Summer Of Science
Project Report
Computational Physics

Thomas Biju Cheeramvelil (22B1073)

August 3, 2023

Contents

1 Introduction

Computational physics is a field that combines theoretical physics with computer simulations and numerical calculations to study and analyze complex physical systems. It involves developing algorithms and mathematical models to solve physics problems that are difficult or impossible to tackle analytically. I have listed the topics of computational physics which I have studied so far:

- Errors
- Numerical Integration
- Numerical Differentiation
- Systems of Linear Equations
- Systems of Non-linear Equations

2 Errors

Computation errors are of two types: round-off errors and truncation errors.

2.1 Round-off errors

In computing, a roundoff error is the difference between the result produced by a given algorithm using exact arithmetic and the result produced by the same algorithm using finite-precision, rounded arithmetic. Rounding errors are due to inexactness in the representation of real numbers and the arithmetic operations done with them. This is a form of quantization error.

2.2 Truncation errors

In computing, truncation error is an error caused by approximating a mathematical process. Tuncation errors arise when truncating an infinite sum and approximating it by a finite sum. Truncation errors arise naturally when using the Taylor series, and using finite step-size for numerical integration, and numerical differentiation.

2.3 Error propogation

Suppose we want to compute $f(x)$ where x is a real number and f is a real function. In practical computations, the number x must be approximated by a rational number r since no computer can store numbers with an infinite number of decimals. The difference $|r - x|$ constitutes the initial error while the difference $\epsilon_0 = |f(r) - f(x)|$ is the corresponding propagated error. In many cases, f is such a function that it must be replaced by a simpler function f_1 (often a truncated power series expansion of f). The difference $\epsilon_1 = |f_1(r) - f(r)|$ is then the truncation error. The calculations performed by the computer, however, are not exact. The result is that instead of getting $f_1(r)$ we get another value $f_2(r)$ which is then a wrongly computed value of a wrong function of a wrong argument. The difference $\epsilon_2 = |f_2(r) - f_1(r)|$ could be termed the propagated error from the roundings. The total error is $\epsilon = \epsilon_0 + \epsilon_1 + \epsilon_2$.

2.4 Error estimation in iterative methods

If the value x is obtained using an iterative method, i.e., each iteration n would produce the approximate value x_n, then, the relative approximate error is defined as:

$$\varepsilon_r = \frac{x_n - x_{n-1}}{x_n}$$

In such cases, the iterative method can be stopped when the absolute value of the relative error reaches a specified error level ε_s:

$$|\varepsilon_r| \leq \varepsilon_s$$

3 Numerical integration

3.1 Introduction

Numerical integration is a computational technique used to approximate the definite integral of a function. It involves dividing the interval of integration into smaller subintervals and approximating the integral within each subinterval using numerical methods. Numerical integration finds applications in various fields, including physics, engineering, economics, and computer science. It is used to calculate the trajectories of a particles, and solve differential equations that arise in areas such as control systems. It also finds applications in financial modeling and image processing.

3.2 Trapezoidal rule

Let $f : [a, b] \to \mathbb{R}$. By dividing the interval $[a, b]$ into many subintervals, the trapezoidal rule approximates the area under the curve by linearly interpolating between the values of the function at the junctions of the subintervals, and thus, on each subinterval, the area to be calculated has a shape of a trapezoid. For simplicity, the width of the trapezoids is chosen to be constant. Let n be the number of intervals with $a = x_0 < x_1 < x_2 < \cdots < x_n = b$ and constant spacing $h = x_{i+1} - x_i$. The trapezoidal method can be implemented as follows:

$$I_T = \int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{2}\sum_{i=1}^{n}(f(x_{i-1}) + f(x_i)) = \frac{h}{2}(f(x_0)+2f(x_1)+2f(x_2)+\cdots+2f(x_{n-1})+f(x_n))$$

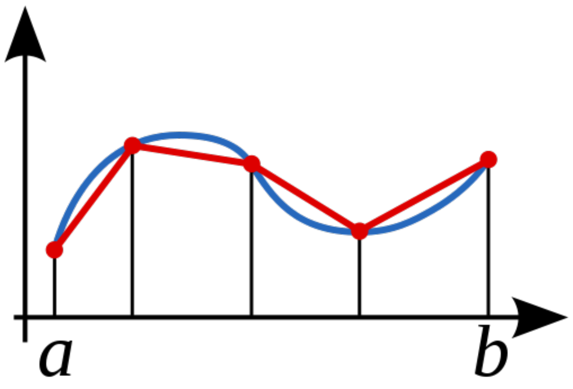

Figure 1: Trapezoidal rule

Error analysis: An extension of Taylor's theorem can be used to find how the error changes as the step size h decreases. Given a function f and its interpolating polynomial of degree n $p_n(x)$, the error term between the interpolating polynomial and the function is given by:

$$f(x) = p_n(x) + \frac{f^{n+1}(\xi)}{(n+1)!}\prod_{i=1}^{n+1}(x - x_i)$$

Where ξ is in the domain of the function f and is dependent on the point x. The error in the calculation of trapezoidal number i between the points x_{i-1} and x_i will be estimated based on the above formula assuming a linear interpolation between the points x_{i-1} and x_i. Therefore:

$$\begin{aligned}|E_i| &= \left|\int_{x_{i-1}}^{x_i} f(x) - p_1(x)\,\mathrm{d}x\right| = \left|\int_{x_{i-1}}^{x_i} \frac{f''(\xi)}{2}(x-x_{i-1})(x-x_i)\,\mathrm{d}x\right| \\ &\leq \max_{\xi\in[x_{i-1},x_i]} \frac{|f''(\xi)|h^3}{12}\end{aligned}$$

If n is the number of subdivisions (number of trapezoids), i.e., $nh = b-a$, then:

$$|E| = |nE_i| \leq \max_{\xi\in[a,b]} \frac{|f''(\xi)|nh^3}{12} = \max_{\xi\in[a,b]} \frac{|f''(\xi)|(b-a)h^2}{12}$$

