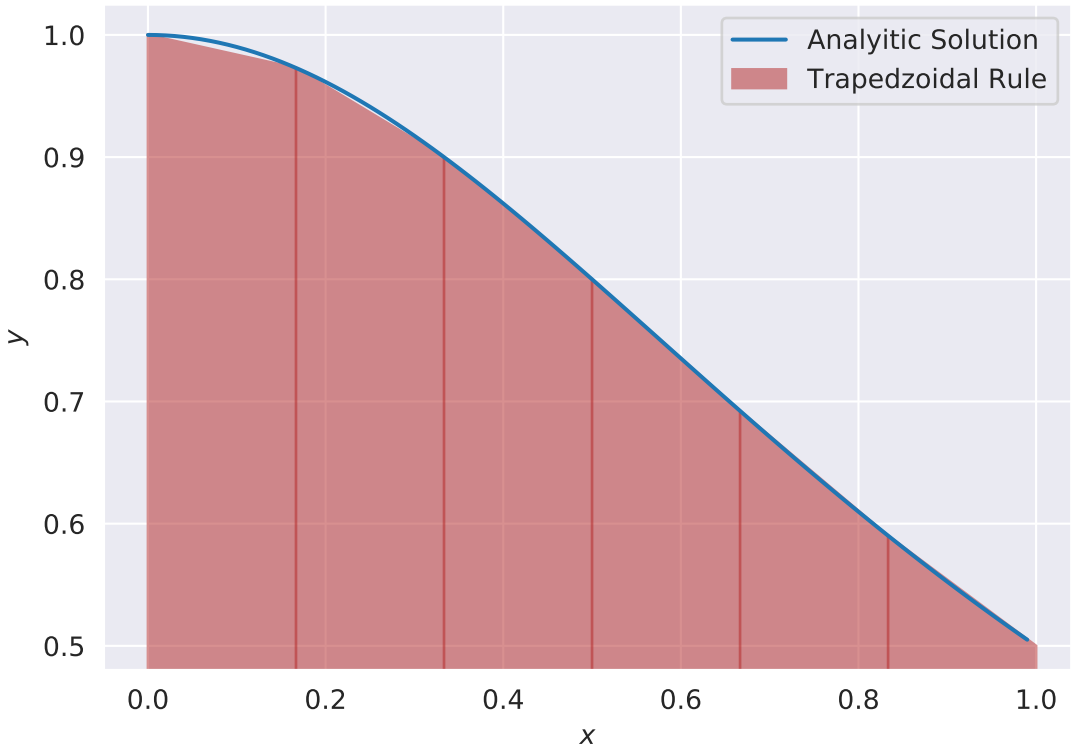
```
x    = np.linspace(0, 1, 7)
h    = 1/6
ans  = trap(f, x, h)
print('The trapezoidal method yields: {:.6f}'.format(ans))
```

```
## The trapezoidal method yields: 0.784241
```

```
tans = pi/4
print('The true answer:               {:.6f}'.format(tans))
```

```
## The true answer:               0.785398
```


Trapedzoidal Rule
Analytical Solution
1.0
0.9
0.8
0.7
0.6
0.5
y
0.0
0.2
0.4
0.6
0.8
1.0
x



2.2.2.1 Exercise

Use the trapezoidal rule on the following integral:

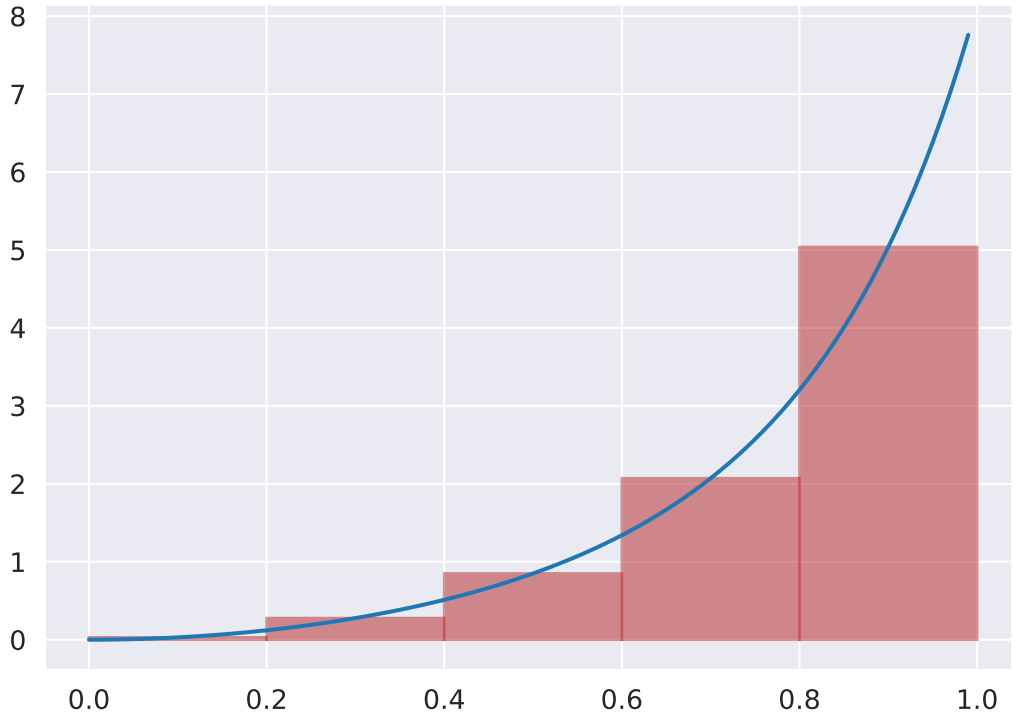
$$\int_0^1 \cos(x)dx,$$

by splitting the interval into 2^k subintervals, for $k = 1,2,\ldots,10$. Report the approximation juxtaposed to the corresponding approximation.

2.2.3 The Midpoint Method

Instead of approximating the area under the curve by trapezoids, we can also use rectangles. This may seem less accurate using horizontal lines versus skew ones, however, it is often more accurate.

In this approach, we construct a rectangle for every subinterval where the height equals f at the midpoint of the subinterval.



Let us now derive the general formula for the midpoint method given n rectangles of equal width:

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \ldots + \int_{x_{n-1}}^{x_n} f(x)dx, \\ &\approx hf\left(\frac{x_0+x_1}{2}\right) + hf\left(\frac{x_1+x_2}{2}\right) + \ldots + hf\left(\frac{x_{n-1}+x_n}{2}\right), \qquad (2.10)\\ &\approx h\left(f\left(\frac{x_0+x_1}{2}\right) + f\left(\frac{x_1+x_2}{2}\right) + \ldots + f\left(\frac{x_{n-1}+x_n}{2}\right)\right) \qquad (2.11)\end{aligned}$$

This can be rewritten as:

$$\int_a^b f(x)dx \approx h\sum_{i=0}^{n-1} f(x_i), \qquad (2.12)$$

where $x_i = (a + h/2) + ih$.

2.2.3.1 Comparing Trapezoidal Vs Midpoint Method

Consider the function $g(y) = e^{-y^2}$ on the domain $[0,2]$, i.e. we wish to compute:

$$\int_0^2 e^{-y^2}\,dy.$$

To compare to the two methods, we will increase the number of panels used in each method, from $n = 2$ to $n = 1048576$.

```
print('    n        midpoint          trapezoidal')
```

```
##     n        midpoint          trapezoidal
```

```
for i in range(1, 21):
    n = 2**i
    m = midpoint_method(g, a, b, n)
    t = trapezoidal(g, a, b, n)
    print('{:7d} {:.16f} {:.16f}'.format(n, m, t))
```

```
##       2 0.8842000076332692 0.8770372606158094
##       4 0.8827889485397279 0.8806186341245393
##       8 0.8822686991994210 0.8817037913321336
##      16 0.8821288703366458 0.8819862452657772
##      32 0.8820933014203766 0.8820575578012112
##      64 0.8820843709743319 0.8820754296107942
##     128 0.8820821359746071 0.8820799002925637
##     256 0.8820815770754198 0.8820810181335849
##     512 0.8820814373412922 0.8820812976045025
##    1024 0.8820814024071774 0.8820813674728968
##    2048 0.8820813936736116 0.8820813849400392
##    4096 0.8820813914902204 0.8820813893068272
##    8192 0.8820813909443684 0.8820813903985197
##   16384 0.8820813908079066 0.8820813906714446
##   32768 0.8820813907737911 0.8820813907396778
##   65536 0.8820813907652575 0.8820813907567422
##  131072 0.8820813907631487 0.8820813907610036
##  262144 0.8820813907625702 0.8820813907620528
##  524288 0.8820813907624605 0.8820813907623183
## 1048576 0.8820813907624268 0.8820813907623890
```

```
print('True Solution to 16 decimals is:', 0.882081390762422)
```

```
## True Solution to 16 decimals is: 0.882081390762422
```

A visual inspection of the numbers shows how fast the digits stabilise in both methods. It appears that 13 digits have stabilised in the last two rows.

2.2.4 Simpson's Rule

The trapezoidal rule approximates the area under a curve by summing over the areas of trapezoids formed by connecting successive points by straight lines. A more accurate estimate of the area can be achieved by using polynomials of higher degree to connect the points. Simpson's rule uses a second degree polynomial (parabola) to connect adjacent points. Interpolating polynomials are convenient for this approximation. So the interval $[a,b]$ is subdivided into an even number of equal subintervals (n is even). Next we pass a parabolic interpolant through through three adjacent nodes. Therefore our approximation is:

$$I = \frac{h}{3}\left[f_{i-1} + 4f_i + f_{i+1}\right]. \tag{2.13}$$

Summing the definite integrals over each subinterval $[x_{i-1}, x_{i+1}]$ for $i = 1,3,5,\cdots,n-1$ provides the approximation:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[(f_0 + 4f_1 + f_2) + (f_2 + 4f_3 + f_4) + \cdots + (f_{n-2} + 4f_{n-1} + f_n)\right] \tag{2.14}$$

By simplifying this sum we obtain the approximation scheme:

$$\begin{aligned}\int_a^b f(x)dx &\approx \frac{h}{3}\left[f_0+4f_1+2f_2+4f_3+\cdots+2f_{n-2}+4f_{n-1}+f_n\right]\\ &\approx \frac{h}{3}\left[f_0+4(f_1+f_3+\cdots f_{n-1})+2(f_2+f_4+\cdots+f_{n-2})+f_n\right] \qquad (2.15)\end{aligned}$$

This method of approximation is known as **Composite Simpson's 1/3 Rule**. The error for Simpson's rule is:

$$E_S = -\frac{(b-a)h^4}{180}f^4(\epsilon), \quad \epsilon \in [a,b], \qquad (2.16)$$

giving an error of order $\mathcal{O}(h^4)$. Hence if the integrand is of degree $n \leq 3$, then the error is zero and we obtain the exact value. The same can be said for the trapezoidal rule the integrand is linear.

2.2.4.1 Exercise

Apply Simpson's 1/3 rule on the following integral:

$$\int_0^1 \cos(x)dx,$$

by splitting the interval into 2^k subintervals, for $k = 1,2,\ldots,10$. Report the approximation juxtaposed to the corresponding approximation.

What can you note about the errors obtained here compared to when the above integral was solved using trapezoidal rule?

2.2.5 Convergence Rates

Often when implementation numerical approximations we may assume certain asymptotic behaviours when considering errors. For example, when implementing experimental results of the problem $\int_0^1 3t^2e^{t^3}dt$, where n is doubled in each run $n = 4,8,16$ using the trapezoidal rule, the errors where 12%, 3% and 0.77% respectively. This illustrates that the error was approximately reduced by a factor of 4 when n was doubled. Therefore, the error converges to zero as n^{-2} and we can say that the *convergence rate* is 2 (quadratic). Numerical integration methods usually have an error that converge to zero as n^p for some p that depends on the method. This implies that it does not matter what the actual approximation error is since we know at what rate it is reducing by. Therefore, running a method for two or more different n values would allows us to see if the expected rate is indeed achieved.

The idea of a corresponding unit test is then to run the algorithm for some n values, compute the error (the absolute value of the difference between the exact analytical result and the one produced by the numerical method), and check that the error has approximately correct asymptotic behaviour, i.e., that the error is proportional to n^2 in case of the trapezoidal and midpoint method.

More formally, assume that the error E depends on n according to:

$$E = Cn^r,$$

where C is an unknown constant and r is the convergence rate. Consider a set of experiments with various n, i.e. $n_0, n_1, \ldots, n_q$. We can compute the errors at each n, i.e. $E_0, E_1, \ldots, E_q$. Therefore, for two consecutive experiments, i and $i-1$, we have the error model:

$$E_i = Cn_i^r, \qquad (2.17)$$

$$E_{i-1} = Cn_{i-1}^r. \qquad (2.18)$$

These are two equations for two unknowns C and r. Eliminating C by dividing the equations by each other. Then solving for r gives:

$$r_{i-1} = \frac{\ln(E_i/E_{i-1})}{\ln(n_i/n_{i-1})}. \tag{2.19}$$

We have a subscript $i-1$ in r since the estimated value for r varies with i. Ideally, r_{i-1} approaches the correct convergence rate as the number of intervals.

2.2.6 Exercises

1. Since every point of measurement in the trapezoidal rule is used in two different subintervals, we must evaluate the function we want to integrate at every point twice. Is this a true statement to make?
2. Apply the trapezoidal rule to approximate the integral:

$$\int_0^1 x^2 dx,$$

   using only 2 intervals. What is the result?
3. Compute an approximation for the integral:

$$\int_0^{\pi/2} \frac{\sin(x)}{1+x^2} dx,$$

   with the trapezoidal rule and 6 subintervals.
4. Using the trapezoidal rule, along with 10 subintervals on the integral:

$$\int_0^1 e^x dx,$$

   determine a value of h which guarantees that the absolute error is smaller than 10^{-10}.
5. When using the trapezoidal rule and h is halved, some function values used with stepsize $h/2$ are the same as those used when the stepsize was h. Derive a formula for the trapezoidal rule with step length $h/2$ that allows one not to recompute the function values that were computed when the stepsize was h.
6. Is the Simpson's Rule exact for polynomials of degree 3 or lower?
7. Compute an approximation for the integral:

$$\int_0^{\pi/2} \frac{\sin(x)}{1+x^2} dx,$$

   with the Simpson's rule and 6 subintervals.
8. How many function evaluations does one need to calculate the integral:

$$\int_0^1 \frac{dx}{1+2x},$$

   with the trapezoidal rule to ensure that the error is smaller than 10^{-10}.
9. Repeat question 8 using the Simpson's rule.

2.3 Romberg Integration

This method of integration uses the trapezoidal rule to obtain the initial approximation to the integral followed by Richardson's approximation to obtain improvements.

We can show that for a trapezoidal approximation:

$$I = \int_a^b f(x)dx = T(h) + R(h), \quad R(h) = a_1h^2 + a_2h^4 + a_3h^6 + \cdots = \mathcal{O}(h^2),$$

where,

$$T(h) = \frac{h}{2}(f_0 + 2f_1 + 2f_2 + \cdots + 2f_{n-1} + f_n), \quad h = \frac{(b-a)}{n}.$$

Consider two trapezoidal approximations with spacing $2h$ and h and n is even.

$$I_2 = T(2h) + a_1(2h)^2 + a_2(2h)^4 + a_3(2h)^6 + \cdots \tag{2.20}$$

$$I_1 = T(h) + a_1h^2 + a_2h^4 + a_3h^6 + \cdots \tag{2.21}$$

If we subtract equation (2.20) from 4 times equation (2.21) we eliminate the leading error term (i.e. of $\mathcal{O}(h^2)$) and we get

$$I = \frac{1}{3}(4T(h) - T(2h)) + 4a_2h^4 + 20a_3h^6 + \cdots$$

after dividing right through by 3. But:

$$\begin{aligned}
\frac{1}{3}(4T(h) - T(2h)) &= \frac{h}{3}[(2f_0 + 4f_1 + 4f_2 + \cdots 4f_{n-1} + 2f_n) - (f_0 + 2f_2 + 2f_4 + \cdots 2f_{n-2} + f_n) \\
&= \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + \cdots 2f_{n-2} + 4f_{n-1} + f_n) \\
&= S(h),
\end{aligned}$$

which is the Simpson's rule, $S(h)$, for h, with an error $\mathcal{O}(h^4)$.

If we repeat for $h/2$, assuming that n is a multiple of 4, we have:

$$I_h = S(h) + c_1h^4 + c_2h^6 + \cdots \tag{2.22}$$

$$I_{h/2} = S(h/2) + c_1\left(\frac{h}{2}\right)^4 + c_2\left(\frac{h}{2}\right)^6 + \cdots \tag{2.23}$$

Multiply (2.23) by 16 and subtract (2.22) to get

$$I = \frac{16S(h/2) - S(h)}{15} + d_1h^6 + \cdots$$

which is now more accurate, with an error $\mathcal{O}(h^6)$.

We now generalise the results for $h_k = (b-a)/2^k$, $n = 2^k$. Hence the trapezoidal rule for 2^k subintervals (i.e. n is even) becomes

$$T_{1,k} = \frac{h_k}{2}(f_0 + 2f_1 + 2f_2 + \cdots + 2f_{2^k-1} + f_{2^k})$$

$$I = T_{1,k} + a_1h_k^2 + a_2h_k^4 + a_3h_k^6 + \cdots$$

We define

$$T_{2,k} = \frac{1}{3}(4T_{1,k+1} - T_{1,k}), \quad k = 1, 2, \cdots$$

which is the Simpson's rule for h_k and hence has an error $\mathcal{O}(h_k^4)$, i.e.,

$$I = T_{1,k} + c_1h_k^4 + c_2h_k^6 + \cdots$$

In general, we define

$$T_j^i = \frac{1}{4^j - 1}(4^jT_{j-1}^i - T_{j-1}^{i-1}), \quad j = 1, \cdots, m \quad i = 1, 2, \cdots n \tag{2.24}$$

We can represent the approximations in the triangular form:

h_i	T_0^i	T_1^i	T_2^i	$\dots$	T_m^i
h_1	T_0^1				
h_2	T_0^2	T_1^1			
h_3	T_0^3	T_1^2	T_2^1		
$\vdots$	$\vdots$	$\vdots$		$\ddots$	
h_m	T_0^m	T_1^{m-1}	$\dots$		T_m^1

2.3.0.1 Example

Use Romberg integration to find the integral of $f(x) = e^{-x}$ for $x \in [0,1]$. Take the initial sub-interval as $h = (1-0)/2 = 0.5$. Use 6 decimal places.

h_i	T_0^k	T_1^k	T_2^k
0.5	0.645235		
0.25	0.635409	0.632134	
0.125	0.632943	0.632121	0.632121

Hence $T_2^1 = 0.632121$ with an error of $\mathcal{O}(h^6)$.

2.3.1 Exercises

- Use (a) the trapezoidal rule (b) Simpson's rule to estimate I for the following:
  - (i) $f(x) = \dfrac{1}{1+x^2}$, over the interval $[0,1]$ for $n = 4$
  - (ii) $f(x) = xe^{-x^2}$ over the interval $[0,2]$ for $n = 4$ Compare your numerical results with the analytical ones.
- Use Romberg's method to approximate to integral

$$I = \int_0^1 \sqrt{1-x^2}\,dx$$

  Use $h_1 = 0.2$, $h_2 = 0.1$ and $h_3 = 0.05$.
- Estimate $\int_0^\pi f(x)\,dx$ as accurately as possible, where $f(x)$ is defined by the data:

x values	0	$\pi/4$	$\pi/2$	$3\pi/4$	π
$f(x)$	1.0000	0.3431	0.2500	0.3431	1.0000

- The period of a simple pendulum of length L is $\tau = 4\sqrt{\dfrac{L}{g}h(\theta_0)}$, where g is the gravitational acceleration, θ_0 represents the angular amplitude and:

$$h(\theta_0) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1-\sin^2(\theta_0/2)\sin^2\theta}}.$$

  Compute $h(15^\circ)$, $h(30^\circ)$ and $h(45^\circ)$.