In the above error analysis, we disregarded higher order terms than f". Although this simplified the calculations and sufficed our purpose, here we demonstrate a general error analysis containing higher order terms. We will use the results later in explaining the Romberg's Method. Consider a continuous function $f(x)$ defined on an interval $[a,b]$. If the interval is discretized into n sub intervals $[x_i, x_{i+1}]$ such that $a = x_0 < x_1 < \cdots < x_n = b$, the trapezoidal rule estimates the integration of $f(x)$ over a sub interval $[x_i, x_{i+1}]$ as:

$$\int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x \approx \frac{h_i}{2}\left(f(x_i)+f(x_{i+1})\right)$$

where $h_i = x_{i+1} - x_i$ To find the order of error of the above estimation, we should find the terms represented by $\mathcal{O}(h_i^k)$ in:

$$\int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x = \frac{h_i}{2}\left(f(x_i)+f(x_{i+1})\right) + \mathcal{O}(h_i^k)$$

where k in the order of accuracy (to be determined). To this end, we first consider writing $f(x)$ for $x \in [x_i, x_{i+1}]$ by its the Taylor series expansion about $y_i = \frac{x_i+x_{i+1}}{2}$ being is the midpoint of the interval $[x_i, x_{i+1}]$. Therefore, substituting the Taylor series expansion into $\int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x$ leads to,

$$\begin{aligned}\int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x &= \int_{x_i}^{x_{i+1}} \left(f(y_i) + (x-y_i)f'(y_i) + \frac{1}{2}(x-y_i)^2 f''(y_i) + \frac{1}{6}(x-y_i)^3 f'''(y_i) + \dots\right)\mathrm{d}x \\ &= h_i f(y_i) + \frac{1}{2}(x-y_i)^2\Big|_{x_i}^{x_{i+1}} f'(y_i) + \frac{1}{6}(x-y_i)^3\Big|_{x_i}^{x_{i+1}} f''(y_i) \\ &+ \frac{1}{24}(x-y_i)^4\Big|_{x_i}^{x_{i+1}} f''(y_i) + \cdots\end{aligned}$$

which is eventually evaluated as,

$$\int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x = h_i f(y_i) + \frac{1}{24}h_i^3 f''(y_i) + \frac{1}{1920}h_i^5 f^{(4)}(y_i) + \cdots$$

Leaving this result for now and getting back to the trapezoidal rule, we can write $\frac{f(x_i)+f(x_{i+1})}{2}$ using its Taylor series expansion as follows. If the Taylor series expansions for $f(x_i)$ and $f(x_{i+1})$ are written as,

$$\begin{aligned}f(x_i) &= f(y_i) - \frac{1}{2}h_i f'(y_i) + \frac{1}{8}h_i^2 f''(y_i) - \frac{1}{48}h_i^3 f'''(y_i) + \cdots \\ f(x_{i+1}) &= f(y_i) + \frac{1}{2}h_i f'(y_i) + \frac{1}{8}h_i^2 f''(y_i) + \frac{1}{48}h_i^3 f'''(y_i) + \cdots\end{aligned}$$

therefore,

$$\frac{f(x_i)+f(x_{i+1})}{2} = f(y_i) + \frac{1}{8}h_i^2 f''(y_i) + \frac{1}{384}h_i^4 f^{(4)}(y_i) + \cdots$$

Note that the derivatives of odd order, i.e. $f', f''', f^{(5)}, \dots$ vanish. Solving the above equation for $f(y_i)$ and substituting the resultant expression into the right-hand side of Eq. II, leads to,

$$\int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x = h_i \frac{f(x_i) + f(x_{i+1})}{2} - \frac{1}{12} h_i^3 f''(y_i) - \frac{1}{480} h_i^5 f^{(4)}(y_i) + \cdots$$

Comparing this result with Eq. I, we can say that the trapezoidal rule is third-order accurate over a sub-interval, in other words,

$$\int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x = \frac{h_i}{2} \left(f(x_i) + f(x_{i+1})\right) + \mathcal{O}(h_i^3)$$

We should now proceed to analyze the error of the trapezoidal rule over the entire interval $[a, b]$. If we consider a similar length, h, for each sub interval $[x_i, x_{i+1}]$ of $[a, b]$ discretized as $a = x_0 < x_1 < \cdots < x_n = b$, we can write,

$$\int_a^b f(x)\mathrm{d}x = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x = \frac{h}{2}\Big(f(a) + f(b) + 2\sum_{j=1}^{n-1} f(x_j)\Big) - \frac{h^3}{12}\sum_{i=0}^{n-1} f''(y_i) - \frac{h^5}{480}\sum_{i=0}^{n-1} f^{(4)}(y_i) + \cdots$$

The summations of terms of derivatives can be replaced with their values determined using the intermediate value theorem. The intermediate value theorem states that if a function $g(x)$ is continuous over the interval $[a_1, a_2]$, there is $c \in [a_1, a_2]$ such that $g(s) = s$ for any s falling within the values of $g(a_1)$ and $g(a_2)$. Therefore, for the continuous function $f''(x)$ over $[a, b]$, if we set a_1 and a_2 within $[a, b]$ such that $f''(a_1) = \min\limits_{x\in[a,b]} f''(y) and f''(a_2) = \max\limits_{x\in[a,b]} f''(y)$, we can write,

$$f''(a_1) \leq \frac{\sum_{i=0}^{n-1} f''(y_i)}{n} \leq f''(a_2)$$

meaning that there is a number $\xi \in [a, b]$ such that $f''(\xi) = \frac{\sum_{i=0}^{n-1} f''(y_i)}{n}$. Consequently,

$$\sum_{i=0}^{n-1} f''(y_i) = nf''(\xi)$$

With the same method, we can write $\sum_{i=0}^{n-1} f^{(4)}(y_i)$ as,

$$\sum_{i=0}^{n-1} f^{(4)}(y_i) = nf^{(4)}(\eta)$$

for $\eta \in [a, b]$. Using the above values of the summations we get,

$$\int_a^b f(x)\mathrm{d}x = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x = \frac{h}{2}\Big(f(a) + f(b) + 2\sum_{j=1}^{n-1} f(x_j)\Big) - \frac{h^3}{12} nf''(\xi) - \frac{h^5}{480} nf^{(4)}(\eta) + \cdots$$

As $n = \frac{b-a}{h}$, we can conclude that

$$\int_a^b f(x)\mathrm{d}x = \frac{h}{2}\Big(f(a) + f(b) + 2\sum_{j=1}^{n-1} f(x_j)\Big) - (b-a)\frac{h^2}{12} f''(\xi) - (b-a)\frac{h^4}{480} f^{(4)}(\eta) + \cdots$$

Indicating that the trapezoidal rule for integration of $f(x)$ over any interval $[a, b]$ is second-order accurate.

3.3 Simpson's $\frac{1}{3}$ rule

Let $f : [a, b] \to \mathbb{R}$. By dividing the interval $[a, b]$ into many subintervals, the Simpson's 1/3 rule approximates the area under the curve in every subinterval by interpolating between the values of the function at the midpoint and ends of the subinterval, and thus, on each subinterval, the curve to be integrated is a parabola. For simplicity, the width of each subinterval is chosen to be constant and is equal to $2h$. Let n be the number of intervals with $a = x_0 < x_1 < x_2 < \cdots < x_n = b$ and constant spacing $2h = x_i - x_{i-1}$. On each interval with end points x_{i-1} and x_i, Lagrange polynomials can be used to define the interpolating parabola as follows:

$$p_2(x) = f(x_{i-1})\frac{(x - x_{mi})(x - x_i)}{2h^2} + f(x_{mi})\frac{(x - x_{i-1})(x - x_i)}{-h^2} + f(x_i)\frac{(x - x_{i-1})(x - x_{mi})}{2h^2}$$

where x_{mi} is the midpoint in the interval i. Integrating the above formula yields:

$$\int_{x_{i-1}}^{x_i} p_2(x)\,\mathrm{d}x = \frac{h}{3}\left(f(x_{i-1}) + 4f(x_{mi}) + f(x_i)\right)$$

The Simpson's 1/3 rule can be implemented as follows:

$$\begin{aligned} I_{S1} &= \int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{3}\sum_{i=1}^{n}\left(f(x_{i-1}) + 4f(x_{mi}) + f(x_i)\right) \\ &= \frac{h}{3}(f(x_0) + 4f(x_{m1}) + 2f(x_1) + 4f(x_{m2}) + 2f(x_2) + \cdots + 2f(x_{n-1}) + 4f(x_{mn}) + f(x_n)) \end{aligned}$$

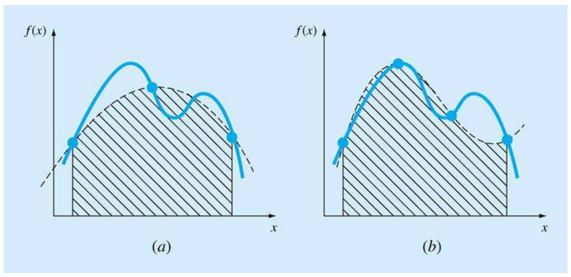

Figure 2: Simpson's 1/3 and 3/8 rules

Error analysis:
Given a function f and its interpolating polynomial of degree n $p_n(x)$, the error term between the interpolating polynomial and the function is given by:

$$f(x) = p_n(x) + \frac{f^{n+1}(\xi)}{(n+1)!}\prod_{i=1}^{n+1}(x - x_i)$$

Where ξ is in the domain of the function f. The error in the calculation of the integral of the parabola number i connecting the points x_{i-1}, $x_m = \frac{x_{i-1}+x_i}{2}$, and x_i will be estimated based on the above formula assuming $2h = x_i - x_{i-1}$. Therefore:

$$|E_i| = \left|\int_{x_{i-1}}^{x_i} f(x) - p_2(x)\,\mathrm{d}x\right| = \left|\int_{x_{i-1}}^{x_i} \frac{f'''(\xi)}{3 \times 2}(x - x_{i-1})(x - x_m)(x - x_i)\,\mathrm{d}x\right|$$

Therefore, the upper bound for the error can be given by:

$$|E_i| \leq \max_{\xi\in[x_{i-1},x_i]} \frac{|f'''(\xi)|}{6}\int_{x_{i-1}}^{x_i} |(x - x_{i-1})(x - x_m)(x - x_i)|\,\mathrm{d}x = \max_{\xi\in[x_{i-1},x_i]} \frac{|f''(\xi)|h^4}{12}$$

If n is the number of subdivisions, where each subdivision has width of $2h$, i.e., $n(2h) = b - a$, then:

$$|E| = |nE_i| \leq \max_{\xi\in[a,b]} \frac{|f'''(\xi)|nh^4}{12} = \max_{\xi\in[a,b]} \frac{|f'''(\xi)|(b-a)h^3}{24}$$

However, it can actually be shown that there is a better estimate for the upper bound of the error. This can be shown using Newton interpolating polynomials through the points $x_{i-1}, x_m, x_i, x_i + h$. The reason we add an extra point is going to become apparent when the integration is carried out. The error term between the interpolating polynomial and the function is given by:

$$\begin{aligned} f(x) =& b_1 + b_2(x - x_{i-1}) + b_3(x - x_{i-1})(x - x_m) + b_4(x - x_{i-1})(x - x_m)(x - x_i) \\ &+ \frac{f''''(\xi)}{4!}(x - x_{i-1})(x - x_m)(x - x_i)(x - x_i - h) \end{aligned}$$

where $\xi \in [x_{i-1}, x_i + h]$ and is dependent on x. The first three terms on the right-hand side are exactly the interpolating parabola passing through the points $f(x_{i-1})$, $f(x_m)$, and $f(x_i)$. Therefore, an estimate for the error can be evaluated as:

$$\begin{aligned} |E_i| =& \left| \int_{x_{i-1}}^{x_i} f(x) - b_1 - b_2(x - x_{i-1}) - b_3(x - x_{i-1})(x - x_m) \,\mathrm{d}x \right| \\ \leq& \left| \int_{x_{i-1}}^{x_i} b_4(x - x_{i-1})(x - x_m)(x - x_i) \,\mathrm{d}x \right| \\ &+ \max_{\xi\in[x_{i-1},x_i+h]} \left| \frac{f''''(\xi)}{4!} \right| \int_{x_{i-1}}^{x_i} |(x - x_{i-1})(x - x_m)(x - x_i)(x - x_i - h)| \,\mathrm{d}x \\ \leq& 0 + \max_{\xi\in[x_{i-1},x_i+h]} \left| \frac{f''''(\xi)}{4!} \right| \frac{4h^5}{15} \\ \leq& \max_{\xi\in[x_{i-1},x_i+h]} |f''''(\xi)| \frac{h^5}{90} \end{aligned}$$

The first term on the right-hand side of the inequality is equal to zero. This is because the point x_m is the average of x_{i-1} and x_i, so, integrating that cubic polynomial term yields zero. This was the reason to consider a third-order polynomial instead of a second-order polynomial which allows the error term to be in terms of h^5. If n is the number of subdivisions, where each subdivision has width of $2h$, i.e., $n(2h) = b - a$, then:

$$|E| = |nE_i| \leq \max_{\xi\in[a,b]} \frac{|f''''(\xi)|nh^5}{90} = \max_{\xi\in[a,b]} \frac{|f''''(\xi)|(b-a)h^4}{180}$$

3.4 Simpson's $\frac{3}{8}$ rule

Let $f : [a, b] \to \mathbb{R}$. By dividing the interval $[a, b]$ into many subintervals, the Simpson's 3/8 rule approximates the area under the curve in every subinterval by interpolating between the values of the function at the ends of the subinterval and at two intermediate points, and thus, on each subinterval, the curve to be integrated is a cube. For simplicity, the width of each subinterval is chosen to be constant and is equal to $3h$. Let n be the number of intervals with $a = x_0 < x_1 < x_2 < \cdots < x_n = b$ and constant spacing $3h = x_i - x_{i-1}$ with the intermediate points for each interval i as $x_{li} = x_{i-1} + h$ and $x_{ri} = x_{i-1} + 2h$. On each interval with end points x_{i-1} and x_i, Lagrange polynomials can be used to define the interpolating cubic polynomial as follows:

$$\begin{aligned} p_3(x) =& f(x_{i-1})\frac{(x - x_{li})(x - x_{ri})(x - x_i)}{-6h^3} + f(x_{li})\frac{(x - x_{i-1})(x - x_{ri})(x - x_i)}{2h^3} \\ &+ f(x_{ri})\frac{(x - x_{i-1})(x - x_{li})(x - x_i)}{-2h^3} + f(x_i)\frac{(x - x_{i-1})(x - x_{li})(x - x_{ri})}{6h^3} \end{aligned}$$