2.4 Double and Triple Integrals

2.4.1 The Midpoint Method for Double Integrals

Given a double integral over the rectangular domain $[a,b] \times [c,d]$:

$$\int_a^b \int_c^d f(x,y)\,dy\,dx.$$

Can we approximate this integral numerically?

This can be done by considering the double integral as two integrals, each in one variable and then approximate each one numerically with our earlier formulae. Therefore:

$$\int_a^b \int_c^d f(x,y)\,dy\,dx = \int_a^b g(x)\,dx, \quad g(x) = \int_c^d f(x,y)\,dy.$$

We can now use the midpoint method and begin with $g(x) = \int_c^d f(x,y)\,dy$. For the interval $[c,d]$ we have n_y and length h_y. Thus the integral becomes:

$$g(x) = \int_c^d f(x,y)\,dy \approx h_y \sum_{j=0}^{n_y-1} f(x,y_j), \quad y_j = c + \frac{1}{2}h_y + jh_y.$$

This looks slightly different than before, since we need to integrate in both x and y directions. Note when integrating in the y direction we use n_y for n, h_y for h and index according to j. When integrating in the x direction, we use h_x, n_x and i respectively.

So, the double integral approximated by the midpoint method:

$$\int_a^b g(x)\,dx \approx h_x \sum_{i=0}^{n_x-1} g(x_i), \quad x_i = a + \frac{1}{2}h_x + ih_x.$$

So finally putting both approximations together we get the *composite midpoint method* for the double integral:

$$\begin{aligned}\int_a^b \int_c^d f(x,y)\,dy\,dx &\approx h_x \sum_{i=0}^{n_x-1} h_y \sum_{j=0}^{n_y-1} f(x_i,y_j) \\ &= h_x h_y \sum_{i=0}^{n_x-1} \sum_{j=0}^{n_y-1} f(a + \frac{h_x}{2} + ih_x, c + \frac{h_y}{2} + jh_y). \end{aligned} \tag{2.25}$$

2.4.1.1 Example

Compute the integral:

$$\int_2^3 \int_0^2 (2x+y)\,dy\,dx.$$

(Solution: 12 - check as an exercise)

```
f  = lambda x, y: 2*x + y
a  = 2; b  = 3; c  = 0; d  = 2; nx = 5; ny = 5
print('Numerical approximation is:', midpoint_method_double(f, a, b, c, d, nx, ny))
# check symbolic solution
```

```
## Numerical approximation is: 12.000000000000009
```

```
import sympy
x, y     = sympy.symbols('x y')
true_ans = sympy.integrate(f(x, y), (x, a, b), (y, c, d))
print('True analytical solution:', true_ans)
```

```
## True analytical solution: 12
```

2.4.2 The Midpoint Method for Triple Integrals

The idea used for double integrals can similarly be extended to three dimensions. Consider the triple integral:

$$\int_a^b \int_c^d \int_e^f g(x,y,z)dzdydx,$$

we wish to approximate the integral via the midpoint rule. Utilising the same strategy as before, we split the integral into one-dimensional integrals:

$$\begin{aligned} p(x,y) &= \int_e^f g(x,y,z)dz \\ q(x) &= \int_c^d p(x,y)dy \\ \int_a^b \int_c^d \int_e^f g(x,y,z)dzdydx &= \int_a^b q(x)dx \end{aligned}$$

Next we apply the midpoint rule to each of these one-dimension integrals:

$$\begin{aligned} p(x,y) = \int_e^f g(x,y,z)dz &\approx \sum_{k=0}^{n_z-1} g(x,y,z_k), \\ q(x) = \int_c^d p(x,y)dy &\approx \sum_{j=0}^{n_y-1} p(x,y_j), \\ \int_a^b \int_c^d \int_e^f g(x,y,z)dzdydx = \int_a^b q(x)dx &\approx \sum_{i=0}^{n_x-1} q(x_i), \end{aligned}$$

where:

$$z_k = e + \frac{1}{2}h_z + kh_z, \quad y_j = c + \frac{1}{2}h_y + jh_y \quad x_i = a + \frac{1}{2}h_x + ih_x.$$

So finally, starting with the formula for $\int_a^b \int_c^d \int_e^f g(x,y,z)dzdydx$ and combining the two previous formulas we have:

$$\begin{aligned} &\int_a^b \int_c^d \int_e^f g(x,y,z)\,dzdydx \approx \\ &h_x h_y h_z \sum_{i=0}^{n_x-1} \sum_{j=0}^{n_y-1} \sum_{k=0}^{n_z-1} g(a + \frac{1}{2}h_x + ih_x, c + \frac{1}{2}h_y + jh_y, e + \frac{1}{2}h_z + kh_z). \end{aligned} \tag{2.26}$$

2.4.2.1 Example

Evaluate the following integral:

$$\int_2^3 \int_1^2 \int_0^1 8xyz\,dzdydx,$$

where $n_x = n_y = n_z = 5$

```
f1 = lambda x, y, z: 8*x*y*z
a = 2; b = 3; c = 1; d = 2; e = 0; f = 1; nx = 5; ny = 5; nz = 5
print(midpoint_method_triple(f1, a, b, c, d, e, f, nx, ny, nz))
# Check symbolic solution
```

```
## 15.000000000000009
```

```
import sympy
x, y, z     = sympy.symbols('x y z')
true_ans    = sympy.integrate(f1(x, y, z), (x, a, b), (y, c, d), (z, e, f))
print('True analytical answer:', true_ans)
```

```
## True analytical answer: 15
```

Chapter 3

Numerical Solutions to Nonlinear Equations

Non-linear equations occur in many world problems and are rarely solvable analytically.

It is of great importance to solve equations of the form

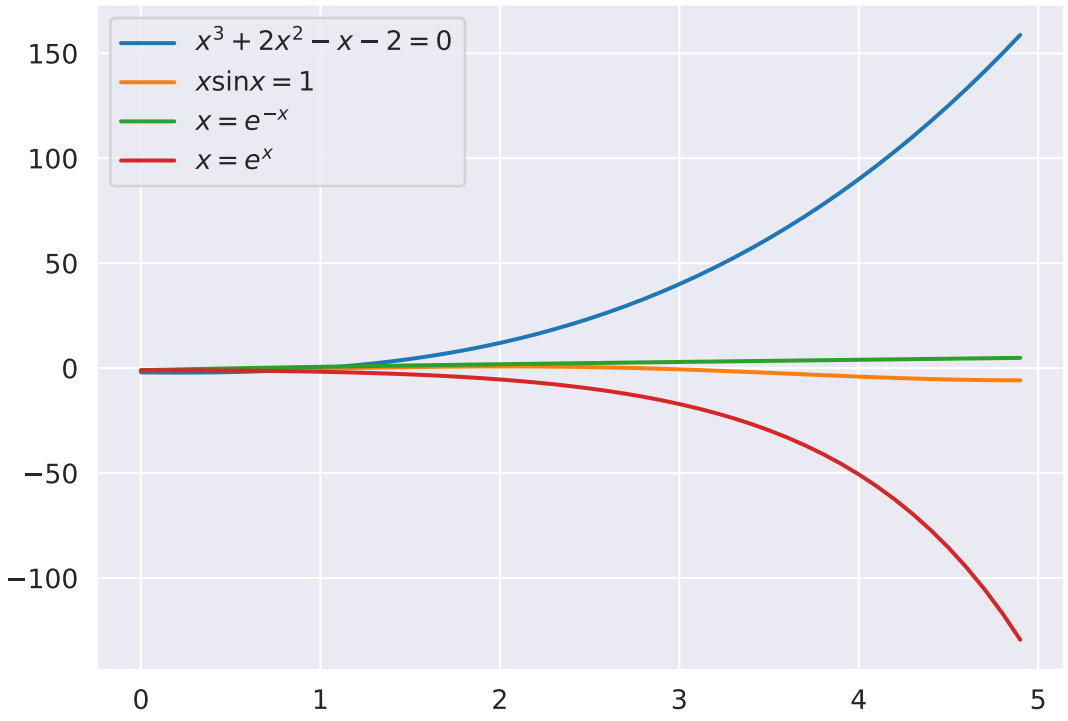
$$f(x) = 0$$

in many applications in science and engineering. The values of x that make $f(x) = 0$ are called the roots (or the zeros) of this equation.

This type of problem also includes determining the points of intersection of curves. If $f(x)$ and $g(x)$ represent equations of two curves, the intersection points correspond to the roots of the function $F(x) = f(x) - g(x) = 0$.

Examples of nonlinear equations:

- $ax^2 + bx + c = 0$ (two roots).
- $x^3 + 2x^2 - x - 2 = 0$ (three roots)
- $x \sin x = 1$ (infinitely many roots).
- $x = e^{-x}$ (one root)
- $x = e^{x}$ (No roots)



3.1 Nonlinear equations in one unknown: $f(x) = 0$

We shall examine two types of iterative methods for determining the roots of the equation $f(x) = 0$, namely:

- **Bracketing methods,** also known as interval methods.
- **Fixed point methods**

To obtain these intervals or initial approximations graphical methods are usually used.

3.1.1 Interval Methods

These methods require an initial interval which is guaranteed to contain a root. The width of this interval (bracket) is reduced iteratively until it encloses the root to a desired accuracy.

3.1.2 Bisection Method

The bisection method is an incremental search method in which the interval is always divided in half.

Intermediate value theorem:

If $f(x)$ is real and continuous in an interval $[a,b]$ and $f(a)f(b) < 0$, then there exists a point $c \in (a,b)$ such that $f(c) = 0$.

If we calculate the midpoint of $[a, b]$ i.e.,

$$c = \frac{1}{2}(a + b)$$

then:

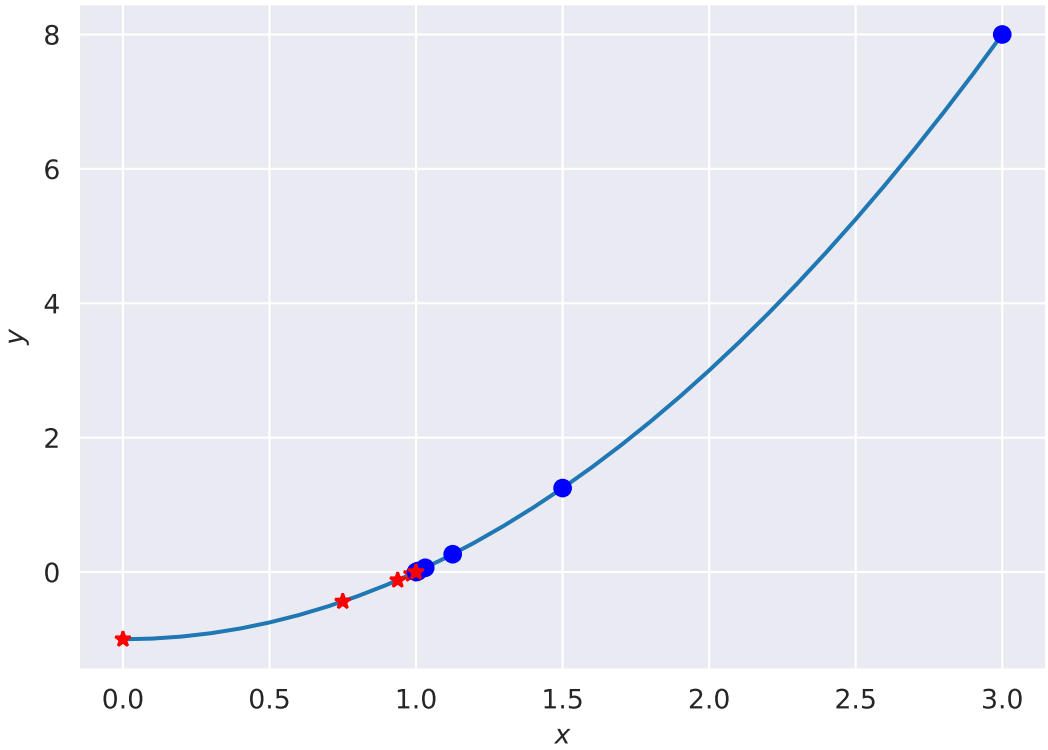
- If $f(a)f(c) < 0$ then $f(a)$ and $f(c)$ have opposite signs and so the root must lie in the smaller interval $[a, c]$.
- If $f(a)f(c) > 0$ then $f(a)$ and $f(c)$ have the same signs and so $f(b)$ and $f(c)$ must have opposite signs, so the root lies in $[c, b]$.

3.1.2.1 Example

Perform two iterations of the bisection method on the function $f(x) = x^2 - 1$, using $[0, 3]$ as your initial interval.

Answer: The root lies at 1, but after two iterations, the interval will be $[0.75, 1.5]$.

```
f   = lambda x: x**2 - 1
x   = np.arange(-1, 3, 0.1)
y   = f(x)
a   = 0
b   = 3
tol = 10**-3
val = bisection_plot(f, a, b, tol)
```



Stopping Criteria:

We use a stopping criteria of

$$|b_n - a_n| < \epsilon$$

We have

$$\begin{aligned} |b_1 - a_1| &= |b - a| \\ |b_2 - a_2| &= \frac{1}{2}|b_1 - a_1| \\ &\vdots \\ |b_n - a_n| &= \frac{1}{2}|b_{n-1} - a_{n-1}| \\ &= \frac{1}{2^2}|b_{n-2} - a_{n-2}| \\ &= \frac{1}{2^{n-1}}|b_1 - a_1| \end{aligned}$$

We require that $|b_n - a_n| \approx \epsilon$ which implies

$$\frac{1}{2^{n-1}}|b_1 - a_1| \approx \epsilon, \qquad \text{or} \quad 2^n = 2\frac{|b_1 - a_1|}{\epsilon}$$

or

$$n = \log\left(2\frac{|b_1 - a_1|}{\epsilon}\right) / \log 2 \tag{3.1}$$

3.1.2.2 Example

Find the root of $f(x) = \sin(x) - 0.5$ between 0 and 1. Iterate until the interval is of length $\frac{1}{2^3}$

Answer: the final interval is $[0.5, 0.625]$. $f(0.5) = -0.0206$

Definition 3.1 (Bisection Method Theorem)**.** If the bisection algorithm is applied to a continuous function f on an interval $[a, b]$, where $f(a)f(b) < 0$, then, after n steps, an approximate root will have been computed with error at most $(b - a)/2^{n+1}$.

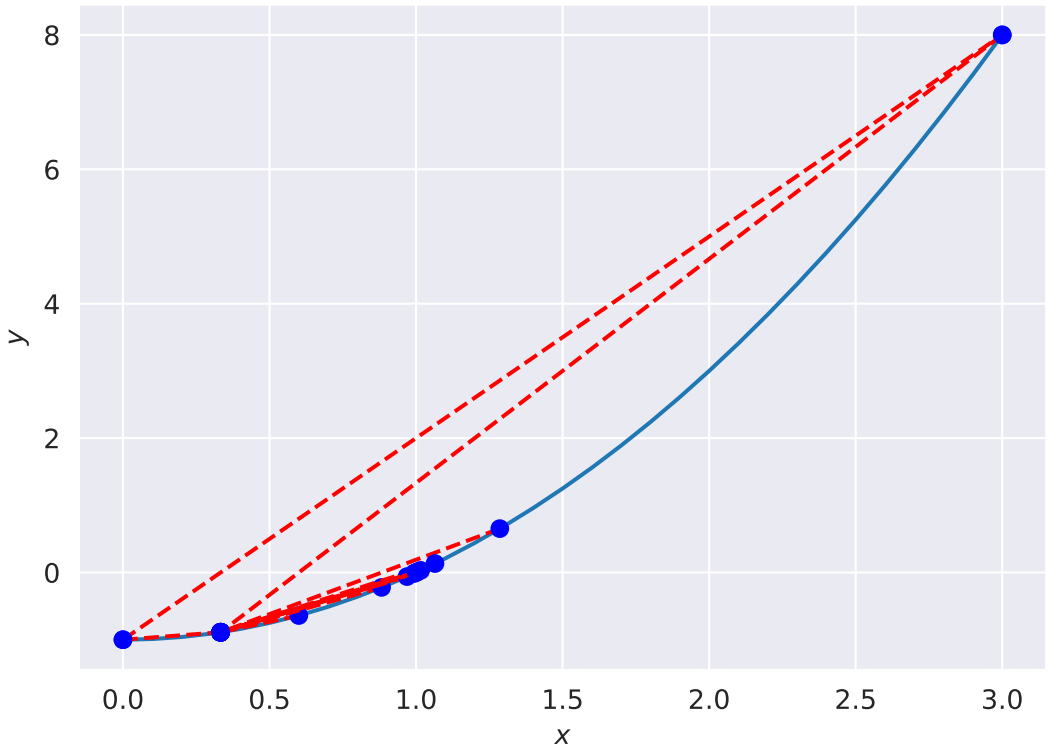
- Bisection will always converge to a root if the function is continuous.
- Reliable but slow. The method does not exploit any knowledge about the function in question.
- Convergence rate is linear. It gains the same amount of accuracy from iteration to iteration.

3.1.3 False position method or Regula Falsi

The bisection method is attractive because of its simplicity and guaranteed convergence. Its disadvantage is that it is, in general, extremely slow.

Regula Falsi algorithm is a method of finding roots based on linear interpolation. Its convergence is linear, but it is usually faster than bisection. On each iteration a line is drawn between the endpoints $(a, f(a))$ and $(b, f(b))$ and the point where this line crosses the $x-$axis taken as the point c.

```
f   = lambda x: x**2 - 1
x   = np.arange(-1, 3, 0.1)
y   = f(x)
a   = 0
b   = 3
tol = 10**-3
val = false_position_plot(f, a, b, tol)
```



The equation of the line through $(a, f(a))$ and $(b, f(b))$ is

$$y = f(a) + \frac{x-a}{b-a}(f(b) - f(a)).$$

We require the point c where $y = 0$, i.e.

$$f(c) = f(a) + \frac{c-a}{b-a}(f(b) - f(a)) = 0,$$

from which we solve for c to get:

$$c = \frac{af(b) - bf(a)}{f(b) - f(a)} \tag{3.2}$$

The sign of $f(c)$ determines which side of the interval does not contain the root, which side is discarded to give a new, smaller interval containing the root. The procedure is continued until the interval is sufficiently small.

3.1.3.1 Example

Perform two iterations of the false position method on the function $f(x) = x^2 - 1$, using $[0, 3]$ as your initial interval. Compare your answers to those of the bisection method.

Answer: False position, in other words, performs a linear fit onto the function, and then directly solves that fit.

With Bisection we obtain the following,

a	c	b
0	1.5	1.5

a	c	b
0.75	0.75	1.5
0.75	1.125	1.125
0.9375	0.9375	1.125
0.9375	1.03125	1.03125
0.984375	0.984375	1.03125

Stopping criteria

The false position method often approaches the root from one side only, so we require a different stopping criteria from that of the bisection method. We usually choose:

$$|c - c^*| < \epsilon$$

where c^* is the value of c calculated from the previous step.

- Normally faster than Bisection Method. Can decrease the interval by more than half at each iteration.
- Superlinear convergence rate. Linear convergence rate in the worst case.
- Usually approaches the root from one side.

3.1.3.2 Exercise

Use the bisection method and the false position method to find the root of $f(x) = x^2 - x - 2$ that lies in the interval $[1, 4]$.