Integrating the above formula yields:

$$\int_{x_{i-1}}^{x_i} p_3(x)\,\mathrm{d}x = \frac{3h}{8}\left(f(x_{i-1}) + 3f(x_{li}) + 3f(x_{ri}) + f(x_i)\right)$$

The Simpson's 3/8 rule can be implemented as follows:

$$\begin{aligned} I_{S2} &= \int_a^b f(x)\,\mathrm{d}x \approx \frac{3h}{8}\sum_{i=1}^{n}\left(f(x_{i-1}) + 3f(x_{li}) + 3f(x_{ri}) + f(x_i)\right) \\ &= \frac{3h}{8}(f(x_0) + 3f(x_{l1}) + 3f(x_{r1}) + 2f(x_1) + \cdots + 3f(x_{rn}) + f(x_n)) \end{aligned}$$

Error analysis:
The estimate for the upper bound of the error can be derived similar to the derivation of the upper bound of the error in the trapezoidal rule as follows. Given a function f and its interpolating polynomial of degree n $p_n(x)$, the error term between the interpolating polynomial and the function is given by :

$$f(x) = p_n(x) + \frac{f^{n+1}(\xi)}{(n+1)!}\prod_{i=1}^{n+1}(x - x_i)$$

Where ξ is in the domain of the function f. The error in the calculation of the integral of the cubic function number i connecting the points x_{i-1}, $x_{li} = x_{i-1} + h$, $x_{ri} = x_{i-1} + 2h$, and $x_i = x_{i-1} + 3h$ will be estimated based on the above formula assuming $3h = x_i - x_{i-1}$. Therefore:

$$|E_i| = \left|\int_{x_{i-1}}^{x_i} f(x) - p_3(x)\,\mathrm{d}x\right| = \left|\int_{x_{i-1}}^{x_i} \frac{f''''(\xi)}{4\times 3\times 2}(x - x_{i-1})(x - x_{li})(x - x_{ri})(x - x_i)\,\mathrm{d}x\right|$$

Therefore, the upper bound for the error can be given by:

$$|E_i| \le \max_{\xi\in[x_{i-1},x_i]} \frac{|f''''(\xi)|}{4\times 3\times 2}\int_{x_{i-1}}^{x_i} |(x - x_{i-1})(x - x_{li})(x - x_{ri})(x - x_i)|\,\mathrm{d}x = \max_{\xi\in[x_{i-1},x_i]} \frac{|f''''(\xi)|3h^5}{80}$$

If n is the number of subdivisions, where each subdivision has width of $3h$, i.e., $n(3h) = b - a$, then:

$$|E| = |nE_i| \le \max_{\xi\in[a,b]} \frac{|f''''(\xi)|3nh^5}{80} = \max_{\xi\in[a,b]} \frac{|f''''(\xi)|(b-a)h^4}{80}$$

4 Numerical differentiation

4.1 Introduction

Let f: $\mathbb{R} \to \mathbb{R}$ be a smooth differentiable function, then the derivative of f at x is defined as the limit:

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

From a geometric perspective, the derivative of the function at a point x gives the slope of the tangent to the function at that point.
Numerical differentiation finds extensive applications in various fields. In physics and engineering, it aids in solving differential equations numerically, analyzing motion and dynamics, and estimating rates of change. In optimization algorithms, numerical differentiation helps determine the direction of steepest descent and improve convergence. It is also employed in data analysis, signal processing, and image processing to extract important information from experimental or noisy data.

4.2 Methods for First Order Derivatives

- **Forward Finite Difference:**
  Let $f : [a, b] \to \mathbb{R}$ be differentiable and let $a \leq x_i < x_{i+1} \leq b$, then, using Taylor theorem:
  $$f(x_{i+1}) = f(x_i) + f'(x_i)h + \mathcal{O}(h^2)$$
  where $h = x_{i+1} - x_i$. In that case, the forward finite-difference can be used to approximate $f'(x_i)$ as follows:
  $$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} + \mathcal{O}(h) \tag{1}$$
  where $\mathcal{O}(h)$ indicates that the error term is directly proportional to the chosen step size h.

- **Backward Finite Difference:**
  Let $f : [a, b] \to \mathbb{R}$ be differentiable and let $a \leq x_{i-1} < x_i \leq b$, then, using Taylor theorem:
  $$f(x_{i-1}) = f(x_i) - f'(x_i)h + \mathcal{O}(h^2)$$
  where $h = x_i - x_{i-1}$. In this case, the backward finite-difference can be used to approximate $f'(x_i)$ as follows:
  $$f'(x_i) = \frac{f(x_{i-1}) - f(x_i)}{x_{i-1} - x_i} + \mathcal{O}(h)$$
  where $\mathcal{O}(h)$ indicates that the error term is directly proportional to the chosen step size h.

- **Centred Finite Difference:**
  The centred finite difference can provide a better estimate for the derivative of a function at a particular point. If the values of a function f are known at the points $x_{i-1} < x_i < x_{i+1}$ and $x_{i+1} - x_i = x_i - x_{i-1} = h$, then, we can use the Taylor series to find a good approximation for the derivative as follows:
  $$\begin{aligned} f(x_{i+1}) &= f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \mathcal{O}(h^3) \\ f(x_{i-1}) &= f(x_i) + f'(x_i)(-h) + \frac{f''(x_i)}{2!}h^2 + \mathcal{O}(h^3) \end{aligned}$$
  subtracting the above two equations and dividing by h gives the following:
  $$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} + \mathcal{O}(h^2)$$
  where $\mathcal{O}(h^2)$ indicates that the error term is directly proportional to the square of the chosen step size h. I.e., the centred finite difference provides a better estimate for the derivative when the step size h is reduced compared to the forward and backward finite differences. Notice that when $x_{i+1} - x_i = x_i - x_{i-1}$, the centred finite difference is the average of the forward and backward finite difference!

4.3 Extending to Higher Order Derivatives

- **Forward Finite Difference:**
  Let $f : [a, b] \to \mathbb{R}$ be differentiable and let $a \leq x_i < x_{i+1} < x_{i+2} \leq b$, with $h = x_{i+2} - x_{i+1} = x_{i+1} - x_i$, then, using the basic forward finite difference formula for the second derivative, we have:
  $$\begin{aligned} f''(x_i) &= \frac{f'(x_{i+1}) - f'(x_i)}{h} + \mathcal{O}(h) \\ &= \frac{\frac{f(x_{i+2}) - f(x_{i+1})}{h} - \frac{f(x_{i+1}) - f(x_i)}{h}}{h} + \mathcal{O}(h) \\ &= \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2} + \mathcal{O}(h) \end{aligned}$$

Notice that in order to calculate the second derivative at a point x_i using forward finite difference, the values of the function at two additional points x_{i+2} and x_{i+1} are needed. Similarly, for the third derivative, the value of the function at another point x_{i+3} with $x_{i+2} < x_{i+3} \leq b$ is required (with the same spacing h). Then, the third derivative can be calculated as follows:

$$\begin{aligned} f'''(x_i) &= \frac{f''(x_{i+1}) - f''(x_i)}{h} + \mathcal{O}(h) \\ &= \frac{\frac{f(x_{i+3})-2f(x_{i+2})+f(x_{i+1})}{h^2} - \frac{f(x_{i+2})-2f(x_{i+1})+f(x_i)}{h^2}}{h} + \mathcal{O}(h) \\ &= \frac{f(x_{i+3}) - 3f(x_{i+2}) + 3f(x_{i+1}) - f(x_i)}{h^3} + \mathcal{O}(h) \end{aligned}$$

Similarly, for the fourth derivative, the value of the function at another point x_{i+4} with $x_{i+3} < x_{i+4} \leq b$ is required (with the same spacing h). Then, the fourth derivative can be calculated as follows:

$$\begin{aligned} f''''(x_i) &= \frac{f'''(x_{i+1}) - f'''(x_i)}{h} + \mathcal{O}(h) \\ &= \frac{\frac{f(x_{i+4})-3f(x_{i+3})+3f(x_{i+2})-f(x_{i+1})}{h^3} - \frac{f(x_{i+3})-3f(x_{i+2})+3f(x_{i+1})-f(x_i)}{h^3}}{h} + \mathcal{O}(h) \\ &= \frac{f(x_{i+4}) - 4f(x_{i+3}) + 6f(x_{i+2}) - 4f(x_{i+1}) + f(x_i)}{h^4} + \mathcal{O}(h) \end{aligned}$$

- **Backward Finite Difference:**
Let $f : [a, b] \to \mathbb{R}$ be differentiable and let $a \leq x_{i-2} < x_{i-1} < x_i \leq b$, with $h = x_i - x_{i-1} = x_{i-1} - x_{i-2}$, then, using the basic backward finite difference formula for the second derivative, we have:

$$\begin{aligned} f''(x_i) &= \frac{f'(x_i) - f'(x_{i-1})}{h} + \mathcal{O}(h) \\ &= \frac{\frac{f(x_i)-f(x_{i-1})}{h} - \frac{f(x_{i-1})-f(x_{i-2})}{h}}{h} + \mathcal{O}(h) \\ &= \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{h^2} + \mathcal{O}(h) \end{aligned}$$

Notice that in order to calculate the second derivative at a point x_i using backward finite difference, the values of the function at two additional points x_{i-2} and x_{i-1} are needed. Similarly, for the third derivative, the value of the function at another point x_{i-3} with $a \leq x_{i-3} < x_{i-2}$ is required (with the same spacing h). Then, the third derivative can be calculated as follows:

$$\begin{aligned} f'''(x_i) &= \frac{f''(x_i) - f''(x_{i-1})}{h} + \mathcal{O}(h) \\ &= \frac{\frac{f(x_i)-2f(x_{i-1})+f(x_{i-2})}{h^2} - \frac{f(x_{i-1})-2f(x_{i-2})+f(x_{i-3})}{h^2}}{h} + \mathcal{O}(h) \\ &= \frac{f(x_i) - 3f(x_{i-1}) + 3f(x_{i-2}) - f(x_{i-3})}{h^3} + \mathcal{O}(h) \end{aligned}$$

Similarly, for the fourth derivative, the value of the function at another point x_{i-4} with $a \leq x_{i-4} < x_{i-3}$ is required (with the same spacing h). Then, the fourth derivative can be calculated as follows:

$$\begin{aligned}
f''''(x_i) &= \frac{f'''(x_i) - f'''(x_{i-1})}{h} + \mathcal{O}(h) \\
&= \frac{\frac{f(x_i)-3f(x_{i-1})+3f(x_{i-2})-f(x_{i-3})}{h^3} - \frac{f(x_{i-1})-3f(x_{i-2})+3f(x_{i-3})-f(x_{i-4})}{h^3}}{h} + \mathcal{O}(h) \\
&= \frac{f(x_i) - 4f(x_{i-1}) + 6f(x_{i-2}) - 4f(x_{i-3}) + f(x_{i-4})}{h^4} + \mathcal{O}(h)
\end{aligned}$$

- **Centred Finite Difference:**
  Let $f : [a, b] \to \mathbb{R}$ be differentiable and let $a \leq x_{i-1} < x_i < x_{i+1} \leq b$, with a constant spacing h, then, we can use the Taylor theorem for $f(x_{i+1})$ and $f(x_{i-1})$ as follows:

$$\begin{aligned}
f(x_{i+1}) &= f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(x_i)}{3!}h^3 + \mathcal{O}(h^4) \\
f(x_{i-1}) &= f(x_i) + f'(x_i)(-h) + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(x_i)}{3!}(-h)^3 + \mathcal{O}(h^4)
\end{aligned}$$

Adding the above two equations and dividing by h^2 gives the following:

$$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2} + \mathcal{O}(h^2)$$

which provides a better approximation for the second derivative than that provided by the forward or backward finite difference as the error is directly proportional to the square of the step size. For the third derivative, the value of the function is required at the points x_{i-2} and x_{i+2}. Assuming all the points to be equidistant with a spacing h, then, the third derivative can be calculated as follows:

$$f'''(x_i) = \frac{f'(x_{i+1}) - 2f'(x_i) + f'(x_{i-1})}{h^2} + \mathcal{O}(h^2)$$

Replacing the first derivatives with the centred finite difference value for those:

$$\begin{aligned}
f'''(x_i) &= \frac{\frac{f(x_{i+2})-f(x_i)}{2h} - 2\frac{f(x_{i+1})-f(x_{i-1})}{2h} + \frac{f(x_i)-f(x_{i-2})}{2h}}{h^2} + \mathcal{O}(h^2) \\
&= \frac{f(x_{i+2}) - 2f(x_{i+1}) + 2f(x_{i-1}) - f(x_{i-2})}{2h^3} + \mathcal{O}(h^2)
\end{aligned}$$

For the fourth derivative, the value of the function at the points x_{i+2} and x_{i-2} is required. Assuming all the points to be equidistant with a spacing h, then, the fourth derivative can be calculated as follows:

$$f''''(x_i) = \frac{f''(x_{i+1}) - 2f''(x_i) + f''(x_{i-1})}{h^2} + \mathcal{O}(h^2)$$