3.1.4 Fixed Point Methods

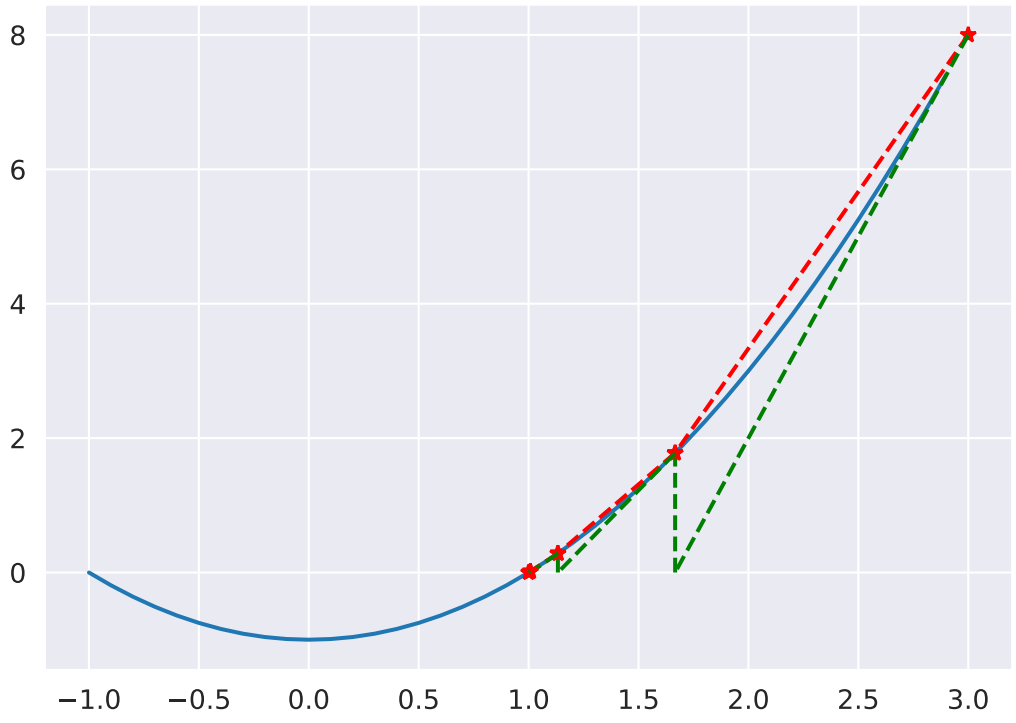
For these methods we start with an initial approximation to the root and produce a sequence of approximations, each closer to the root than its predecessor.

3.1.5 Newton's Method

This is one of the most widely used of all root-finding formulae. It works by taking as the new approximation the point of intersection of the tangent to the curve $y = f(x)$ at x_i with the x–axis. Thus we seek to solve the equation $f(x) = 0$, where f is assumed to have a continuous derivative f'.

> Newton developed this method for solving equations while wanting the find the root to the equation $x^3 - 2x - 5 = 0$. although he demonstrated the method only for polynomials, it is clear he realised its broader applications.

```
plt.show()
```



Newton's method can be derived in several ways; we choose to do it using Taylor series.

Let $x_{i+1} = x_i + h$ and obtain a Taylor's expansion of $f(x_{i+1})$ about x_i,

$$f(x_{i+1}) = f(x_i) + hf'(x_i) + \frac{h^2}{2}f''(x_i) + \cdots \tag{3.3}$$

An approximation is obtained by truncating the Taylor series after two terms:

$$f(x_{i+1}) \approx f(x_i) + hf'(x_i)$$

Thus this series has an error $O(h^2)$.

Ideally $f(x_{i+1}) = 0$ so that solving for h gives

$$h = -\frac{f(x_i)}{f'(x_i)}, \qquad \text{provided} \qquad f'(x_i) \neq 0.$$

Therefore

$$x_{i+1} = x_i + h = x_i - \frac{f(x_i)}{f'(x_i)}, \quad i = 0, 1, 2, \cdots \tag{3.4}$$

which is called Newton's (or Newton-Raphson's) iterative formula.

- Requires the derivative of the function.
- Has quadratic convergence rate. Linear in worst case.
- May not converge if too far from the root.
- Could get caught in basins of attraction with certain sinusoidals .

3.2 Newton's Method for Systems of Nonlinear Equations

Newton's method may also be used to find roots of a system of two or more non-linear equations.

Consider a system of two equations:

$$f(x,y)=0, \qquad g(x,y)=0, \tag{3.5}$$

Using Taylor's expansion of the two functions near (x,y) we have

$$f(x+h,y+k) = f(x,y)+h\frac{\partial f}{\partial x}+k\frac{\partial f}{\partial y}+\text{terms in } h^2,\ k^2,\ hk \tag{3.6}$$

$$g(x+h,y+k) = g(x,y)+h\frac{\partial g}{\partial x}+k\frac{\partial g}{\partial y}+\text{terms in } h^2,\ k^2,\ hk \tag{3.7}$$

and if we keep only the first order terms, we are looking for a couple (h,k) such that:

$$f(x+h,y+k) = 0 \approx f(x,y)+h\frac{\partial f}{\partial x}+k\frac{\partial f}{\partial y} \tag{3.8}$$

$$g(x+h,y+k) = 0 \approx g(x,y)+h\frac{\partial g}{\partial x}+k\frac{\partial g}{\partial y} \tag{3.9}$$

hence it is equivalent to the linear system:

$$\begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix} \begin{bmatrix} h \\ k \end{bmatrix} = -\begin{bmatrix} f(x,y) \\ g(x,y) \end{bmatrix} \tag{3.10}$$

The 2×2 matrix is called the Jacobian matrix (or Jacobian) and is sometimes denoted as:

$$J(x,y) = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{bmatrix}$$

The couple (h,k) is thus

$$\begin{bmatrix} h \\ k \end{bmatrix} = -J^{-1}(x,y)\begin{bmatrix} f(x,y) \\ g(x,y) \end{bmatrix}$$

The general Jacobian of a a $(n\times n)$ matrix for a system of n equations and n variables, $(x_1,x_2,\ldots,x_n)$ is immediate:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2},\cdots\frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2}\cdots\frac{\partial f_2}{\partial x_n} \\ \vdots & \ddots\ \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2}\cdots\frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

If we define $x_{i+1}=x_i+h$ and $y_{i+1}=y_i+k$ then the equation(3.10) suggests the iteration formula:

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \end{bmatrix} - J^{-1}(x_n,y_n)\begin{bmatrix} f(x_i,y_i) \\ g(x_i,y_i) \end{bmatrix}$$

Starting with an initial guess (x_0,y_0) and under certain conditions it's possible to show that this iteration process converges to a root of the system.

3.2.0.1 Exercise

Use Newton's method to look for a root near $x_0 = -0.6, \quad y_0 = 0.6$.

$$\begin{aligned} f(x,y) &= x^3 - 3xy^2 - 1 \\ g(x,y) &= 3x^2y - y^3 \end{aligned}$$

Answer: $x^* = [-0.5,\ 0.8660]$

```
F   = lambda x: np.array([x[0]**3 - 3*x[0]*x[1]**2 - 1,
                          3*x[0]**2*x[1] - x[1]**3])
J   = lambda x: np.array([[3*x[0]**2 - 3*x[1]**2, -6*x[0]*x[1]],
                          [6*x[0]*x[1], 3*x[0]**2 - 3*x[1]**2]])
x0     = np.array([-.6, .6])
tol    = 1e-6
o1, o2, o3 = newton_system(F, J, x0, tol)
print('The final solution of x is:')
```

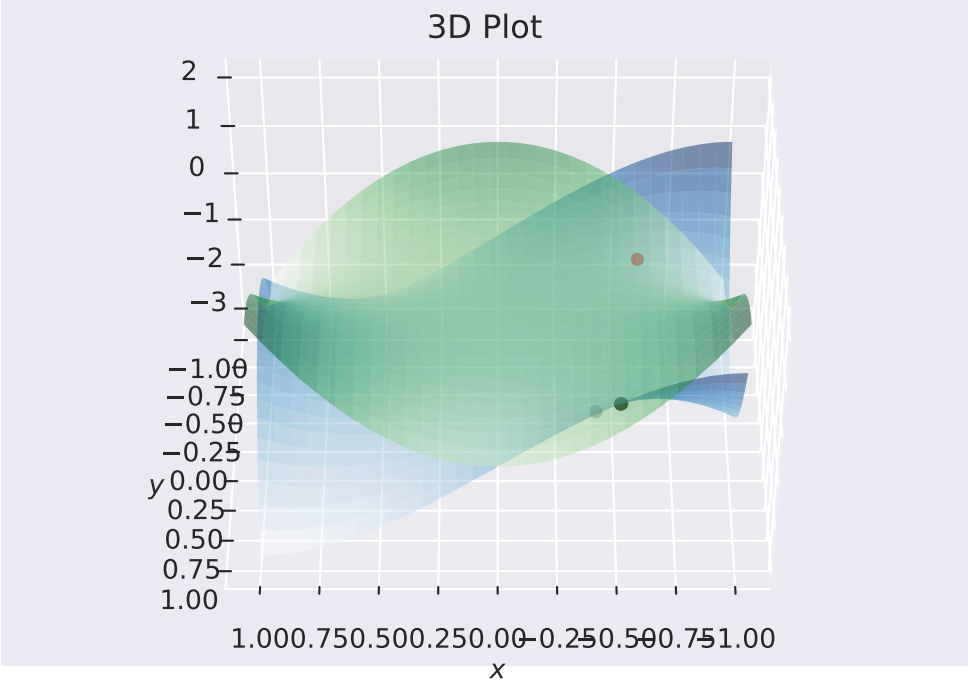
```
## The final solution of x is:
```

```
print(o1)
```

```
## [-0.5         0.86602539]
```

```
print('It took', o2, 'iterations')
```

```
## It took 4 iterations
```



3.2.1 Exercises

1. Show that the equation $x = \cos x$ has a solution in the interval $[0, \pi/2]$. Use the bisection method to reduce the interval containing the solution to a length of 0.2.
2. Use the bisection method to solve
$$e^{-x} = \ln x, \quad a = 1, \quad b = 2$$
3. Apply (i) the bisection method (ii) False Position and (iii) Newton's method to solve each of the following equations to, at least, 6D.
$$(a)\ x^2 = e^{-x} \qquad (b)\ 2x = \tan x, \text{ near} \qquad x = 1$$
4. Make one Newton iteration for each of the following systems:
$$(a)\ xy^2 = ye^x, \qquad x\cos y - 1 = e^{-y}, \qquad \text{near} \qquad (0,1)$$
$$(b)\ f_1(x,y) = x^2 - 2y^2 - 1, \qquad f_2(x,y) = x^3y^2 - 2, \qquad \text{near} \quad (1.5,1)$$
5. Briefly explain how bracketing algorithms work to find zeros of one dimensional functions and describe two variations used in practice.
6. Is Newton's Method guaranteed to find the zero of any continuous function that has a zero and for any starting point?
7. Given an initial bracket of $[0,100]$, how many steps of Bisection Method are required to reduce the bracket size below 10^{-15}?
8. Explain the meaning of the phrase: *A convergent numerical method is qualitatively just as good as an analytical solution*
9. Motivate the False-Position Method, why is it generally preferable to the Bisection Method?
10. Use Newton's method to find a solution to the following system:
$$v - u^3 = 0, \tag{3.11}$$
$$u^2 + v^2 - 1 = 0, \tag{3.12}$$
given a starting value of $(1,2)$. Plot the curves along with successive approximations to determine if it is indeed true that the approximations approach the intercept.

Chapter 4

Eigenvalues and Eigenvectors

Let $\mathbf{A}$ be an $n \times n$ matrix. If we can find a scalar λ and a nonzero vector x satisfying

$$\mathbf{A}x = \lambda x \tag{4.1}$$

then λ and x are called an **eigenpair** of $\mathbf{A}$, λ being the eigenvalue (or a characteristic value) and x the corresponding eigenvector. Equation (4.1) can be rewritten as:

$$(\mathbf{A} - \lambda\mathbf{I})x = 0 \ \ (0 \text{ null column vector}) \tag{4.2}$$

The set of homogeneous equation (4.2) admits a non–trivial solution if and only if

$$\det(\mathbf{A} - \lambda\mathbf{I}) = |\mathbf{A} - \lambda\mathbf{I}| = 0.$$

The determinant $|\mathbf{A} - \lambda\mathbf{I}|$ is an n^{th} degree polynomial in λ and is called the **characteristic polynomial** of $\mathbf{A}$. Thus one way to find the eigenvalues of $\mathbf{A}$ is to obtain its characteristic polynomial and then find the n zeros of this polynomial.

Although the characteristic polynomial is easy to work, for large values of n finding the roots of the polynomial equations is difficult and time consuming.

If we are interested in the eigenvalue of largest magnitude, then the **power method** becomes a popular approach.

4.1 The Power Method

The power method, like the Jacobi and the Gauss-Seidel, is an iterative method for approximating the dominant eigenvalue and its associated eigenvector. The main motivation behind the power method is that multiplication by a matrix tends to move vectors towards the dominant eigenvector direction. With slight modifications it can be used to determine the smallest eigenvalue and the intermediate values.

Note: The power method works only for $n \times n$ matrices with real eigenvalues.

Let $\mathbf{A}$ be an $m \times m$ matrix and let $\lambda_1, \lambda_2, \cdots, \lambda_m$ be the eigenvalues of $\mathbf{A}$ and $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_m$ the associated eigenvectors.

λ_1 is called the **dominant** (largest in magnitude) eigenvalue of $\mathbf{A}$ if:

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_m|.$$

The eigenvector corresponding to λ_1 is called the **dominant eigenvector** of $\mathbf{A}$.

Let $\mathbf{A}$ have the eigenvalues $2, 5, 0, -7, -2$. Does $\mathbf{A}$ have a dominant eigenvalue? If so, what is it?

Solution: Since $|-7| > |5| > |2| \geq |-2| > 0$ **A** has $\lambda_1 = -7$ as its dominant eigenvalue.

Not every matrix has a dominant eigenvalue. For instance the matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

has eigenvalues $\lambda_1 = 1$ and $\lambda_2 = -1$ and therefore has no dominant eigenvalue.

The eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_n$ form a basis of $\mathbb{R}^n$ so that any vector $\mathbf{x}$ can be written as a linear combination of them:

$$\mathbf{x} = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_n\mathbf{v}_n.$$

Derive the following equations:

$$\begin{aligned} \mathbf{Ax} &= c_1\mathbf{Av}_1 + c_2\mathbf{Av}_2 + \cdots + c_n\mathbf{Av}_n \\ &= c_1\lambda_1\mathbf{v}_1 + c_2\lambda_2\mathbf{v}_2 + \cdots + c_n\lambda_n\mathbf{v}_n \\ \mathbf{A}^2\mathbf{x} &= c_1\lambda_1\mathbf{Av}_1 + c_2\lambda_2\mathbf{Av}_2 + \cdots + c_n\lambda_n\mathbf{Av}_n \\ &= c_1\lambda_1^2\mathbf{v}_1 + c_2\lambda_2^2\mathbf{v}_2 + \cdots + c_n\lambda_n^2\mathbf{v}_n \end{aligned}$$

It can be seen that:

$$\mathbf{A}^m\mathbf{x} = c_1\lambda_1^m\mathbf{v}_1 + c_2\lambda_2^m\mathbf{v}_2 + \cdots + c_n\lambda_n^m\mathbf{v}_n$$

If λ_1 is the largest eigenvalue in magnitude and m is sufficiently large, then we have:

$$\begin{aligned} \mathbf{A}^m\mathbf{x} &= \lambda_1^m(c_1\mathbf{v}_1 + c_2\left(\frac{\lambda_2}{\lambda_1}\right)^m\mathbf{v}_2 + \cdots + c_n\left(\frac{\lambda_n}{\lambda_1}\right)^m\mathbf{v}_n \\ &= c_1\lambda_1^m\mathbf{v}_1 + \text{small corrections} \end{aligned} \tag{4.3}$$

When m is sufficiently large, we have:

$$\lambda_1 = \lim_{m\to\infty}(\frac{\mathbf{A}^{m+1}\mathbf{x})_r}{(\mathbf{A}^m\mathbf{x})_r}$$

where r denotes the r-th element of a vector.

One may wish to ask, how to we find approximate eigenvalues? Well, assuming that a matrix A and an approximate eigenvector are known, then what is the best guess for the associated eigenvalue? We can use least squares. Given the eigenvalue equation $x\lambda = Ax$, where x is an approximate eigenvector and λ is unknown. Then the normal equations say that the least squares answer is the solution of $x^T x\lambda = x^T A\lambda$, which rewritten gives us:

$$\lambda = \frac{x^T Ax}{x^T x}, \tag{4.4}$$

known as the **Rayleigh Quotient**. Thus, given an *approximate eigenvector*, the Rayleigh Quotient will be the best approximate *eigenvalue*. Applying this to the normalised eigenvector adds an eigenvalue approximation to the Power Method.

Therefore, the Power Method algorithm may be implemented via:

Input: an initial vector x_0
for $j = 1, 2, ..., n$ do
$u_{j-1} = x_{j-1}/||x_{j-1}||_2$
$x_j = A \times u_{j-1}$
$\lambda_j = u_{j-1}^T A u_{j-1}$
end
$u_j = x_j/||x_j||_2$

4.1.0.1 Example

Using the power method, find the dominate eigenvalue and vector of the following matrix:

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 2 \end{bmatrix}.$$

Use a starting value of $x_0 = [-5, 5]$.

(check this by hand)

```
A  = np.array([[1, 3], [2, 2]])
x0 = np.array([-5, 5])
n  = 15
print(power_method(A, x0, n))
```

```
## Current lambda value: -0.9999999999999998
## Current lambda value: 1.0
## Current lambda value: 3.8
## Current lambda value: 3.894117647058823
## Current lambda value: 4.017678708685626
## Current lambda value: 3.995003093323181
## Current lambda value: 4.001213545191886
## Current lambda value: 3.999694377268076
## Current lambda value: 4.000076266004321
## Current lambda value: 3.9999809247674634
## Current lambda value: 4.000004768262443
## Current lambda value: 3.9999988079002833
## Current lambda value: 4.0000002980227976
## Current lambda value: 3.9999999254941665
## Current lambda value: 4.00000001862645
## (array([1.        , 0.99999999]), 4.00000001862645)
```

```
from sympy import *
A = Matrix([[1, 3], [2, 2]])
print(A.eigenvals())
```

```
## {4: 1, -1: 1}
```

Remark: If $\mathbf{x}$ is an eigenvector corresponding to some given eigenvalue, then so is $k\mathbf{x}$ for any k. Thus, only the direction of a vector matters. We can choose its length by changing k. Therefore we will seek eigenvectors of length unity, called the normalised eigenvectors.

The previous pseudocode for power method above can be accomplished by hand with the following:

Step 1:

Let $\mathbf{x}_0$ be an initial guess of the eigenvector corresponding to λ_1. Normalise it. Call the result $\mathbf{y}_0$

$$\mathbf{y}_0 = \frac{\mathbf{x}_0}{\|\mathbf{x}_0\|}$$

Step 2:

Multiply $\mathbf{y}_0$ once by $\mathbf{A}$ to get a new vector. Normalise the result and call it $\mathbf{y}_1$

$$\mathbf{x}_1 = \mathbf{A}\mathbf{y}_0, \quad \mathbf{y}_1 = \frac{\mathbf{x}_1}{\|\mathbf{x}_1\|}$$

Step 3:

Multiply $\mathbf{y}_1$ once by $\mathbf{A}$ to get a new vector. Normalise the result and call it $\mathbf{y}_2$

$$\mathbf{x}_2 = \mathbf{A}\mathbf{y}_1, \quad \mathbf{y}_2 = \frac{\mathbf{x}_2}{\|\mathbf{x}_2\|}$$

We have the iteration formula:

$$\mathbf{x}_i = \mathbf{A}\mathbf{y}_{i-1}, \quad \mathbf{y}_i = \frac{\mathbf{x}_i}{\|\mathbf{x}_i\|}$$

Repeat m times. If m is sufficiently large enough, $\mathbf{y}_{m-1}$ should be approximately equal to $\mathbf{y}_m$ then we stop. Thus $\mathbf{y}_m$ is approximately the normalised eigenvectors of $\mathbf{A}$ corresponding to the dominant eigenvalue λ_1 and hence:

$$\mathbf{A}\mathbf{y}_{m-1} \approx \mathbf{A}\mathbf{y}_m = \lambda_1 \mathbf{y}_m,$$

which we can use to read off the eigenvalue.