Using the centred finite difference for the second derivatives yields:

$$\begin{aligned}
f''''(x_i) &= \frac{\frac{f(x_{i+2})-2f(x_{i+1})+f(x_i)}{h^2} - 2\frac{f(x_{i+1})-2f(x_i)+f(x_{i-1})}{h^2} + \frac{f(x_i)-2f(x_{i-1})+f(x_{i-2})}{h^2}}{h^2} + \mathcal{O}(h^2) \\
&= \frac{f(x_{i+2}) - 4f(x_{i+1}) + 6f(x_i) - 4f(x_{i-1}) + f(x_{i-2})}{h^4} + \mathcal{O}(h^2)
\end{aligned}$$

5 Systems of Linear Equations

5.1 Introduction

Let $A \in \mathbb{R}^{mxn}$ and given a vector $b \in \mathbb{R}^n$, a linear system of equations seeks the solution x to the equation:

$$Ax = b$$

In a matrix component form the above equation is:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Another way of viewing this set of equations is as follows:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1m}x_m &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2m}x_m &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nm}x_m &= b_n \end{aligned}$$

The solution for a linear system of equations is the numerical value for the variables $x_1, x_2, \cdots, x_m$ that would satisfy the above equations.

5.2 Gaussian Elimination

5.2.1 Naive Gauss Elimination

The following are the steps to program the Naive Gauss Elimination method. Assuming an n number of equations in n unknowns of the form $Ax = b$:

1. Form the combined matrix $G = A|b$
2. Forward elimination: Use the pivot elements G_{kk} on the row k as "Pivot" elements. Use the "Pivot" elements to eliminate the components G_{ik} with i from $k+1$ to n. This is done by iterating i from $k+1$ to n and for each row i, doing the operation $row_i = row_i - row_k \frac{G_{ik}}{G_{kk}}$.
3. Backward substitution: The element x_i with i running backwards from n to 1 can be found using the equation:
$$x_i = \frac{G_{i(n+1)} - \sum_{l=i+1}^{n} G_{il}x_l}{G_{ii}}$$

Python code:

```
import numpy as np
# Procedure for naive Gauss Elimination
def GaussElimination(A,b):
  n = len(A)
  G = (np.vstack([A.astype(np.float).T, b.astype(np.float)])).T
  for k in range(n):
    for i in range(k + 1, n):
      G[i] = G[i] - G[i][k]/G[k][k]*G[k]
  x = np.zeros(n)
  for i in range(n-1,-1,-1):
    x[i] = (G[i][n] - sum([G[i][l]*x[l] for l in range(i + 1, n)]))/G[i][i]
  return x

# An example
A = np.array([[1, 2, 3, 5], [2, 0, 1, 4], [1, 2, 2, 5], [4, 3, 2, 2]])
b = np.array([-4, 8, 0, 10])
# Applying the procedure to the example
GaussElimination(A, b)
```

This method is not perfect for two reasons. The first reason is that it fails to find a solution if a pivot element (one of the diagonal elements) is zero. In general, the naive Gauss elimination code presented above can be augmented so that the pivot elements are chosen to be the largest elements in the column below. This augmentation is termed: “Partial Pivoting”. Partial Pivoting is performed by exchanging the rows. In this case, the order of the variables x_i stays the same. In rare cases, there could be a need to exchange the columns as well; a procedure termed “Complete Pivoting”, but this adds the complication of re-arranging the variables.
The second reason why it is called naive is that the method essentially produces an upper triangular matrix that can be easily solved. It is important to note that if the system is already a lower triangular matrix, then there is no need to employ the forward elimination procedure, because we can simply use forward substitution to find $x_1, x_2, \cdots, x_n$ in this order.

5.2.2 Gauss Jordan Elimination

Another method that is rooted in the Gauss elimination method is the Gauss-Jordan elimination method. Two steps are added to the previous algorithm. The first step is that each pivot row is normalized by the pivot element. The second step is that the coefficients above the pivot element are also eliminated. This results in not needing the backward substitution step.
Python code:

```
import numpy as np
def GJElimination(A,b):
  n = len(A)
  G = (np.vstack([A.astype(np.float).T, b.astype(np.float)])).T
  for k in range(0,n):
    G[k] = G[k]/G[k][k]
    for i in range(k):
      G[i] = G[i] - G[i][k]*G[k]
    for i in range(k+1,n):
      G[i] = G[i] - G[i][k]*G[k]
  x = G[:,n]
  return x
A = np.array([[1, 2, 3, 5], [2, 0, 1, 4], [1, 2, 2, 5], [4, 3, 2, 2]])
b = np.array([-4, 8, 0, 10])
GJElimination(A, b)
```

5.3 LU decomposition

Let $Ax = b$ be a linear system of equations. Let $A = LU$ be an LU decomposition for A with L being a lower triangular matrix and U being an upper triangular matrix. Then: $Ax = LUx = L(Ux)$. Set $y = Ux$, then $Ly = b$. The elements of the vector y can be solved using forward substitution. Then, the system $Ux = y$ can be solved using backward substitution.

Python code:

```
import numpy as np
def LU(A):
  n = len(A)
  L = np.identity(n)
  U = np.array(A).astype(np.float)
  for k in range(n):
    for i in range(k + 1, n):
      L[i,k] = U[i,k]/U[k,k]
      U[i] = U[i] - U[i,k]/U[k,k]*U[k]
  return [L, U]

def LUAB(A,b):
  n = len(A)
  L = LU(A)[0]
  U = LU(A)[1]
  y = np.zeros(n)
  x = np.zeros(n)
  for i in range(n):
    BB = 0
    for l in range(0, i):
      BB = L[i,l]*y[l] + BB
    y[i] = (b[i] - BB)/L[i,i]
  for i in range(n-1,-1,-1):
    BB = 0
    for l in range(i,n):
      BB = U[i,l]*x[l] + BB
      x[i] = (y[i] - BB)/U[i,i]
  return x

A = [[1, 2, 3, 5], [2, 0, 1, 4], [1, 2, 2, 5], [4, 3, 2, 2]]
b = [-4, 8, 0, 10]
LUAB(A, b)
```

5.3.1 Cholesky factorisation for positive definite symmetric matrices

This decomposition is similar to the LU decomposition and if a linear system $Ax = b$ is such that A is positive definite, then the Cholesky decomposition enables the quick calculation of U and then the backward and forward substitution can be used to solve the system. The factorization can be found by noticing that the diagonal entry of A admits the following:

$$A_{ii} = \sum_{k=1}^{n} U_{ki}U_{ki} = \sum_{k=1}^{i} U_{ki}U_{ki} = \sum_{k=1}^{i-1} U_{ki}U_{ki} + U_{ii}^2$$

where the fact that $U_{ki} = 0$ whenever $k > i$ was used. Therefore:

$$U_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} U_{ki}U_{ki}}$$

On the other hand, an off diagonal component A_{ij} admits the following:

$$A_{ij} = \sum_{k=1}^{n} U_{ki}U_{kj} = \sum_{k=1}^{i} U_{ki}U_{kj} = \sum_{k=1}^{i-1} U_{ki}U_{kj} + U_{ii}U_{ij}$$

where again, the fact that $U_{ki} = 0$ whenever $k > i$ was used. Therefore:

$$U_{ij} = \frac{A_{ij} - \sum_{k=1}^{i-1} U_{ki}U_{kj}}{U_{ii}}$$

Notice that for the factorization to work, we can't have negative values for A_{ii} or zero values for U_{ii} which is guaranteed not to occur because A is positive definite.
in general the Cholesky Decomposition should be much more efficient than the LU decomposition for positive definite symmetric matrices.
Python code for finding U in Cholesky Decomposition:

```
import numpy as np
def CD(A):
  A = np.array(A)
  n = len(A)
  U = np.zeros([n,n])
  for i in range(n):
    BB = 0
    for k in range(i):
      BB = BB + U[k, i]*U[k, i]
    U[i, i] = np.sqrt(A[i, i] - BB)
    for j in range(i,n):
      BB = 0
      for k in range(i):
        BB = BB + U[k, i]*U[k, j]
      U[i, j] = (A[i, j] - BB)/U[i, i]
  return U
```

6 Root finding [Non-linear equations]

6.1 Introduction

Root finding refers to the process of finding the values or locations of roots (solutions) of an equation or function numerically. It involves applying iterative methods to approximate the values where the function crosses the x-axis or becomes zero. It is crucial for solving transcendental equations and systems of equations that arise in physics, such as finding energy eigenvalues in quantum mechanics, determining equilibrium points in classical mechanics, or solving nonlinear equations in fluid dynamics.

6.2 Bracketing methods

Bracketing methods for finding the roots of $f(x) = 0$, where f is continuous, rely on the Intermediate Value Theorem.

Intermediate Value Theorem: Let $f : [a, b] \to \mathbb{R}$ be continuous and $f(a) \leq f(b)$. Then, $\forall y \in [f(a), f(b)] : \exists x \in [a, b]$ such that $y = f(x)$. The same applies if $f(b) \leq f(a)$.
The consequence of the theorem is that if the function f is such that $f(a) \leq 0 \leq f(b)$, then, there is $x \in [a, b]$ such that $f(x) = 0$. In the bracketing methods to find the roots of $f(x) = 0$, we assume that the function f is continuous and there exists only one single real root in the given interval.

6.2.1 Bisection method

In the bisection method, if $f(a)f(b) < 0$, an estimate for the root of the equation $f(x) = 0$ can be found as the average of a and b:

$$x_i = \frac{a+b}{2}$$

Upon evaluating $f(x_i)$, the next iteration would be to set either $a = x_i$ or $b = x_i$ such that for the next iteration the root x_{i+1} is between a and b. The following describes an algorithm for the bisection method given $a < b$, $f(x)$, ε_s, and maximum number of iterations:
Step 1: Evaluate $f(a)$ and $f(b)$ to ensure that $f(a)f(b) < 0$. Otherwise, exit with an error.
Step 2: Calculate the value of the root in iteration i as $x_i = \frac{a_i+b_i}{2}$. Check which of the following

applies:
If $f(x_i) = 0$, then the root has been found, the value of the error $\varepsilon_r = 0$. Exit.
If $f(x_i)f(a_i) < 0$, then for the next iteration, x_{i+1} is bracketed between a_i and x_i. The value of $\varepsilon_r = \frac{x_{i+1}-x_i}{x_{i+1}}$.
If $f(x_i)f(b_i) < 0$, then for the next iteration, x_{i+1} is bracketed between x_i and b_i. The value of $\varepsilon_r = \frac{x_{i+1}-x_i}{x_{i+1}}$.
Step 3: Set $i = i + 1$. If i reaches the maximum number of iterations or if $\varepsilon_r \leq \varepsilon_s$, then the iterations are stopped. Otherwise, return to step 2 with the new interval a_{i+1} and b_{i+1}.

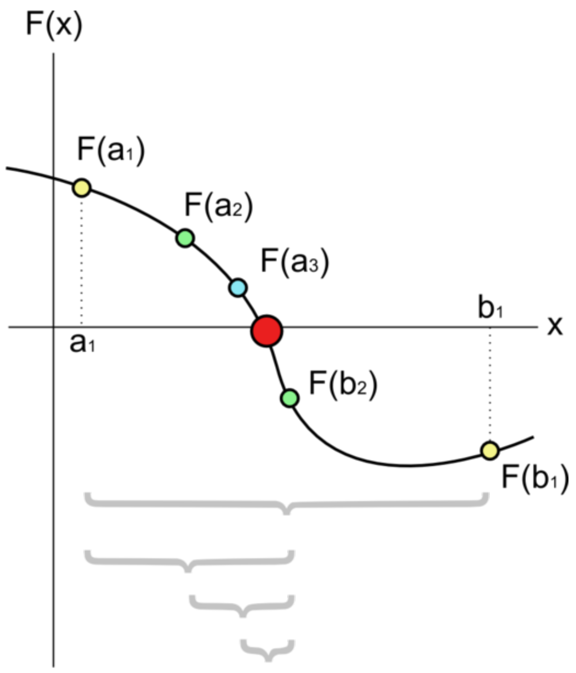

Figure 3: Bisection method

Python code:

```
def bisect(xl,xr,eps,imax):
  xm = xr
  x=[];e=[];it=[]
  for i in range(imax):
      xm_old = xm
      xm = (xl+xr)*0.5
      if (xm != 0):
          ea = abs((xm - xm_old)/xm)*100
      it.append(i); x.append(xm); e.append(ea)
      test = f(xl)*f(xm)
      if (test < 0.0):
          xu = xm
      elif (test > 0.0):
          xl = xm
      else:
          ea = 0.0
      if(ea < eps):
          break
  return it, x, e
```

6.2.2 False position method

In the false position method, the new estimate x_i at iteration i is obtained by considering the linear function passing through the two points $(a, f(a))$ and $(b, f(b))$. The point of intersection of this line with the x axis can be obtained using one of the following formulas:

$$x_i = a - f(a)\frac{(b-a)}{f(b)-f(a)} = b - f(b)\frac{(b-a)}{f(b)-f(a)} = \frac{af(b)-bf(a)}{f(b)-f(a)}$$

Upon evaluating $f(x_i)$, the next iteration would be to set either $a = x_i$ or $b = x_i$ such that for the next iteration the root x_{i+1} is between a and b. The following describes an algorithm for the false position method method given $a < b$, $f(x)$, ε_s, and maximum number of iterations:
Step 1: Evaluate $f(a)$ and $f(b)$ to ensure that $f(a)f(b) < 0$. Otherwise exit with an error. Step 2: Calculate the value of the root in iteration i as $x_i = \frac{a_i f(b_i) - bf(a_i)}{f(b_i)-f(a_i)}$. Check which of the following applies:
If $f(x_i) = 0$, then the root has been found, the value of the error $\varepsilon_r = 0$. Exit.
If $f(x_i)f(a_i) < 0$, then for the next iteration, x_{i+1} is bracketed between a_i and x_i. The value of $\varepsilon_r = \frac{x_{i+1}-x_i}{x_{i+1}}$.
If $f(x_i)f(b_i) < 0$, then for the next iteration, x_{i+1} is bracketed between x_i and b_i. The value of $\varepsilon_r = \frac{x_{i+1}-x_i}{x_{i+1}}$.
Step 3: If i reaches the maximum number of iterations or if $\varepsilon_r \leq \varepsilon_s$, then the iterations are stopped. Otherwise, return to step 2.