The power method will converge if the **initial estimate** of the eigenvector **has a nonzero component** in the direction of the eigenvector corresponding to the dominant eigenvalue. For this reason, starting vector with **all components equal to 1** is usually used in the computations of the power method.

Exercise:

Use the power method to estimate the dominant eigenpair of the matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 2 \end{bmatrix}$$

to two digits of accuracy. Use the initial guess $\mathbf{x}_0 = (1,1)^T$.

4.2 The Inverse Power Method

To find the smallest eigenpair we can use the **Inverse Power Method**. If the Power Method is applied to the inverse of the matrix, then the smallest eigenvalue can be found.

We should try to avoid computing the inverse of A as much as possible, so we may rewrite the application of the Power Method to A^{-1} from:

$$x_{j+1} = A^{-1}x_j,$$

to:

$$Ax_{j+1} = x_j,$$

and solve for x_{j+1} using Gaussian Elimination.

You will cover many further methods to compute eigenvalues and eigenvectors in the third year numerical methods course. Therefore, we will only consider the above two for this course.

4.2.1 Exercises

1. Use the power method to calculate approximations to the dominant eigenpair (if a dominant eigenpair exists). Perform 5 iterations in each case. Use $\mathbf{x}_0 = [1\ 1]^T$. Check your answers analytically and with Python
   1. $\begin{pmatrix} 1 & 5 \\ 5 & 6 \end{pmatrix}$
   2. $\begin{pmatrix} 2 & 3 \\ -2 & 1 \end{pmatrix}$
2. Use the inverse power method for find the smallest eigenpair on previous two questions.
3. Find the characteristic polynomial and the eigenvalues and eigenvectors of the following matrics:
   - $$\begin{bmatrix} 3.5 & -1.5 \\ -1.5 & 3.5 \end{bmatrix}$$
   - $$\begin{bmatrix} 3.5 & -1.5 \\ -1.5 & 3.5 \end{bmatrix}$$

Chapter 5

Interpolation

Typically, from experimental observations or statistical measurements we may have the value of a function f at a set of points $x_0, x_1, \cdots, x_n$ $(x_0 < x_1 < \cdots < x_n)$. However, we do not have an analytic expression for f which would allow us to calculate the value of f at an arbitrary point.

You will frequently have occasion to estimate intermediate values between precise data points when dealing with real world data sets. The most common method used for this purpose is polynomial interpolation.

Polynomial functions which fit the known data are commonly used to allow us to approximate these arbitrary points. If we use this function to approximate f for some point $x_0 < x < x_n$ then the process is called **interpolation**. If we use it to approximate f for $x < x_0$ or $x > x_n$ then it is called **extrapolation**.

Polynomials are used because:

- Computers can handle them easily. Which makes for fast and efficient programming.
- The integration and differentiation of polynomials is straightforward computationally.
- Polynomials are smooth functions - i.e. not only is a polynomial a continuous function, but all the derivatives exist and are themselves continuous.
- Polynomials are **uniformly approximate continuous** functions. This means that, given any function, which is continuous on some interval $[a, b]$ and any positive number ϵ (no matter how small) we can find a polynomial P such that

$$|f(x) - P(x)| < \epsilon, \quad x \in [a, b]$$

This result is known as Weierstrass Approximation theorem.

For $n+1$ data points, there is one and only one polynomial of order n that passes through all the points. For example, there is only one straight line (that is, a first-order polynomial) that connects two points. Similarly, only one parabola connects a set of three points. Polynomial interpolation consists of determining the unique nth-order polynomial that fits $n+1$ data points. This polynomial then provides a formula to compute intermediate values.

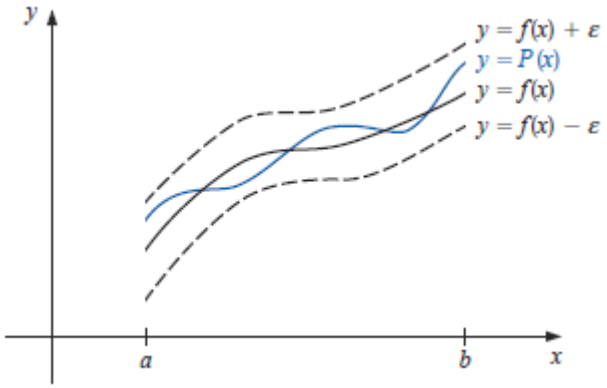
5.1 Weierstrauss Approximation Theorem

One of the most useful and well-known classes of functions mapping the set of real numbers into itself is the *algebraic polynomials*, the set of functions of the form,

$$P_n(x) = a_n x^n + a_{n-1}^{n-1} + ... + a_1 x + a_0,$$

where n is a nonnegative integer and $a_0, ..., a_n$ are real constants. One reason for their importance is that they uniformly approximate continuous functions. By this we mean that given any function, defined and continuous

on a closed and bounded interval, there exists a polynomial that is as "close" to the given function as desired. This result is expressed precisely in the Weierstrass Approximation Theorem.



Theorem 5.1 (Weierstrauss Approximation Theorem)**.** *Suppose that f is defined and continous on $[a,b]$. For each, $\epsilon > 0$, there exists a polynomial $P(x)$, with the property that,*

$$|f(x) - P(x)| < \epsilon, \qquad \text{for all } x \text{ in } \quad [a,b].$$

Note: Karl Weierstrass (1815-1897) is often referred to as the father of modern analysis because of his insistence on rigour in the demonstration of mathematical results. He was instrumental in developing tests for convergence of series, and determining ways to rigorously define irrational numbers. He was the first to demonstrate that a function could be everywhere continuous but nowhere differentiable, a result that shocked some of his contemporaries.

5.2 Linear Interpolation

Given only two points $(x_0, f(x_0))$ and $(x_1, f(x_1))$ $(y = f(x))$ the obvious interpolating function is the (unique) straight line that passes through them.

Let $P_1(x) = a_0 + a_1 x = f(x)$. Since this polynomial has to pass through these two points, it is required that:

$$a_0 + a_1 x_0 = f(x_0) \tag{5.1}$$
$$a_0 + a_1 x_1 = f(x_1) \tag{5.2}$$

By solving for a_0 and a_1, it is easy to show that:

$$a_0 = \frac{f(x_0)x_1 - f(x_1)x_0}{x_1 - x_0}, \quad a_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

and hence:

$$P_1(x) = \frac{f(x_0)x_1 - f(x_1)x_0}{x_1 - x_0} + x\frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

which can be rearranged to yield:

$$P_1(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$$

which is a linear interpolating formula.

Hence at $x = x^*$ the linear interpolate is :

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

Note that the quotient $\frac{f(x_1)-f(x_0)}{x_1-x_0}$ is the slope of the line joining $(x_0, f(x_0))$ and $(x_1, f(x_1))$. It is also a **finite divided difference approximation** to the first derivative.

5.2.0.1 Example

Estimate $\ln(2)$ using linear interpolation given $x_0 = 1$ and $x_1 = 6$.

Solution:

$$P_{(2)} = \ln 1 + \frac{\ln 6 - \ln 1}{6-1}(2-1) = 0.3583519$$

Calculator value $\ln 2 = 0.6931472$.

In this case the error is large because for one the interval between the data points is large and secondly we are linearly approximating a non-linear function.

5.3 Quadratic Interpolation

The error in the above example results because we approximated a curve with a straight line. We can improve the estimate by introducing some curvature into the line connecting the data points.

Given three distinct points $(x_i, f(x_i))$, $i = 0, 1, 2$, a unique **parabola** (i.e., a second degree polynomial) can be fitted through them:

$$P_2(x) = b_0 + b_1 x + b_2 x^2, \tag{5.3}$$

by finding suitable coefficients b_0, b_1 and b_2. A particularly convenient form for representing this polynomial is:

$$P_2(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) \tag{5.4}$$

Note: This polynomial is just equivalent to the general polynomial (5.3). This can be shown by multiplying out the terms in (5.4).

$$P_2(x) = (a_0 - a_1 x_0 + a_2 x_0 x_1) + (a_1 - a_2 x_0 - a_2 x_1)x + a_2 x^2$$

and hence:

$$b_0 = a_0 - a_1 x_0 + a_2 x_0 x_1 \quad b_1 = a_1 - a_2 x_0 - a_2 x_1 \quad b_2 = a_2$$

Thus equations (5.3) and (5.4) are equivalent formulations of the unique second degree polynomial joining three points.

Determination of the coefficients a_0, a_1 and a_2: The polynomial has to pass through the three points. Substituting in $x = x_0$ and $x = x_1$ gives:

$$P_2(x_0) = a_0 = f(x_0) \tag{5.5}$$

$$P_2(x_1) = f(x_0) + a_1(x_1 - x_0) = f(x_1), \quad \Rightarrow a_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \tag{5.6}$$

Finally, substituting in $x = x_2$ in (5.4) and making use of the evaluated values of a_0 and a_1, we can show, after some algebraic manipulations that:

$$a_2 = \frac{\frac{f(x_2)-f(x_1)}{x_2-x_1} - \frac{f(x_1)-f(x_0)}{x_1-x_0}}{x_2 - x_0}$$

Note: that a_1 still represents the slope of the line joining $(x_0, f(x_0))$ and $(x_1, f(x_1))$. The last term $a_2(x - x_0)(x - x_1)$ introduces the second order curvature into the formula.

5.3.0.1 Example

Fit a second degree polynomial that goes through the points $x_0 = 1$, $x_1 = 4$ and $x_2 = 6$ for $f(x) = \ln x$. Use this polynomial to approximate $\ln 2$.

Solution:

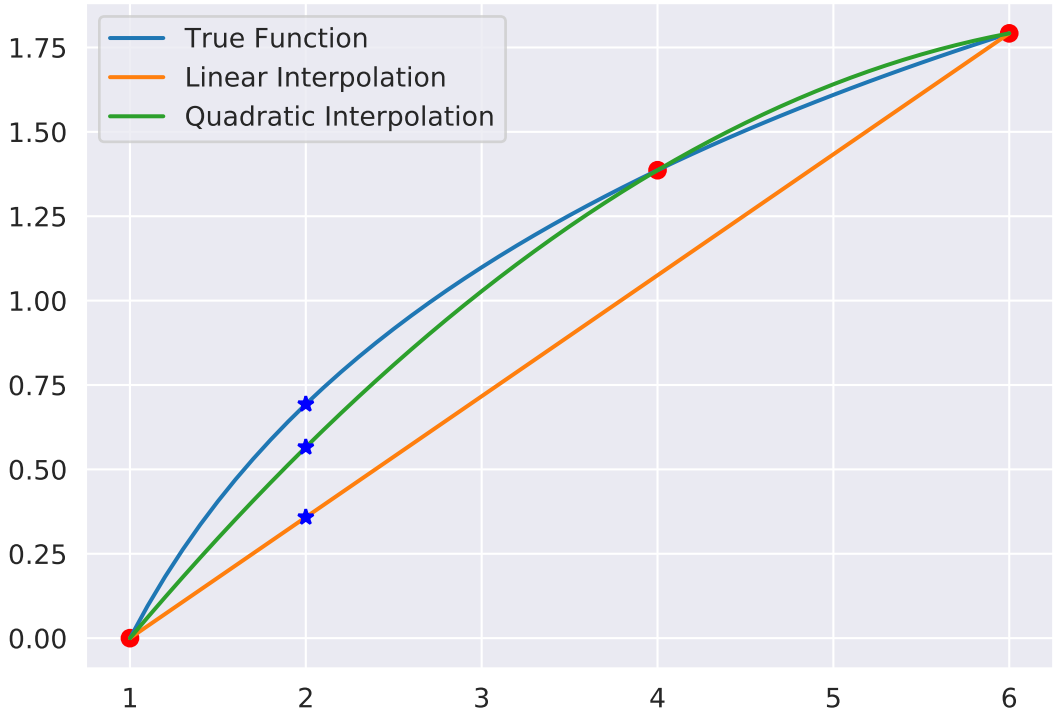
Polynomial,

$$P_2(x) = 0 + 0.46209813(x-1) - 0.051873116(x-1)(x-4)$$

Estimate for $\ln 2$, put $x = 2$ in $P_2(x)$

$$P_2(2) = 0 + 0.46209813(2-1) - 0.051873116(2-1)(2-4) = 0.56584436$$

This is a more accurate result than obtained using linear interpolation. We now have a relative error of $\epsilon = 18.4\%$. Thus, the curvature introduced by the quadratic formula improves the interpolation compared with the result obtained using straight lines.



5.4 Lagrange Interpolating Polynomials

The general class of interpolating polynomials that require specification of certain points through which they must pass is called Lagrange polynomials. Suppose we want to determine a first degree polynomial that passes through two points (x_0, y_0) and (x_1, y_1). Let such a polynomial have the form:

$$\begin{aligned} P(x) &= \frac{(x-x_1)}{(x_0-x_1)} y_0 + \frac{(x-x_0)}{(x_1-x_0)} y_1 \\ &= L_0(x) y_0 + L_1(x) y_1 \end{aligned}$$

It is easy to verify that $P(x_0) = y_0$ and $P(x_1) = y_1$. Thus the polynomial agrees with the functional values at the two stipulated points. We also note the following about the quotients $L_0(x)$ and $L_1(x)$. When $x = x_0$, $L_0(x_0) = 1$ and $L_1(x_0) = 0$. When $x = x_1$, $L_0(x_1) = 0$ and $L_1(x_1) = 1$. Thus we need to construct the quotients $L_0(x)$ and $L_1(x)$ to determine the polynomial.

In general, to construct a polynomial of degree at most n that passes through the $n+1$ points $(x_0, f(x_0))$, $(x_1, f(x_1)), \ldots, (x_n, f(x_n))$, we need to construct for $k = 0, 1, \ldots, n$, a quotient $L_{n,k}(x)$ with the property that $L_{n,k}(x_i) = 0$ when $i \neq k$ and $L_{n,k}(x_k) = 1$. To satisfy $L_{n,k}(x_i) = 0$ for each $i \neq k$ requires that the numerator of $L_{n,k}$ to contain the term:

$$(x - x_0)(x - x_1)\ldots(x - x_{k-1})(x - x_{k+1})\ldots(x - x_n).$$

To satisfy $L_{n,k}(x_k) = 1$, the denominator of $L_{n,k}$ must equal the denominator of the above numerator evaluated at $x = x_k$. Thus:

$$\begin{aligned} L_{n,k}(x) &= \frac{(x - x_0)\ldots(x - x_{k-1})(x - x_{k+1})\ldots(x - x_n)}{(x_k - x_0)\ldots(x_k - x_{k-1})(x_k - x_{k+1})\ldots(x_k - x_n)} \\ &= \prod_{i=0, i\neq k}^{n} \frac{(x - x_i)}{(x_k - x_i)}. \end{aligned}$$

The lagrange interpolating polynomial is thus given by:

$$P(x) = L_{n,0}(x) f(x_0) + L_{n,1}(x) f(x_1) + \ldots + L_{n,n}(x) f(x_n) \tag{5.7}$$

If there is no confusion about the degree of the required polynomial we shall simply use L_k instead of $L_{n,k}$.

Error in Lagrange polynomial:

The error in the approximation by the Lagrange interpolating polynomial can be estimated if $f(x)$ is known as:

$$E(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{i=0}^{n} (x - x_i), \tag{5.8}$$

for some $\xi(x) \in (a, b)$, $a \leq x_0 \leq x - 1 \leq \ldots \leq x_n \leq b_n$, assuming $f^{(n+1)}(x)$ is continuous on $[a, b]$.

5.4.0.1 Example

Use a Lagrange interpolating polynomial of the first and second order to evaluate $ln(x)$ on the basis of the given data points and estimate the value at $x = 2$

$$L_i(x) = \prod_{i=0, i\neq k}^{n} \frac{x - x_k}{x_i - x_k}$$

$$\begin{aligned} x_0 = 1; &\qquad f(x_0) = 0 \\ x_1 = 4; &\qquad f(x_1) = 1.386294 \\ x_2 = 6; &\qquad f(x_2) = 1.791760 \end{aligned}$$

First Order:

We have the following equation for the first order Lagrange polynomial,

$$P_1(x) = \frac{x - x_1}{x_0 - x_1} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1)$$

Therefore we obtain,

$$\begin{aligned} P_1(2) &= \frac{2-4}{1-4}(0) + \frac{2-1}{4-1}(1.386294) \\ &= 0.4620981. \end{aligned}$$

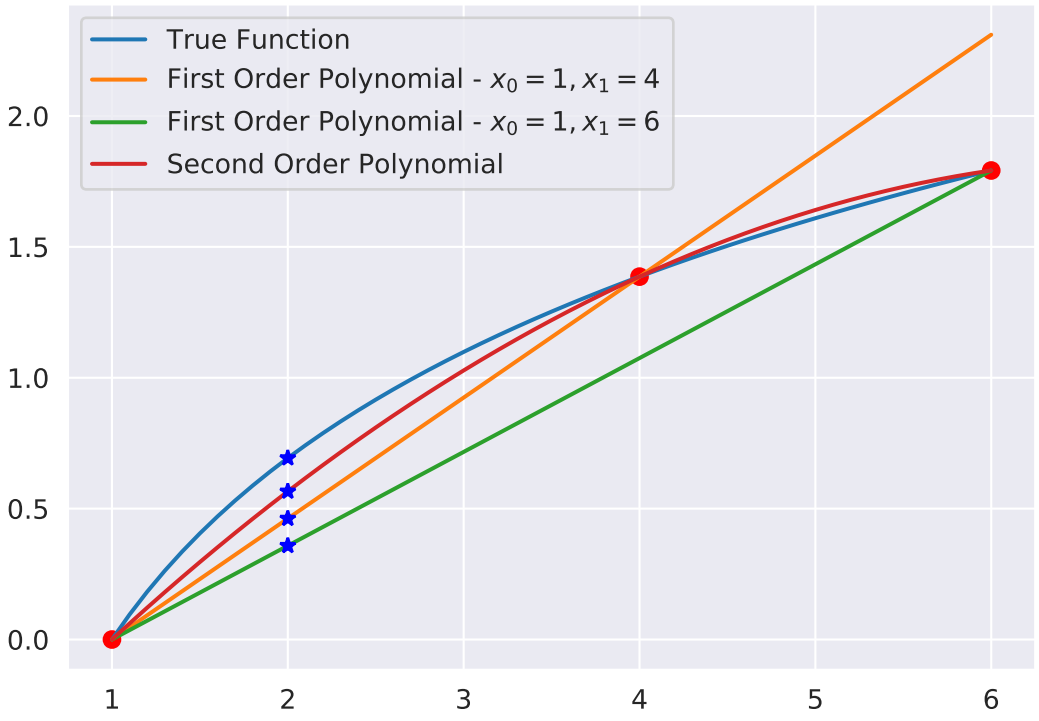
Second Order:

We have the following equation for the second order Lagrange polynomial,

$$P_2(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} f(x_1) + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} f(x_2)$$

Therefore we obtain,

$$\begin{aligned} P_2(2) &= \frac{(2-4)(2-6)}{(1-4)(1-6)}(0) + \frac{(2-1)(2-6)}{(4-1)(4-6)}(1.386294) + \frac{(2-1)(2-4)}{(6-1)(6-4)}(1.791760) \\ &= 0.565844 \end{aligned}$$



5.4.0.2 Example

Use the following data to approximate $f(1.5)$ using the Lagrange interpolating polynomial for $n = 1, 2$, and 3.

x_i values	1	1.3	1.6	1.9	2.2
$f(x_i)$	0.7651977	0.6200860	0.4554022	0.2818186	0.1103623

The interpolating polynomial show be,

$$P(x) = (((0.0018251x + 0.0552928)x - 0.343047)x + 0.0733913)x + 0.977735,$$

which gives,

$$P(1.5) = 0.508939.$$

5.5 Newton's Divided Differences

We first introduce the notation for the divided differences:

- The zeroth divided difference of f w.r.t. x_i is $f[x_i] = f(x_i) = f_i$.
- The first divided difference of f w.r.t. x_i and x_{i+1} is:

$$f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i} = \frac{f_{i+1} - f_i}{x_{i+1} - x_i}$$

- The Second divided difference of f w.r.t. x_i, x_{i+1} and x_{i+2} is:

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i}$$

- The k^{th} divided difference of f w.r.t. $x_i, x_{i+1}, \cdots, x_{i+k}$ is:

$$f[x_i, x_{i+1}, \cdots, x_{i+k}] = \frac{f[x_{i+1}, x_{i+2}, \cdots, x_{i+k}] - f[x_i, x_{i+2}, \cdots, x_{i+k-1}]}{x_{i+k} - x_i}$$

We now fit an nth degree interpolating polynomial to the $n+1$ data points $(x_i, f(x_i))$, $i = 0, 1, \cdots, n$ in the form:

$$P_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_n(x - x_0)(x - x_1)\cdots(x - x_{n-1}).$$

Since the polynomial must pass through the points (x_i, f_i) we have:

- $x = x_0$ $P_n(x_0) = f_0 = a_0 = f[x_0]$
- $x = x_1$ $P_n(x_1) = f_1 = f[x_0] + a_1(x_1 - x_0) = f[x_1] \quad \Rightarrow a_1 = \frac{f[x_1] - f[x_0]}{x_1 - x_0} = f[x_0, x_1]$.
- $x = x_2$

  $$P_n(x_2) = f_2 = f[x_2] = f[x_0] + f[x_0, x_1](x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1),$$

  and therefore:

  $$a_2 = \frac{f[x_2] - f[x_0] - f[x_0, x_1](x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)}$$

  With some algebraic manipulation it can be shown that:

$$a_2 = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} = f[x_0, x_1, x_2]$$

In general:

$$a_k = f[x_0, x_1, \cdots, x_k]$$

so that:

$$\begin{aligned} P_n(x) &= f[x_0] + \sum_{k=1}^{n} f[x_0, \cdots, x_k](x - x_0) \cdots (x - x_{k-1}) \\ &= f[x_0] + \sum_{k=1}^{n} f[x_0, \cdots, x_k] \prod_{i=0}^{k-1} (x - x_i) \end{aligned} \tag{5.9}$$

called **Newton's divided difference interpolating polynomial**. All divided differences are calculated in a similar process and the results are usually tabulated in:

a divided difference table:

x_i	$f[x_i]$	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}]$	$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}, x_{i+4}]$
x_0	$f[x_0]$				
		$f[x_0, x_1]$			
x_1	$f[x_1]$		$f[x_0, x_1, x_2]$		
		$f[x_1, x_2]$		$f[x_0, x_1, x_2, x_3]$	
x_2	$f[x_2]$		$f[x_1, x_2, x_3]$		$f[x_0, x_1, x_2, x_3, x_4]$
		$f[x_2, x_3]$		$f[x_1, x_2, x_3, x_4]$	
x_3	$f[x_3]$		$f[x_2, x_3, x_4]$		
		$f[x_3, x_4]$			
x_4	$f[x_4]$				

5.5.0.1 Exercise

Use a third degree polynomial passing through the points $(1, \ln 1)$, $(4, \ln 4)$, $(5, \ln 5)$ and $(6, \ln 6)$ to estimate $\ln 2$. (Ans: $P_3(2) = 0.62876869$).

5.5.0.2 Example

Find a polynomial satisfied by $(-4, 1245)$, $(-1, 33)$, $(0, 5)$, $(2, 9)$, $(5, 1335)$.

Solution:

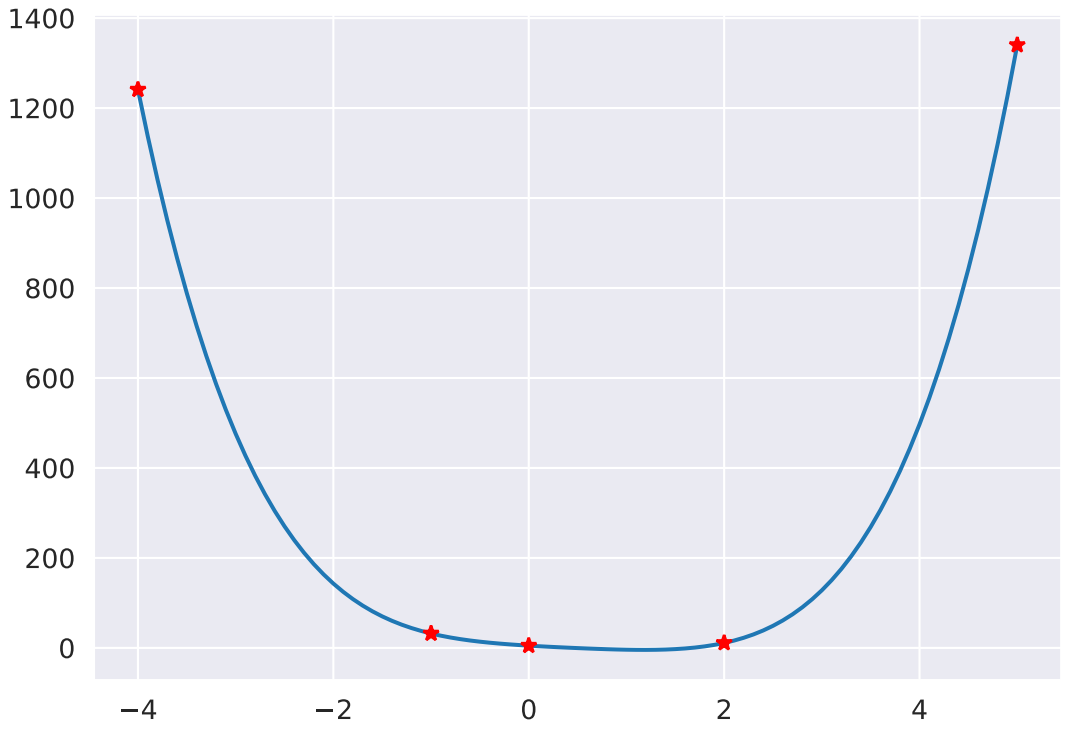
x_i	$f(x_i)$	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}]$	$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}, x_{i+4}$
-4	1245				
		-404			
-1	33		94		
		-28		-14	
0	5		10		3
		2		13	
2	9		88		
		442			
5	1335				

Hence,

$$\begin{aligned} P_4(x) &= 1245 - 404(x+4) + 94(x+4)(x+1) - 14(x+4)(x+1)(x) \\ &\quad +3(x+4)(x+1)x(x-2) \\ &= 3x^4 - 5x^3 + 6x^2 - 14x + 5. \end{aligned} \tag{5.10}$$

Note: If an extra data point $(x, f(x))$ is added, we only need to add an additional term to the $P_n(x)$ already found.

```
x = np.array([-4, -1, 0, 2, 5])
y = np.array([1245, 33, 5, 9, 1335])
f = lambda x: 3*x**4 - 5*x**3 + 6*x**2 -13*x + 5
X = np.arange(-4, 5.1, 0.1)
Y = f(X)
plt.figure()
plt.plot(X, Y)
plt.plot(x, f(x), 'r*')
plt.show()
```



In general if $P_n(x)$ is the interpolating polynomial through the $(n+1)$ points (x_i, f_i), $i = 0, 1, \cdots, n$, then the Newton's divided difference formula gives P_{n+1} through these points plus one more point (x_{n+1}, f_{n+1}) as i.e.,

$$P_{n+1}(x) = P_n(x) + f[x_0, x_1, \cdots, x_n, x_{n+1}] \prod_{i=0}^{n} (x - x_i) \tag{5.11}$$

$P_{n+1}(x)$ improves the interpolation by introducing additional curvature.

5.5.1 Errors of Newton's interpolating polynomials

Let $P_{n+1}(x)$ be the $(n+1)$th degree polynomial which fits $y = f(x)$ at the $n+2$ points, $(x_0, f(x_0))$, $(x_1, f(x_1)), \cdots, (x_n, f(x_n)), (x, f(x))$. The last point is a general point. Then:

$$P_{n+1}(x) = P_n(x) + f[x_0, x_1, \cdots, x_n, x] \prod_{i=0}^{n} (x - x_i)$$

since $f(x) \approx P_{n+1}(x)$, we have

$$\epsilon_n(x) = P_{n+1}(x) - P_n(x) = f[x_0, x_1, \cdots, x_n, x] \prod_{i=0}^{n} (x - x_i)$$

Remarks: For $n = 0$,

$$f[x_0, x] = \frac{f(x) - f(x_0)}{x - x_0}.$$

We have:

- (Mean value theorem) $f[x_0, x] = \frac{f(x)-f(x_0)}{x-x_0} = f'(\xi), \quad \xi \in [x_0, x]$.
- (Definition of a derivative) $\lim x \to x_0 f[x_0, x] = f'(x_0)$.

In general, it can be shown that

$$f[x_0, x_1, \cdots, x_n] = \frac{1}{n!} f^{(n)}(\xi), \quad \xi \in [x_0, x_n]$$

and hence:

$$f[x_0, x_1, \cdots, x_n, x] = \frac{1}{(n+1)!} f^{(n+1)}(\xi), \quad \xi \in [x_0, x] \tag{5.12}$$

The error is then:

$$\begin{aligned} \epsilon_n(x) &= f[x_0, x_1, \cdots, x_n, x] \prod_{i=0}^{n} (x - x_i) \\ &= \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^{n} (x - x_i), \quad \xi \in [x_0, x] \end{aligned} \tag{5.13}$$

5.6 Cubic Splines Interpolation

The previous sections concerned the approximation of arbitrary functions on closed intervals by the use of polynomials. However, the oscillatory nature of the high-degree polynomials, and the property that a fluctuation over a small portion of the interval can induce large fluctuations over the entire range, restricts their use.

The concept of the spline fit originated from the drafting technique of using a thin, flexible strip to draw a smooth curve through a set of given points. The flexible spline was pinned or held by weights so that the curve passed through all the data points. The spline passed smoothly from one interval to the next because of the laws governing beam flexure.

The most widely used spline fitting is the **cubic spline**. In the cubic spline procedure, a cubic polynomial is passed through each pair of points in such a manner that the first and second derivatives are continuous throughout the table of points.

A cubic spline s with knots $x_0 < x_1 < \cdots < x_n$ satisfies:

- s is a polynomial of degree ≤ 3 in each knot interval $I_i = [x_{i-1}, x_i]$, $i = 1, 2, \cdots, n$

For $x_{i-1} < x < x_i$ let $s(x) = s_i(x)$

The first condition is that the spline must pass through all the data points. So:

$$f_i = a_i + b_i(x_i - x_i) + c_i(x_i - x_i)^2 + d_i(x_i - x_i)^3, \tag{5.14}$$

which simplifies to,

$$a_i = f_i. \tag{5.15}$$

Therefore, the constant in each cubic must be equal to the value of the dependent variable at the beginning of the interval. This result can incorporated into,

$$s_i(x) = f_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3. \tag{5.16}$$

Where the coefficients, b_i, d_i are solved using the following,

$$b_i = \frac{f_{i+1} - f_i}{h_i} - \frac{h_i}{3}(2c_i + c_{i+1}), \tag{5.17}$$

$$d_i = \frac{c_{i+1} - c_i}{3h_i}, \tag{5.18}$$

where h_i is simply,

$$h_i = x_{i+1} - x_i. \tag{5.19}$$

The solution for c_i is somewhat more complicated. This enforces us to make use of systems of linear equations by solving the following tridiagonal system,

$$\begin{bmatrix} 1 & & & & \\ h_1 & 2(h_1+h_2) & h_2 & & \\ \ddots & \ddots & \ddots & & \\ & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ & & & 1 \end{bmatrix} \begin{Bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \\ c_n \end{Bmatrix} = \begin{bmatrix} 0 \\ 3(f[x_3,x_2] - f[x_2,x_1]) \\ \vdots \\ 3(f[x_n,x_{n-1}] - f[x_{n-1},x_{n-2}]) \\ 0 \end{bmatrix}.$$

5.6.0.1 Example

Consider the table below. Fit cubic splines to the data and utilize the results to estimate the value at $x = 5$.

i	x_i	f_i
1	3	2.5
2	4.5	1
3	7	2.5
4	9	0.5

Solution:

The first step is to generate the set of simultaneous equations that will be utilized to determine the c coefficients:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1.5 & 8 & 2.5 & 0 \\ 0 & 2.5 & 9 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{Bmatrix} = \begin{bmatrix} 0 \\ 3(0.6+1) \\ 3(-1-0.6) \\ 0 \end{bmatrix}.$$

$$\Rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1.5 & 8 & 2.5 & 0 \\ 0 & 2.5 & 9 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{Bmatrix} = \begin{bmatrix} 0 \\ 4.8 \\ -4.8 \\ 0 \end{bmatrix}.$$

Therfore:

$$\Rightarrow \bar{c} = \begin{bmatrix} 0 \\ 0.839543726 \\ -0.766539924 \\ 0 \end{bmatrix}.$$

Using our values for c we obtain the following for our d's,

$$\begin{aligned} d_1 &= 0.186565272, \\ d_2 &= -0.214144487, \\ d_3 &= 0.127756654. \end{aligned}$$

We can then compute the b's using equation (1.4),

$$\begin{aligned} b_1 &= -1.419771863, \\ b_2 &= -0.160456274, \\ b_3 &= 0.022053232. \end{aligned}$$

These results allow us to develop the cubic splines for each interval using Equation (5.16):

$$\begin{aligned} s_1(x) &= 2.5 - 1.419771863(x-3) + 0.186565272(x-3)^3, \\ s_2(x) &= 1 - 0.160456274(x-4.5) + 0.839543726(x-4.5)^2 - 0.214144487(x-4.5)^3, \\ s_3(x) &= 2.5 + 0.022053232(x-7) - 0.766539924(x-7)^2 + 0.127756654(x-7)^3. \end{aligned}$$

The three equations can then be employed to compute values within each interval. For example, the value at $x = 5$, which falls within the second interval, is calculated as,

$$\begin{aligned} s_2(5) &= 1 - 0.160456274(5-4.5) + 0.839543726(5-4.5)^2 - 0.214144487(5-4.5)^3 \\ &= 1.102889734. \end{aligned}$$

5.6.1 Runge's Phenomenon

A major problem with interpolation is **Runge's Phenomenon**. Let us consider an example in *Mathematica*:

```
ClearAll[data, x];
data = RandomReal[{-10, 10}, 20];

ListPlot[data]

Manipulate[
 Show[
  Plot[InterpolatingPolynomial[data[[1 ;; n]], x], {x, 1, n},
   PlotRange -> All],
  ListPlot[data, PlotStyle -> Directive[PointSize[Large], Red]],
  PlotRange -> All
  ], {n, 2, Length[data], 1}]

pctrl[d_, param_, noeud_] :=
 LinearSolve[
  Module[{n = Length[d]},
   Table[BSplineBasis[{3, noeud}, j - 1, param[[i]]], {i, n}, {j,
     n}]], d]
```

```
tcentr[d_] :=
 Module[{a},
  a = Accumulate[
    Table[Norm[d[[i + 1]] - d[[i]]]^(1/2), {i, Length[d] - 1}]];
  N[Prepend[a/Last[a], 0]]]

noeudmoy[d_, param_] :=
 Join[{0, 0, 0, 0},
  Table[1/3*Sum[param[[i]], {i, j, j + 2}], {j, 2,
    Length[param] - 3}], {1, 1, 1, 1}]

dpts = Table[{i, data[[i]]}, {i, Length[data]}];


Manipulate[Module[{pCt},
  pCt = pctrl[dpts[[1 ;; n]], tcentr[dpts[[1 ;; n]]],
    noeudmoy[dpts[[1 ;; n]], tcentr[dpts[[1 ;; n]]]]];
  Show[
   ParametricPlot[
    BSplineFunction[pCt,
      SplineKnots ->
       noeudmoy[dpts[[1 ;; n]], tcentr[dpts[[1 ;; n]]]][[x]], {x, 0,
     1}, PlotRange -> All],
   ListPlot[data, PlotStyle -> Directive[PointSize[Large], Red]],
   PlotRange -> All
   ]], {n, 4, Length[data], 1}]
```

Thus we can see that high order polynomials lead to an exponential growth of the infinity norm error. To overcome this we used the splines technique from above, however, another method one could use is Chebyshev polynomials. Here points are distributed more densely towards the bounds of the interval.

5.6.2 Exercises

1. Given the data points:

x	-1.2	0.3	1.1
y	-5.76	-5.61	-3.69

determine y at $x = 0$ using (a) Lagrange's method and (b) Newton's Divided Differences.

2. Given the data:

x	0.4	0.5	0.7	0.8
y	1.086	1.139	1.307	1.435

Estimate $f(0.6)$ from the data using: (a) a second degree Lagrange polynomial (b) a third degree Lagrange polynomial

3. Given $f(-2) = 46$, $f(-1) = 4$, $f(1) = 4$, $f(3) = 156$, $f(4) = 484$, use Newton Divided Differences to estimate

$f(0)$.

4. Let $P(x)$ be the degree 5 polynomial that takes the value 10 at $x = 1,2,3,4,5$ and the value 15 at $x = 6$. Find $P(7)$.
5. Write down a polynomial of degree exactly 5 that interpolates the four points $(1,1),(2,3),(3,3),(4,4)$.
6. Find $P(0)$, where $P(x)$ is the degree 10 polynomial that is zero at $x = 1,\ldots,10$ and satisfies $P(12) = 44$.
7. Write down the degree 25 polynomial that passess through the points $(1,-1),(2,-2),\ldots,(25,-25)$ and has constant term equal to 25.
8. Use the method of divided differences to find the degree 4 interpolating polynomial $P_4(x)$ for the data $(0.6,1.433329),(0.7,1.632316),(0.8,1.896481),(0.9,2.247908)$ and $(1.0,2.718282)$. Next calculate $P_4(0.82)$ and $P_4(0.98)$. The aforementioned data is sampled from the function $f(x) = e^x$. Compute the absolute and relative errors of your approximations at $P_4(0.82)$ and $P_4(0.98)$.

Chapter 6

Least Squares

When considering experimental data it is commonly associated with noise. This noise could be resultant of measurement error or some other experimental inconsistency. In these instances, we want to find a curve that fits the data points "on the average". That is, we do not want to overfit the data, thereby amplifying any of the noise. With this in mind, the curve should have the simplest form (i.e. lowest order polynomial possible). Let:

$$f(x) = f(x, a_1, a_2, \ldots, a_m),$$

be the function that is to be fitted to the n data points (x_i, y_i), $i = 1, 2, \ldots, n$. Thus, we have a function of x that contains the parameters a_j, $j = 1, 2, \ldots, m$, where $m < n$. The shape of $f(x)$ is known a priori, normally from the theory associated with the experiment in question. This means we are looking to fit the best parameters. Thus curve fitting is a two step process; (i) selecting the correct form of $f(x)$ and (ii) computing the parameters that produce the best fit to the data.

The notion of **best** fit (at least for the purpose of this course) considers noise bound to the y-coordinate. The most common of which is measured by the *least squares fit*, which minimises:

$$S(a_1, a_2, \ldots, a_m) = \sum_{i=1}^{n} \left[y_i - f(x_i) \right]^2, \tag{6.1}$$

with respect to each a_j. The optimal values of the parameters are given by the solution of the equations:

$$\frac{\partial S}{\partial a_k} = 0, \quad k = 1, 2, \ldots, m. \tag{6.2}$$

We measure the residual as $r_i = y_i - f(x_i)$ from Equation (6.1) which represent the discrepancy between the data points and the fitting function at x_i. The function S is the sum of the squares of all residuals.

A Least squares problem is said to be **linear** if the fitting function is chosen as a linear combination of functions $f_j(x)$:

$$f(x) = a_1 f_1(x) + a_2 f_2(x) + \ldots + a_m f_m(x). \tag{6.3}$$

Here an example could be where $f_1(x) = 1$, $f_2(x) = x$, $f_3(x) = x^2$ etc. Often these polynomials can be nonlinear and become increasingly difficult to solve. For the purpose of this course we will only consider linear least squares.

6.1 Linear Least Squares

We fit the straight line $y = a_0 + a_1 x$ through some given n points. The sum of the squares of the deviations is

$$S = \sum_{i=1}^{n} [y_i - f(x_i)]^2 = \sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i)]^2$$

A necessary condition for $S(a_0, a_1)$ to be a minimum is that the first partial derivatives of S w.r.t. a_0 and a_1 must be zero:

$$\frac{\partial E}{\partial a_0} = -2\sum_{i=1}^{n}[y_i - a_0 - a_1 x_i)] = 0 \tag{6.4}$$

$$\frac{\partial E}{\partial a_1} = -2\sum_{i=1}^{n} x_i[y_i - a_0 - a_1 x_i)] = 0 \tag{6.5}$$

We can rewrite these sums as:

$$a_0 n + a_1 \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \tag{6.6}$$

$$a_0 \sum_{i=1}^{n} x_i + a_1 \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} x_i y_i \tag{6.7}$$

These equations are called the **normal equations.** They can be solved simultaneously for a_1:

$$a_1 = \frac{n\sum_i x_i y_i - \sum_i x_i \sum_i y_i}{n\sum_i x_i^2 - \left(\sum_i x_i\right)^2} \tag{6.8}$$

This result can then be used in conjunction with the Equation (6.6) to solve for a_0:

$$a_0 = \frac{1}{n}\left(\sum_{i=1}^{n} y_i - a_1 \sum_{i=1}^{n} x_i\right). \tag{6.9}$$

So in matrix form:

$$\begin{bmatrix} n & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i y_i \end{bmatrix}. \tag{6.10}$$

Therefore:

$$\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} n & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \end{bmatrix}^{-1} \begin{bmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i y_i \end{bmatrix}. \tag{6.11}$$

6.1.0.1 Example

Consider the data:

x_i	1	2	3	4	5	6	7
y_i	0.5	2.5	2.0	4.0	3.5	6.0	5.5

To find the least squares line approximation of this data, extend the table and sum the columns, as below:

x_i	y_i	x_i^2	$x_i y_i$
1	0.5	1	0.5
2	2.5	4	5.0
3	2.0	9	6.0
4	4.0	16	16.0
5	3.5	25	16.5
6	6.0	36	36.0

x_i	y_i	x_i^2	$x_i y_i$
7	5.5	49	37.5
$\sum = 28$	$\sum = 24$	$\sum = 140$	$\sum = 119.5$

$$a_1 = \frac{7(119.5) - 28(24)}{7(140) - 28^2} = 0.8393$$

and hence:

$$a_0 = \frac{24 - 0.8393(28)}{7} = 0.0714$$

The least squares linear fit is:

$$y = 0.0714 + 0.8393x$$

Or alternatively in matrix form we have:

$$\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 7 & 28 \\ 28 & 140 \end{bmatrix}^{-1} \begin{bmatrix} 24 \\ 119.5 \end{bmatrix}$$

Solving gives the following:

```
import numpy as np
import numpy.linalg as LA
A    = np.array([[7, 28],[28, 140]])
b    = np.array([24, 119.5])
tans = np.dot(LA.inv(A), b)
# Solving the matrix equation gives:
print('The value for a_0 is:', tans[0])
```

```
## The value for a_0 is: 0.0714285714285694
```

```
print('The value for a_1 is:', tans[1])
# Check using the builtin functions:
```

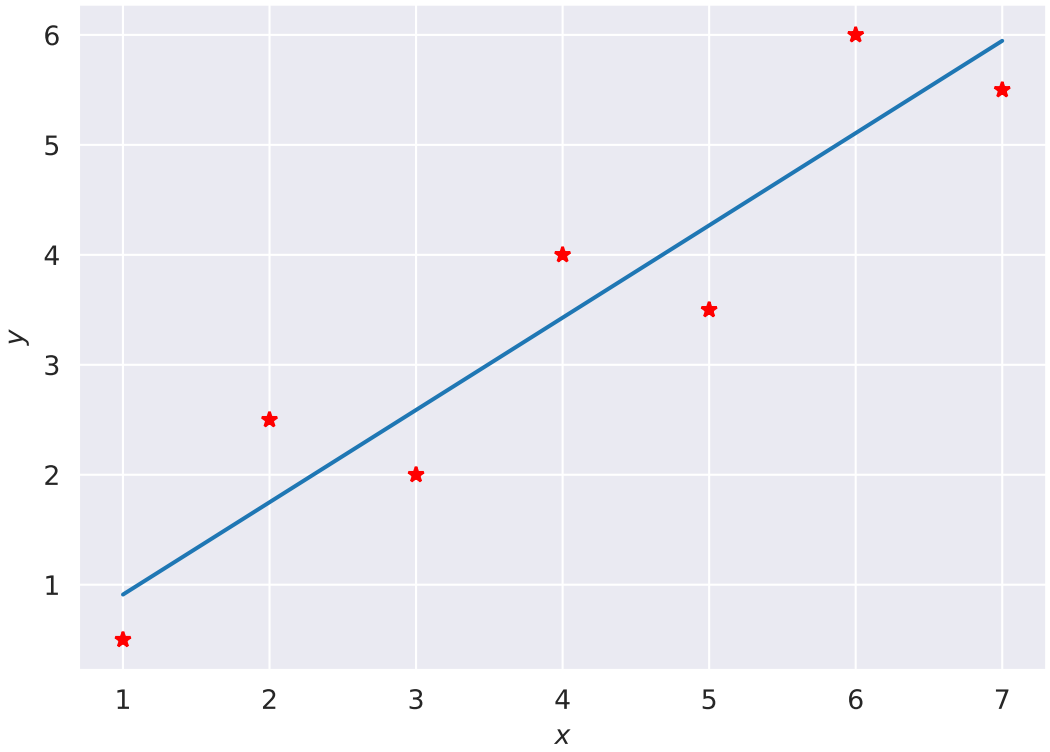
```
## The value for a_1 is: 0.839285714285714
```

```
x    = np.array([1, 2, 3, 4, 5, 6, 7])
y    = np.array([0.5, 2.5, 2.0, 4.0, 3.5, 6.0, 5.5])
bans = np.polyfit(x, y, 1)
# Note polyfit returns function of form P(x) = p[0]*x**degree .... (This is the opposite direction of a
bans = bans[::-1]
print('The value for a_0 with builtin is:', bans[0])
```

```
## The value for a_0 with builtin is: 0.07142857142857066
```

```
print('The value for a_1 with builtin is:', bans[1])
```

```
## The value for a_1 with builtin is: 0.8392857142857142
```



6.2 Polynomial Least Squares

The least squares procedure above can be readily extended to fit the data to an mth degree polynomial:

$$f(x) = P_m(x) = a_0 + a_1 x + \cdots + a_m x^m \tag{6.12}$$

through some n data points $(x_1, P_m(x_1)), (x_2, P_m(x_2)), \ldots, (x_m, P_m(x_n))$, where $m \leq n-1$. Then, S takes the form:

$$S = \sum_{i=1}^{n} [y_i - f(x_i)]^2 \tag{6.13}$$

which depends on the $m+1$ parameters $a_0,\ a_1, \cdots,\ a_m$. We then have $m+1$ conditions:

$$\frac{\partial E}{\partial a_0} = 0,\ \frac{\partial E}{\partial a_1} = 0,\ \cdots, \frac{\partial E}{\partial a_m} = 0$$

which gives a system of $m+1$ normal equations:

$$a_0\, n \quad + \quad a_1 \sum_{i=1}^{n} x_i + a_2 \sum_{i=1}^{n} x_i^2 + \cdots + a_m \sum_{i=1}^{n} x_i^m = \sum_{i=1}^{n} y_i \tag{6.14}$$

$$a_0 \sum_{i=1}^{n} x_i \quad + \quad a_1 \sum_{i=1}^{n} x_i^2 + a_2 \sum_{i=1}^{n} x_i^3 + \cdots + a_m \sum_{i=1}^{n} x_i^{m+1} = \sum_{i=1}^{n} x_i y_i \tag{6.15}$$

$$a_0 \sum_{i=1}^{n} x_i^2 \quad + \quad a_1 \sum_{i=1}^{n} x_i^3 + a_2 \sum_{i=1}^{n} x_i^4 + \cdots + a_m \sum_{i=1}^{n} x_i^{m+2} = \sum_{i=1}^{n} x_i^2 y_i \tag{6.16}$$

$$\vdots \qquad \vdots \tag{6.17}$$

$$a_0 \sum_{i=1}^{n} x_i^m \quad + \quad a_1 \sum_{i=1}^{n} x_i^{m+1} + a_2 \sum_{i=1}^{n} x_i^{m+2} + \cdots + a_m \sum_{i=1}^{n} x_i^{2m} = \sum_{i=1}^{n} x_i^m y_i \tag{6.18}$$

These are $m+1$ equations and have $m+1$ unknowns: a_0, a_1, $\cdots a_m$.

So for a quadratic polynomial fit, $m = 2$,and the required polynomial is $f(x) = a_0 + a_1 x + a_2 x^2$ obtained from solving the normal equations:

$$a_0 n \quad + \quad a_1 \sum_{i=1}^{n} x_i + a_2 \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} y_i \tag{6.19}$$

$$a_0 \sum_{i=1}^{n} x_i \quad + \quad a_1 \sum_{i=1}^{n} x_i^2 + a_2 \sum_{i=1}^{n} x_i^3 = \sum_{i=1}^{n} x_i y_i \tag{6.20}$$

$$a_0 \sum_{i=1}^{n} x_i^2 \quad + \quad a_1 \sum_{i=1}^{n} x_i^3 + a_2 \sum_{i=1}^{n} x_i^4 = \sum_{i=1}^{n} x_i^2 y_i \tag{6.21}$$

for a_0, a_1, and a_2.

Note: This system is symmetric and can be solved using Gauss elimination.

6.2.0.1 Exercise

Fit a second degree polynomial to the data

x_i	0	1	2	3	4	5
y_i	2.1	7.7	13.6	27.2	40.9	61.1

```
import numpy.linalg as LA
x      = np.array([0, 1, 2, 3, 4, 5])
y      = np.array([2.1, 7.7, 13.6, 27.2, 40.9, 61.1])
n      = len(x)
sumX   = sum(x)
sumY   = sum(y)
sumX2  = sum(x**2)
sumX3  = sum(x**3)
sumX4  = sum(x**4)
sumXY  = sum(x * y)
sumXXY = sum(x**2 * y)
A    = np.array([[n, sumX, sumX2],[sumX, sumX2, sumX3], [sumX2, sumX3, sumX4]])
print(A)
```

```
## [[  6  15  55]
##  [ 15  55 225]
##  [ 55 225 979]]
```

```
b    = np.array([sumY, sumXY, sumXXY])
print(b)
```

```
## [ 152.6  585.6 2488.8]
```

```
tans = np.dot(LA.inv(A), b)
# Solving the matrix equation gives:
print('The value for a_0 is:', tans[0])
```

```
## The value for a_0 is: 2.478571428571229
```

```
print('The value for a_1 is:', tans[1])
```

```
## The value for a_1 is: 2.3592857142858747
```

```
print('The value for a_2 is:', tans[2])
# Check using the builtin functions:
```

```
## The value for a_2 is: 1.8607142857142804
```

```
bans = np.polyfit(x, y, 2)
# Note polyfit returns function of form P(x) = p[0]*x**degree .... (This is the opposite direction of a
bans = bans[::-1]
print('The value for a_0 with builtin is:', bans[0])
```

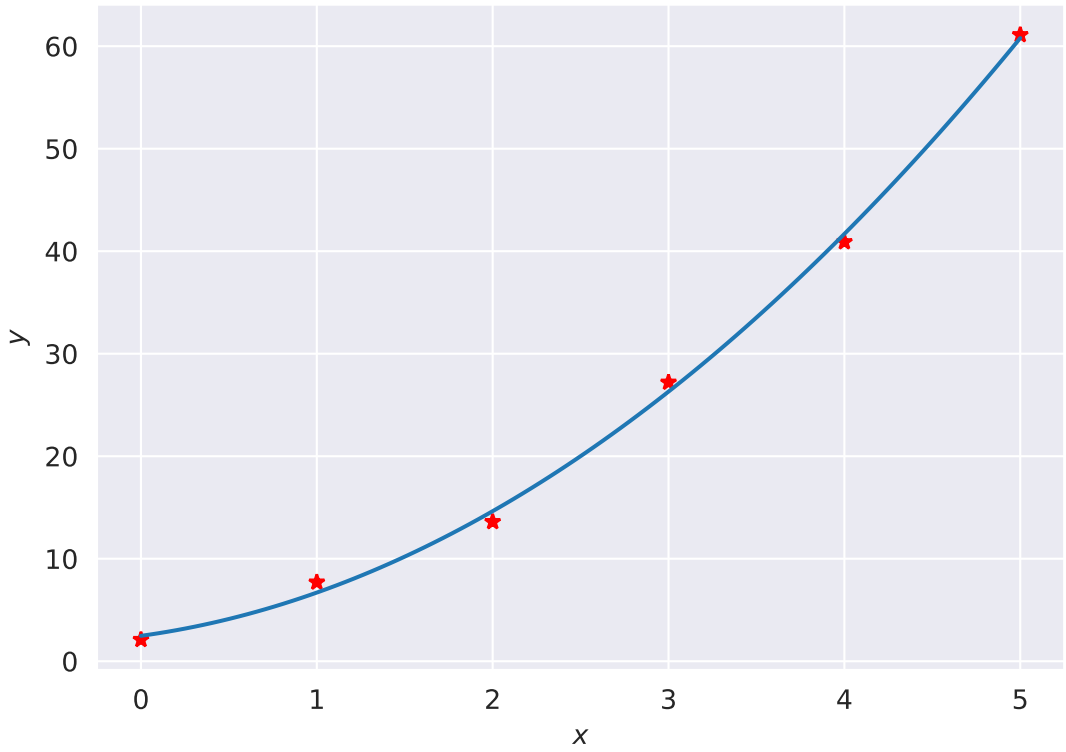
```
## The value for a_0 with builtin is: 2.4785714285714135
```

```
print('The value for a_1 with builtin is:', bans[1])
```

```
## The value for a_1 with builtin is: 2.3592857142857286
```

```
print('The value for a_1 with builtin is:', bans[2])
```

```
## The value for a_1 with builtin is: 1.8607142857142822
```



Remark: As the degree m increases the coefficient matrix becomes extremely ill-conditioned. It is therefore not recommended to fit least squares polynomials of degree greater than 4 to given data points.

Also, it would be common practice to use built-in libraries to do these computations instead of programming it yourself. In addition, any real world scenario would likely involve a massive number of data points. Gradient descent techniques could also be applied. You may find these within machine learning courses etc.

6.3 Least Squares Exponential Fit

Frequently a theory may suggest a model other than a polynomial fit. A common functional form for the model is the exponential function:

$$y = ae^{bx}. \tag{6.22}$$

for some constants a and b. We have from Equation (6.13):

$$S = \sum_{i=1}^{n} [y_i - ae^{bx_i}]^2. \tag{6.23}$$

When the derivatives of S with respect to a and b are set equal to zero the resulting equations are:

$$\frac{\partial E}{\partial a} = -2\sum_{i=1}^{n} e^{bx_i}[y_i - ae^{bx_i}] = 0 \tag{6.24}$$

$$\frac{\partial E}{\partial b} = -2\sum_{i=1}^{n} ax_i e^{bx_i}[y_i - ae^{bx_i}] = 0 \tag{6.25}$$

These two equations in two unknowns are nonlinear and generally difficult to solve.

It is sometimes possible to "linearise" the normal equations through a change of variables. If we take natural logarithm of our equation (6.22) we have:

$$\ln(y) = \ln(ae^{bx}) = \ln(a) + bx$$

We introduce the variable $Y = \ln(y)$, $a_0 = \ln(a)$ and $a_1 = b$. Then the linearized equation becomes:

$$Y(x) = a_0 + a_1 x, \tag{6.26}$$

and the ordinary least squares analysis may then be applied to the problem. Once the coefficients a_0 and a_1 have been determined, the original coefficients can be computed as $a = e^{a_0}$ and $b = a_1$.

6.3.0.1 Example

Fit an exponential function to the following data

x_i	1.00	1.25	1.50	1.75	2.00
y_i	5.10	5.79	6.53	7.45	8.46

To fit an exponential least squares fit to this data, extend the table as:

x_i	y_i	$Y_i = \ln y_i$	x_i^2	$x_i Y_i$
1.00	5.10	1.629	1.0000	1.629
1.25	5.79	1.756	1.5625	2.195
1.50	6.53	1.876	2.2500	2.814
1.75	7.45	2.008	3.0625	3.514
2.00	8.46	2.135	4.000	4.270
$\sum = 7.5$	$\sum = 33.3$	$\sum = 9.404$	$\sum = 11.875$	$\sum = 14.422$

Using the normal equations for linear least squares give:

$$a_1 = b = \frac{5(14.422) - 7.5(9.404)}{5(11.875) - (7.5)^2} = 0.5056$$

and hence:

$$a_0 = \ln a = \frac{9.404 - 0.5056(7.5)}{5} = 1.122, \quad a = e^{1.122}$$

The exponential fit is:

$$Y = 1.122 + 0.5056x \tag{6.27}$$

$$\ln y = 1.122 + 0.5056x \tag{6.28}$$

$$y = 3.071e^{0.5056x} \tag{6.29}$$

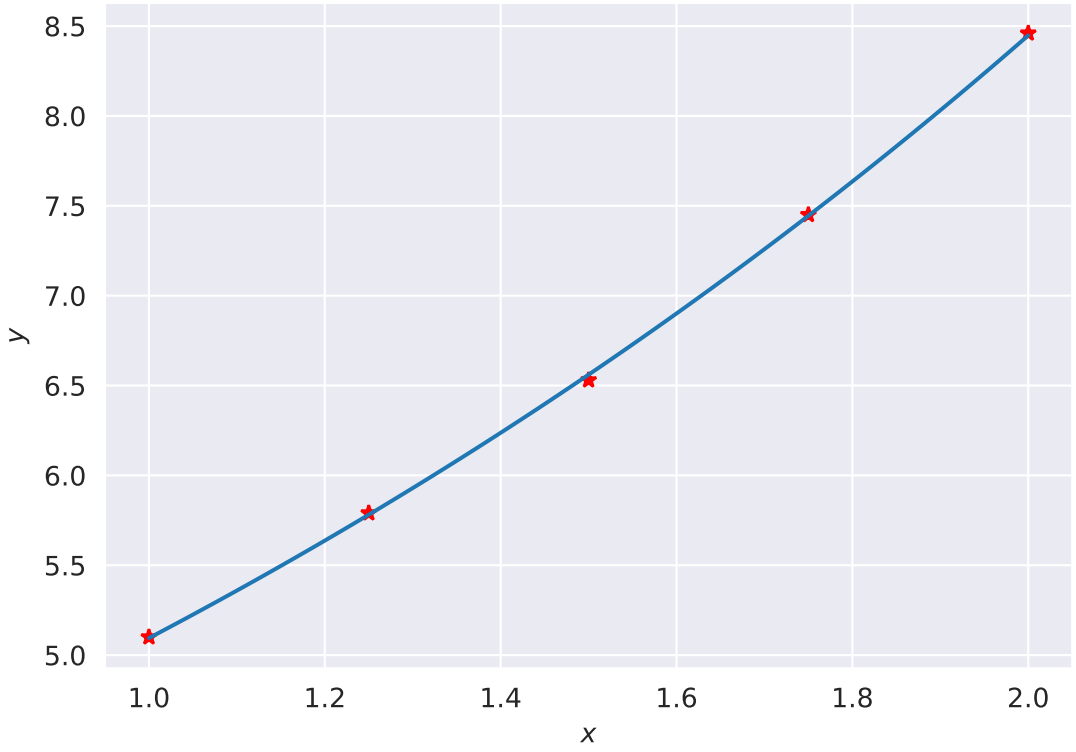
```
import numpy.linalg as LA
x      = np.array([1.0, 1.25, 1.5, 1.75, 2.0])
y      = np.array([5.1, 5.79, 6.53, 7.45, 8.46])
sumX   = sum(x)
sumY   = sum(y)
```

```
bans    = np.polyfit(x, np.log(y), 1)
# Note polyfit returns function of form P(x) = p[0]*x**degree .... (This is the opposite direction of a
bans    = bans[::-1]
bans[0] = np.exp(bans[0])
print('The value for a_0 with builtin is:', bans[0])
```

```
## The value for a_0 with builtin is: 3.072492713621622
```

```
print('The value for a_1 with builtin is:', bans[1])
```

```
## The value for a_1 with builtin is: 0.5057196034329074
```



6.3.1 Exercises

- Find the least squares polynomials of degrees one, two and three for the data, computing the error S in each case.

x	1.0	1.1	1.3	1.5	1.9	2.1
y	1.84	1.96	2.21	2.45	2.94	3.18

Ans:

$$y = 0.6209 + 1.2196x, \quad y = 0.5966 + 1.2533x - 0.0109x^2,$$
$$y = -0.01x^3 + 0.0353x^2 + 1.185x + 0.629$$

- An experiment is performed to define the relationship between applied stress and the time to fracture for a stainless steel. Eight different values of stress are applied and the resulting data is:

Applied stress, x, kg/mm^2	5	10	15	20	25	30	35	40
Fracture time, t, h	40	30	25	40	18	20	22	15

Use a linear least squares fit to determine the fracture time for an applied stress of 33 kg/mm^{2}to a stress. (Ans: $t = 39.75 - 0.6x, \quad t = 19.95$ hours)

- Fit a least squares exponential model to:

x	0.05	0.4	0.8	1.2	1.6	2.0	2.4
y	550	750	1000	1400	2000	2700	3750

(Ans: $y = 530.8078e^{0.8157x}$)

Chapter 7

Ordinary Differentiable Equations (ODEs)

Ordinary differential equations govern a great number of many important physical processes and phenomena. Not all differential equations can be solved using analytic techniques. Consequently, numerical solutions have become an alternative method of solution, and these have become a very large area of study.

Importantly, we note the following:

- By itself $y' = f(x, y)$ does not determine a unique solution.
- This simply tells us the slope $y'(x)$ of the solution function at each point, but not the actual value $y(x)$ at any point.
- There are an infinite family of functions satisfying an ODE.
- To single out a particular solution, a value y_0 of the solution function must be specified at some point x_0. These are called initial value problems.

7.1 Initial Value Problems

The general first order equation can be written as:

$$\frac{dy}{dx} = f(x, y), \tag{7.1}$$

with $f(x, y)$ given. Together with this may be given an initial condition, say $y(x_0) = y_0$, in which case (7.1) and this condition form an initial value problem. Its general solution contains a single arbitrary constant of integration which can be determined from the given initial condition.

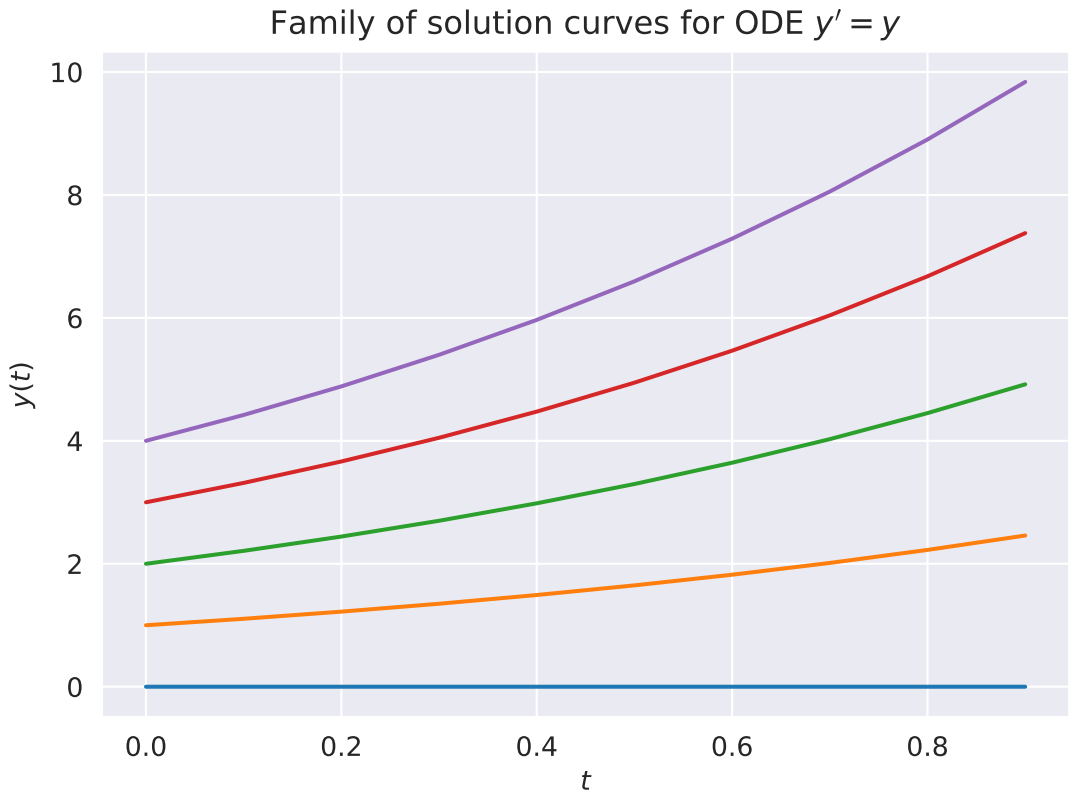
7.1.1 Stability of ODEs

Should members of the solution family of an ODE move away from each other over time, then the equation is said to be **unstable**. If the family members move closer to one another with time then the equation is said to be **stable**. Finally, if the solution curves do not approach or diverge from one another with time, then the equation is said to be **neutrally stable**. So small perturbations to a solution of a stable equation will be damped out with time since the solution curves are converging. Conversely, an unstable equation would see the perturbation grow with time as the solution curves diverge.

To give physical meaning to the above, consider a 3D cone. If the cone is stood on its circular base, then applying a perturbation to the cone will see it return to its original position standing up, implying a stable position. If the cone was balanced on its tip, then a small perturbation would see the cone fall, there the position is unstable. Finally, consider the cone resting on its side, applying a perturbation will simply roll the cone to some new position and thus the position is neutrally stable.

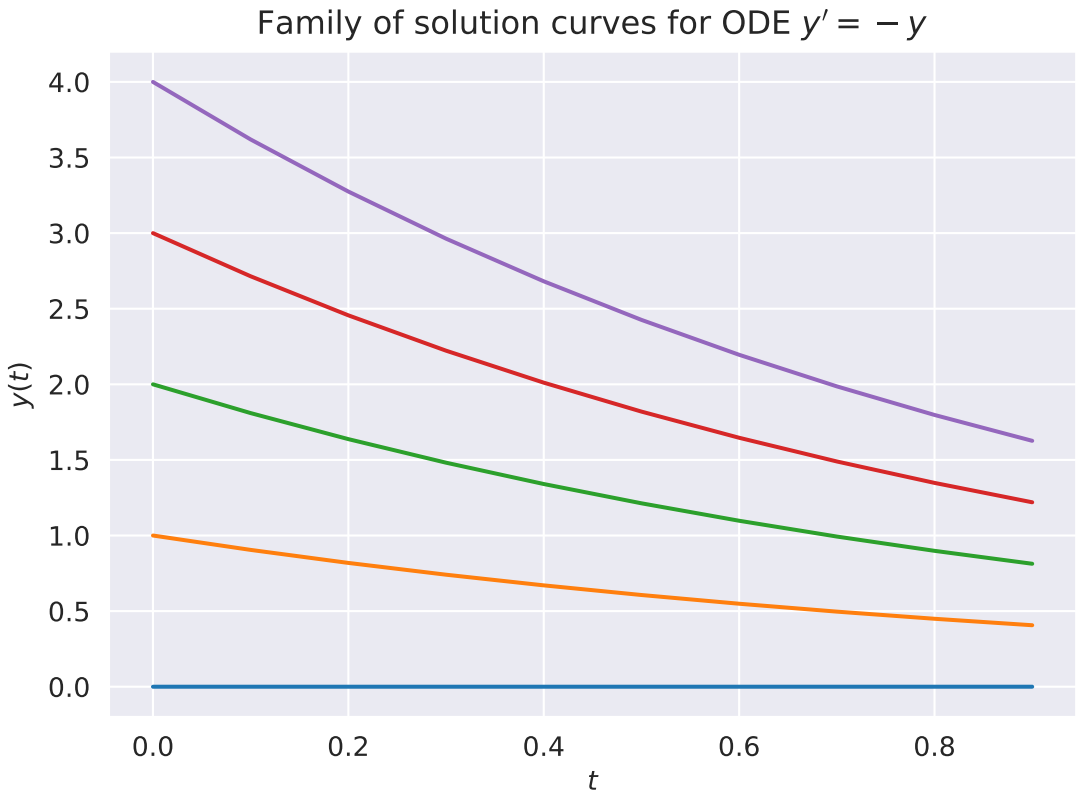
7.1.2 Unstable ODE

An example of an unstable ODE is $y' = y$. Its family of solutions are given by the curves $y(t) = ce^t$. From the exponential growth of the solutions we can see that the solution curves move away from one another as time increases implying that the equations is unstable. We can see this is the plot below.



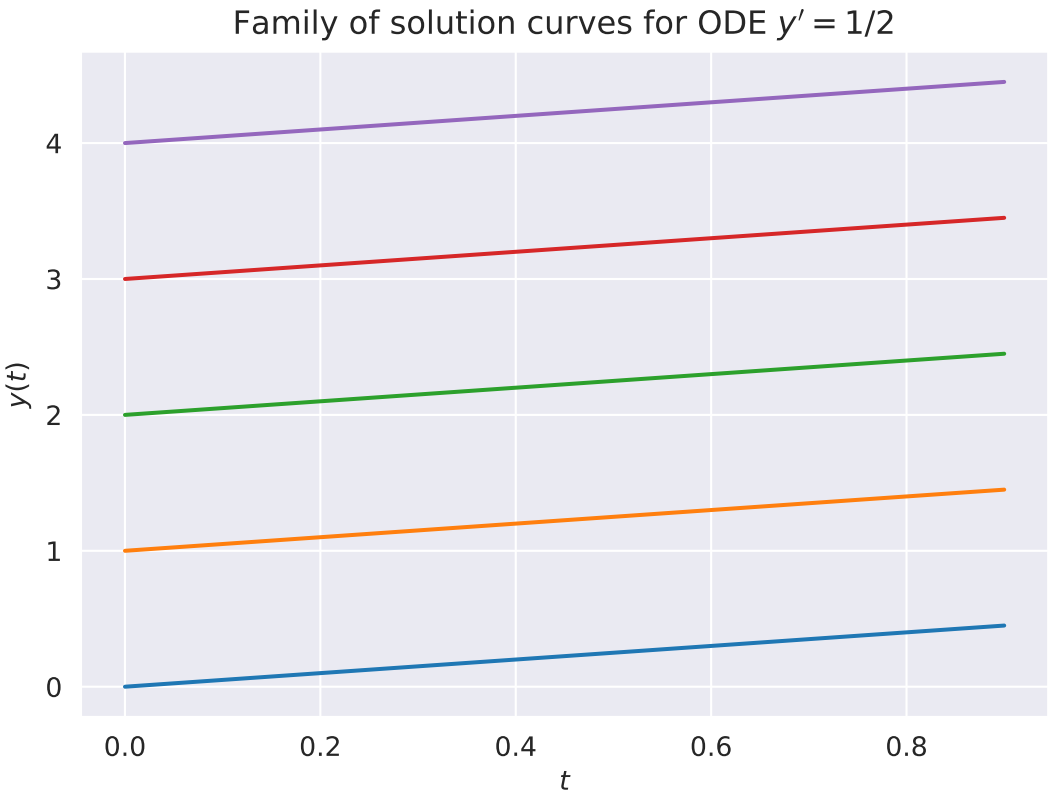
7.1.3 Stable ODE

Now consider the equation $y' = -y$. Here the family of solutions is given by $y(t) = ce^{-t}$. Since we have exponential decay of the solutions we can see that the equation is stable as seen in Figure below.



7.1.4 Neutrally Stable ODE

Finally, consider the ODE $y' = a$ for a given constant a. Here the family of solutions is given by $y(t) = at + c$, where c again is any real constant. Thus, in the example plotted below where $a = \frac{1}{2}$ the solutions are parallel straight lines which neither converge or diverge. Therefore, the equation is neutrally stable.



7.2 Euler's Method

The simplest numerical technique for solving differential equations is Euler's method. It involves choosing a suitable step size h and an initial value $y(x_0) = y_0$, which are then used to estimate $y(x_1)$, $y(x_2)$,... by a sequence of values y_i, $i = 1, 2, \ldots$. Here use the notation $x_i = x_0 + ih$.

A method of accomplishing this is suggested by the Taylor's expansion

$$y(x+h) = y(x) + hy'(x) + \frac{1}{2!}h^2 y''(x) + \frac{1}{3!}h^3 y'''(x) + \cdots$$

or, in terms of the notation introduced above:

$$y_{i+1} = y_i + hy_i' + \frac{1}{2!}h^2 y_i'' + \frac{1}{3!}h^3 y_i''' + \cdots \tag{7.2}$$

By the differential equation (7.2), we have:

$$y_i' = f(x_i, y_i)$$

which when substituted in (7.2) yields:

$$y_{i+1} = y_i + hf(x_i, y_i) + \frac{1}{2!}h^2 f'(x_i, y_i) + \frac{1}{3!}h^3 f''(x_i, y_i) + \cdots \tag{7.3}$$

and so if we truncate the Taylor series (7.3) after the term in h, we have the approximate formula:

$$y_{i+1} = y_i + hf(x_i, y_i) \tag{7.4}$$

This is a difference formula which can be evaluated step by step. This is the formula for **Euler's (or Euler-Cauchy)** method. Thus given (x_0, y_0) we can calculate (x_i, y_i) for $i = 1, 2, \cdots, n$. Since the new value y_{i+1} can be calculated from known values of x_i and y_i, this method is said to be **explicit**.

7.2.1 Error in Euler's Method

Each time we apply an equation such as(7.4) we introduce two types of errors:

- Local truncation error introduced by ignoring the terms in h^2, $h^3, \cdots$ in equation (7.2). For Euler's method, this error is
$$E = \frac{h^2}{2!} y_i''(\xi), \quad \xi \in [x_i, x_{i+1}],$$
i.e. $\epsilon_E = \mathcal{O}(h^2)$. Thus the local truncation error per step is $\mathcal{O}(h^2)$.
- A further error introduced in y_{i+1} because y_i is itself in error. The size of this error will depend on the function $f(x, y)$ and the step size h.

The above errors are introduced at each step of the calculation.

7.2.2 Example

Apply the Euler's method to solve the simple equation:

$$\frac{dy}{dx} = x + y, \quad y(0) = 1$$

(Exercise: Solve the equation analytically and show that the analytic solution is $y = 2e^x - x - 1$.)

Solution:

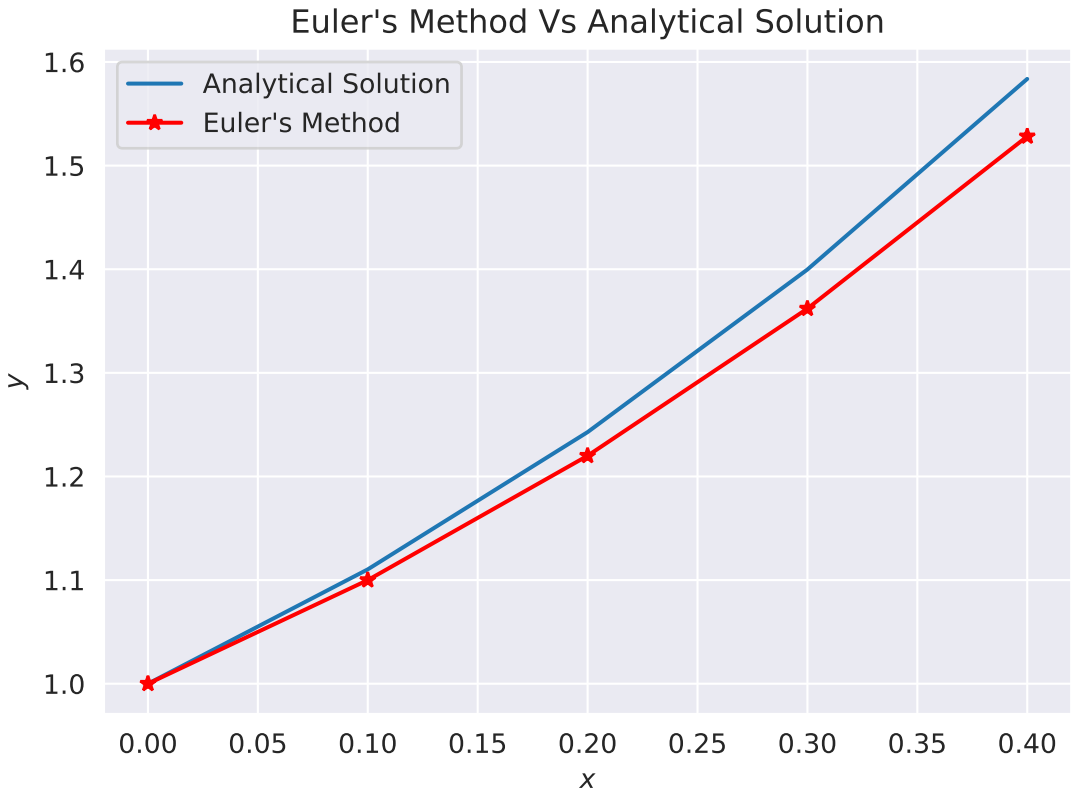
Here $f(x_i, y_i) = x_i + y_i$. With $h = 0.1$, and $y_0 = 1$ we compute y_1 as:

$$y_1 = y_0 + hf(x_0, y_0) = 1 + 0.1(0 + 1) = 1.1$$

The numerical results of approximate solutions at subsequent points $x_1 = 0.2, \ldots$ can be computed in a similar way, rounded to 3 decimal, to obtain places.

x	y	$y' = f(x, y)$	$y'h$
0	1.000	1.000	0.100
0.1	1.100	1.200	0.120
0.2	1.220	1.420	0.142
0.3	1.362	1.662	0.166
0.4	1.528	1.928	0.193

The analytical solution at $x = 0.4$ is 1.584. The numerical value is 1.528 and hence the error is about 3.5%. The accuracy of the Euler's method can be improved by using a smaller step size h. Another alternative is to use a more accurate algorithm.



7.3 Modified Euler's Method

A fundamental source of error in Euler's method is that the derivative at the beginning of the interval is assumed to apply across the entire subinterval.

There are two ways we can modify the Euler method to produce better results. One method is due to Heun (**Heun's method**) and is well documented in numerical text books. The other method we consider here is called the **improved polygon** (or **modified Euler**) method.

The modified Euler technique uses Euler's method to predict the value of y at the midpoint of the interval $[x_i, x_{i+1}]$:

$$y_{i+\frac{1}{2}} = y_i + f(x_i, y_i)\frac{h}{2}. \tag{7.5}$$

Then this predicted value is used to estimate a slope at the midpoint:

$$y'_{i+\frac{1}{2}} = f(x_{i+1/2}, y_{i+1/2}), \tag{7.6}$$

which is assumed to represent a valid approximation of the average slope for the entire subinterval. This slope is then used to extrapolate linearly from x_i to x_{i+1} using Euler's method to obtain:

$$y_{i+1} = y_i + f(x_{i+1/2}, y_{i+1/2})h \tag{7.7}$$

For the modified Euler method, the truncation error can be shown to be:

$$\epsilon_E = -\frac{h^3}{12}y'''_i(\xi), \quad \xi \in [x_i, x_{i+1}] \tag{7.8}$$

7.3.0.1 Example

Solve

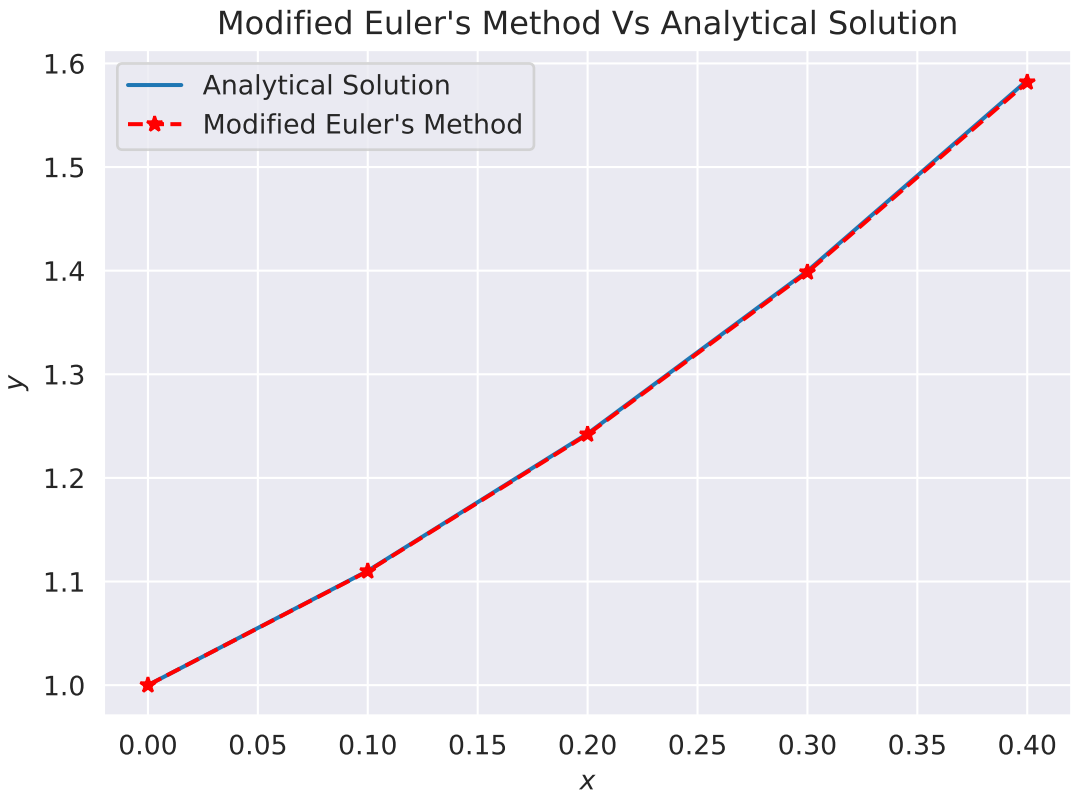
$$\frac{dy}{dx} = x + y, \quad y(0) = 1, \quad h = 0.1$$

using the modified Euler's method described above.

Solution:

x_i	y_i	$y_{i+1/2}$	$y'_{i+1/2}$	$y'_{i+1/2}h$
0	1.000	1.050	1.100	0.110
0.1	1.110	1.1705	1.3205	0.13205
0.2	1.24205	1.1705	1.3205	0.13205
0.3	1.39847	1.31415	1.56415	0.15641
0.4	1.58180	1.48339	1.83339	0.18334

The numerical solution is now 1.5818 which much more accurate that the result obtained using Euler's method. In this case the error is about 0.14%.



7.4 Runge-Kutta Methods

Runge and Kutta were German mathematicians. They suggested a group of methods for numerical solutions of ODEs.

The general form of the Runge-Kutta method is:

$$y_{i+1} = y_i + h\phi(x_i, y_i; h), \tag{7.9}$$

where $\phi(x_i, y_i; h)$ is called the increment function.

In Euler's method, $\phi(x_i, y_i; h) = f(x_i, y_i) = y_i'$, i.e we are using the slope at the point x_i to extrapolate y_i and obtain y_{i+1}. In the modified Euler's method:

$$\phi(x_i, y_i; h) = f(x_{i+\frac{1}{2}}, y_{i+\frac{1}{2}}) = y'_{i+\frac{1}{2}}$$

The increment function can be written in a general form as:

$$\phi = w_1 k_1 + w_2 k_2 + \cdots + w_n k_n \tag{7.10}$$

where the k's are constants and the w's are weights.

7.4.1 Second Order Runge-Kutta Method

The second order R-K method has the form:

$$y_{i+1} = y_i + (w_1 k_1 + w_2 k_2), \tag{7.11}$$

where

$$k_1 = hf(x_i, y_i) \tag{7.12}$$

$$k_2 = hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}), \tag{7.13}$$

and the weights $w_1 + w_2 = 1$. If $w_1 = 1$, then $w_2 = 0$ and we have Euler's method. If $w_2 = 1$, then $w_1 = 0$ we have the Euler's improved polygon method:

$$y_{i+1} = y_i + k_2 \tag{7.14}$$

$$= y_i + hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}), \tag{7.15}$$

If $w_1 = w_2 = \frac{1}{2}$, then we have:

$$y_{i+1} = y_i + \frac{1}{2}(k_1 + k_2), \tag{7.16}$$

$$k_1 = hf(x_i, y_i) \tag{7.17}$$

$$k_2 = hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}), \tag{7.18}$$

called Heun's method.

7.4.2 Fourth Order Runge-Kutta Method

The **classical fourth order R–K method** has the form:

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \tag{7.19}$$

where

$$k_1 = hf(x_i, y_i) \tag{7.20}$$

$$k_2 = hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}) \tag{7.21}$$

$$k_3 = hf(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}) \tag{7.22}$$

$$k_4 = hf(x_i + h, y_i + k_3), \tag{7.23}$$

This is the most popular R–K method. It has a local truncation error $\mathcal{O}(h^4)$

7.4.2.1 Example

Solve the DE $y' = x + y$, $y(0) = 1$ using 4^{th} order Runge–Kutta method. Compare your results with those obtained from Euler's method, modified Euler's method and the actual value. Determine $y(0.1)$ and $y(0.2)$ only.

The solution using Runge-Kutta is obtained as follows:

For y_1 :

$$k_1 = 0.1(0+1) = 0.1 \tag{7.24}$$

$$k_2 = 0.1\left(\left(0+\frac{0.1}{2}\right)+\left(1+\frac{0.1}{2}\right)\right) = 0.01 \tag{7.25}$$

$$k_3 = 0.1\left(\left(0+\frac{0.1}{2}\right)+\left(1+\frac{0.11}{2}\right)\right) = 0.1105 \tag{7.26}$$

$$k_4 = 0.1((0+0.1)+(1+0.1105)) = 0.1211 \tag{7.27}$$

and therefore:

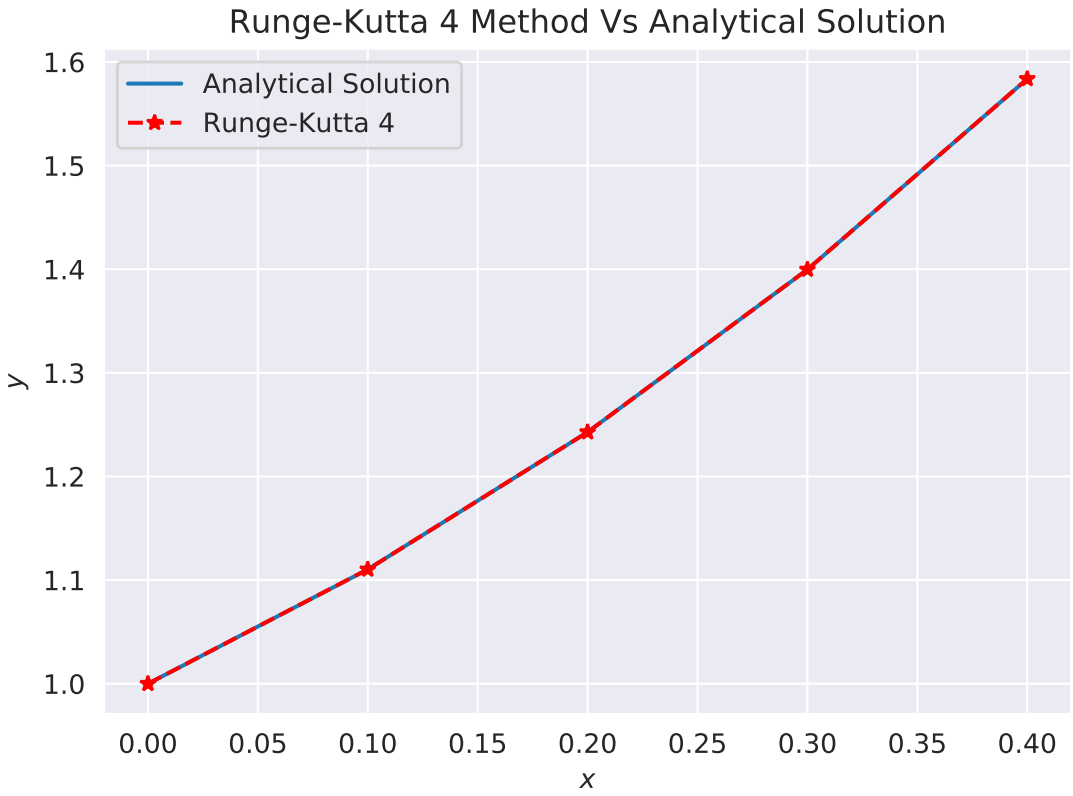
$$y_1 = y_0 + \frac{1}{6}(0.1 + 2(0.01) + 2(0.1105) + 0.1211) = 1.1103$$

A similar computation yields

$$y(0.2) = y_2 = 1.1103 + \frac{1}{6}(0.1210 + 2(0.1321) + 2(0.1326) + 0.1443 = 1.2428$$

A table for all the approximate solutions using the required methods is:

x	Euler	Modified Euler	4^{th} Order RK	Actual value
0.1	1.1000000	1.1100000	1.1103417	1.1103418
0.2	1.2300000	1.2420500	1.2428052	1.2428055



7.5 Multistep Methods

As previously, Euler's method, Modified Euler's method and Runge-Kutta methods are single-step methods. They work by computing each successive value y_{i+1} only utilising information from the preceding value y_n. Another approach are *multistep methods*, where values from several computed previously computed steps are used to obtain y_{i+1}. There are numerous methods using this approach, however, for the purpose of this course we will only consider one - the **Adam Bashforth Method**.

7.5.1 Adam-Bashforth-MoultonMethod

This is a multistep method is similar to the Modified Euler's method in that it is a predictor-corrector method, i.e. uses one formula to predict a value y'_{i+1}, which is then used to obtain a corrected value y_{i+1}. The predictor in this method is the Adams-Bashforth formula. Specifically,

$$\begin{aligned}
y^*_{i+1} &= y_i + \frac{h}{24}(55y'_i - 59y'_{i-1} + 37y'_{i-2} - 9y'_{i-3}),\\
y'_i &= f(x_i, y_i),\\
y'_{i-1} &= f(x_{i-1}, y_{i-1}),\\
y'_{i-2} &= f(x_{i-2}, y_{i-2}),\\
y'_{i-3} &= f(x_{i-3}, y_{i-3}),
\end{aligned}$$

for $i \geq 3$, which is then substituted into the Adams-Moulton corrector:

$$y_{i+1} = y_i + \frac{h}{24}(9y'_{i+1} + 19'_i - 5y'_{i-1} + y'_{i-2}) \tag{7.28}$$

$$y'_{i+1} = f(x_{i+1}, y^*_{i+1}). \tag{7.29}$$

Note that Equation (7.28) requires that we know the initial values of y_0, y_1, y_2 and y_3 in order to obtain y_4. The value y_0 is the initial condition. Since Adams-Bashforth method is of $\mathcal{O}(h^5)$, we need a high order accurate method to first obtain y_1, y_2 and y_3. Therefore, we compute these values using the RK-4 formula.

7.5.1.1 Example

Use the Adam-Bashforth method with $h = 0.2$ to obtain an approximation to $y(0.8)$ for the IVP:

$$y' = x + y - 1, \quad y(0) = 1.$$

Solution:

Using the RK-4 method to get started, we obtain the following:

$$y_1 = 1.0214, \quad y_2 = 1.09181796, \quad y_3 = 1.22210646.$$

Since $h = 0.2$, we know that $x_1 = 0.2, x_2 = 0.4, x_3 = 0.6$ and $f(x, y) = x + y - 1$. Now we can proceed:

$$\begin{aligned} y'_0 &= f(x_0, y_0) = 0 + 1 - 1 = 0, \\ y'_1 &= f(x_1, y_1) = 0.2 + 1.0214 - 1 = 0.2214, \\ y'_2 &= f(x_2, y_2) = 0.4 + 1.09181796 - 1 = 0.49181796, \\ y'_3 &= f(x_3, y_3) = 0.6 + 1.22210646 - 1 = 0.82210646. \end{aligned}$$

Now we can compute the predictor y^*_4:

$$y'_4 = y_3 + \frac{0.2}{24}(55y'_3 - 59y'_2 + 37y'_1 - 9y'_0) = 1.42535975.$$

Next, we need y'_4:

$$y'_4 = f(x_4, y^*_4) = 0.8 + 1.42535975 - 1 = 1.22535975.$$

Finally, this gives y_4 by:

$$y_4 = y_3 + \frac{0.2}{24}(9y'_4 + 19'_3 - 5y'_2 + y'_1) = 1.42552788.$$

The exact solution of this ODE at $y(0.8)$ is 1.42554093.

7.5.2 Advantages of Multistep Methods

There are a number of decisions to make when choosing a numerical method to solve a differential equation. While single step explicit methods such as RK-4 are often chosen due to their accuracy and easily programmable implementation, the right hand side of the equation needs to be evaluated many times. In the case of RK-4, the method is required to make four function evaluations at each step. On the Implicit side, if the function valuations in the previous step have been computed and stored, then a multistep method would require only one new function evaluation at each step - saving computational time.

In general the Adam-Bashforth method requires slightly more than one quarter of the number of function evaluations required for the RK-4 method.

7.6 Systems of First Order ODEs

A nth order system of first order initial value problems can be expressed in the form:

$$\begin{aligned}
\frac{dy_1}{dx} &= f_1(x, y_1, y_2, \cdots, y_n), \quad y_1(x_0) = \alpha_1 \\
\frac{dy_2}{dx} &= f_2(x, y_1, y_2, \cdots, y_n), \quad y_2(x_0) = \alpha_2 \\
&\vdots \\
\frac{dy_n}{dx} &= f_n(x, y_1, y_2, \cdots, y_n), \quad y_n(x_0) = \alpha_n,
\end{aligned}$$

for $x_0 \le x \le x_n$.

The methods we have seen so far were for a single first order equation, in which we sought the solution $y(x)$. Methods to solve first order systems of IVP are simple generalization of methods for a single equations, bearing in mind that now we seek n solutions $y_1,\ y_2, \ldots, y_n$ each with an intial condition $y_k(x_0); k = 1, \ldots, n$ at the points $x_i,\ i = 1, 2 \ldots..$

7.6.1 R-K Method for Systems

Consider the system of two equations:

$$\frac{dy}{dx} = f(x, y, z), \quad y(0) = y_0 \tag{7.30}$$

$$\frac{dz}{dx} = g(x, y, z), \quad z(0) = z_0. \tag{7.31}$$

Let $y = y_1,\ z = y_2,\ f = f_1,$ and $g = f_2$. The fourth order R-K method would be applied as follows. For each $j = 1, 2$ corresponding to solutions $y_{j,i}$, compute

$$k_{1,j} = hf_j(x_i, y_{1,i}, y_{2,i}), \ \ j = 1, 2 \tag{7.32}$$

$$k_{2,j} = hf_j(x_i + \frac{h}{2},\ y_{1,i} + \frac{k_{1,1}}{2},\ y_{2,i} + \frac{k_{1,2}}{2}) \ \ j = 1, 2 \tag{7.33}$$

$$k_{3,j} = hf_j(x_i + \frac{h}{2},\ y_{1,i} + \frac{k_{2,1}}{2},\ y_{2,i} + \frac{k_{2,2}}{2}), \ \ j = 1, 2 \tag{7.34}$$

$$k_{4,j} = hf_j(x_i + h,\ y_{1,i} + k_{3,1},\ y_{2,i} + k_{3,2}), \ \ j = 1, 2 \tag{7.35}$$

and:

$$y_{i+1} = y_{1,i+1} = y_{1,i} + \frac{1}{6}(k_{1,1} + 2k_{2,1} + 2k_{3,1} + k_{4,1}) \tag{7.36}$$

$$z_{i+1} = y_{2,i+1} = z_i + \frac{1}{6}(k_{1,2} + 2k_{2,2} + 2k_{3,2} + k_{4,2}). \tag{7.37}$$

Note that we must calculate $k_{1,1},\ k_{1,2},\ k_{2,1},\ k_{2,2},\ k_{3,1},\ k_{3,2},\ k_{4,1},\ k_{4,2}$ in that order.

7.7 Converting an n^{th} Order ODE to a System of First Order ODEs

Consider the general second order initial value problem

$$y'' + ay' + by = 0, \quad y(0) = \alpha_1, \quad y'(0) = \alpha_2$$

If we let

$$z = y', \quad z' = y''$$

then the original ODE can now be written as

$$y' = z, \quad y(0) = \alpha_1 \tag{7.38}$$
$$z' = -az - by, \quad z(0) = \alpha_2 \tag{7.39}$$

Once transformed into a system of first order ODEs the methods for systems of equations apply.

7.7.0.1 Exercise

Solve the second order differential equation:

$$y'' + 3xy' + 2x^2 y = 0, \quad y(0) = 3, \quad y'(0) = 1$$

(i) Second order R–K method (ii) 4th order R–K. Use $h = 0.1$. Do only two steps.

7.7.1 Exercises

Use (i) Euler's method (ii) modified Euler's formula to solve the following IVP;

- $y' = \sin(x + y), \quad y(0) = 0$
- $y' = yx^2 - y, \quad y(0) = 1$ for $h = 0.2$ and $h = 0.1$.
- Determine $y(0.4)$ for each of the above IVP.
- Use Richardson's extrapolation to get improved approximations to the solutions at $x = 0.4$
- If f is a function of x only, show that the fourth-order Runge-Kutta formula, applied to the differential equation $dy/dx = f(x)$ is equivalent to the use of Simpson's rule (over one interval) for evaluating $\int_0^x f(x)dx$.
- Use fourth order Runge–Kutta method to solve the following IVPs:
  - $y' = 2xy, \quad y(0) = 1$
  - $y' = 1 + y^2, \quad y(0) = 0$, Use $h = 0.2$ and determine the solutions at $x = 0.4$.
- Solve the following systems of IVPs:
  - $y' = yz, \quad z' = xz, \quad y(0) = 1, \quad z(0) = -1$
  - $y' = x - z^2, \quad z' = x + y, \quad y(0) = 1 \quad z(0) = 2$, using (i) Euler's method (ii) Second order Runge-Kutta with $h = 0.1$. Compute y and z, at $x = 0.2$.
- Use Euler's method to solve the differential equation:

  $$y' = 1 + y^2,$$

  on the domain $[0,\ 1]$ with the initial condition of (a) $y_0 = 0$ and (b) $y_0 = 1$. Plot these solutions along with the exact solution. Use step sizes of $h = 0.1$ and $h = 0.05$.
- Given the IVP $y' = (x + y - 1)^2$, $y(0) = 2$. Using the Modified Euler's method with $h = 1$ and $h = 0.05$, obtain approximate solutions of the solution at $x = 0.5$. Compare these values with the analytical solution.