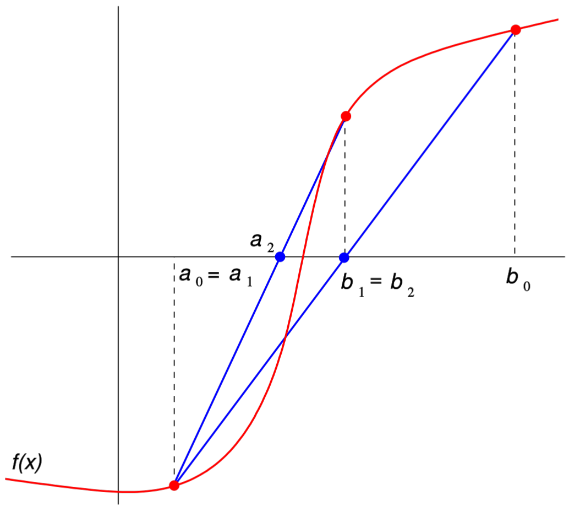

Figure 4: False position method

Python code:

```
def falsepos(xl,xr,eps,imax):
xm = xr
x=[];e=[];it=[]
for i in range(imax):
    xm_old = xm
    xm = (xl*f(xr)+xr*f(xl))/(f(xl)+f(xr))
    if (xm != 0):
        ea = abs((xm - xm_old)/xm)*100
    it.append(i); x.append(xm); e.append(ea)
    test = f(xl)*f(xm)
    if (test < 0.0):
        xu = xm
    elif (test > 0.0):
        xl = xm
    else:
        ea = 0.0
    if(ea < eps):
        break
return it, x, e
```

6.3 Open methods

Unlike bracketing methods, open methods require only a single starting value or two starting value that do not neccessarily bracket a root. Open methods may diverge as the computation progresses, but when they do converge, they usually do so much faster than bracketing methods.

6.3.1 Fixed point iteration method

Let $g : V \to V$. A fixed point of g is defined as $x \in V$ such that $x = g(x)$. If $g : \mathbb{R} \to \mathbb{R}$, then a fixed point of g is the intersection of the graphs of the two functions $y = x$ and $y = g(x)$. The fixed-point iteration method relies on replacing the expression $f(x) = 0$ with the expression $x = g(x)$. Then, an initial guess for the root x_1 is assumed and input as an argument for the function g. The output $g(x_1)$ is then the estimate x_2. The process is then iterated until the output $x_{n+1} \approx g(x_n)$. The following is the algorithm for the fixed-point iteration method. Assuming x_0, ε_s, and maximum number of iterations N: Set $x_{n+1} = g(x_n)$, and calculate $\varepsilon_r = \frac{x_{n+1}-x_n}{x_{n+1}}$ and compare with ε_s. If $|\varepsilon_r| \leq \varepsilon_s$ or if $n = N$, then stop the procedure, otherwise, repeat.

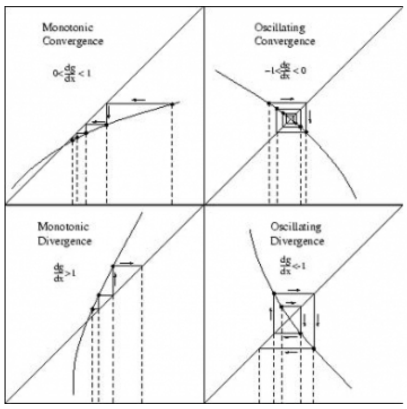

Figure 5: Fixed point iteration method

Convergence of Fixed Point Iteration method:
The objective of the fixed-point iteration method is to find the true value x_t that satisfies $x_t = g(x_t)$. In each iteration we have the estimate $x_{i+1} = g(x_i)$. Using the mean value theorem, we can write the following expression:

$$g(x_i) = g(x_t) + g'(\xi)(x_i - x_t)$$

for some ξ in the interval between x_i and the true value V_t. Replacing $g(V_t)$ and $g(x_i)$ in the above expression yields:

$$x_{i+1} = x_t + g'(\xi)(x_i - x_t) \Rightarrow x_{i+1} - x_t = g'(\xi)(x_i - x_t)$$

The error after iteration $i+1$ is equal to $E_{i+1} = x_t - x_{i+1}$ while that after iteration i is equal to $E_i = x_t - x_i$. Therefore, the above expression yields:

$$E_{i+1} = g'(\xi)E_i$$

For the error to reduce after each iteration, the first derivative of g, namely g', should be bounded by 1 in the region of interest (around the required root):

$$|g'(\xi)| < 1$$

When this condition is satisfied, then the sequence $\{x_i\}$ converges with $E_{i+1} \propto E_i$. This is called linear convergence.

6.3.2 Newton-Raphson method

The Newton-Raphson method converges very fast in most cases, and it can be extended quite easily to multi-variable equations. To find the root of the equation $f(x_t) = 0$, the Newton-Raphson method depends on the Taylor Series Expansion of the function around the estimate x_i to find a better estimate x_{i+1}:

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + \mathcal{O}(h^2)$$

where x_{i+1} is the estimate of the root after iteration $i+1$ and x_i is the estimate at iteration i. Assuming $f(x_{i+1}) = 0$ and rearranging:

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

The procedure is as follows. Setting an initial guess x_0, tolerance ε_s, and maximum number of iterations N: At iteration i, calculate $x_i \approx x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})}$ and ε_r. If $\varepsilon_r \leq \varepsilon_s$ or if $i \geq N$, stop the procedure. Otherwise repeat. Note: unlike the previous methods, the Newton-Raphson method relies on calculating the first derivative of the function $f(x)$. This makes the procedure very fast, however, it has two disadvantages. The first is that this procedure doesn't work if the function $f(x)$ is not differentiable. Second, the inverse $(f'(x_i))^{-1}$ can be slow to calculate when dealing with multi-variable equations.

Convergence of Newton-Raphson method:
The error in the Newton-Raphson Method can be roughly estimated as follows. The estimate x_{i+1} is related to the previous estimate x_i using the equation:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \Rightarrow 0 = f(x_i) + f'(x_i)(x_{i+1} - x_i)$$

Additionally, using Taylor's theorem, and if x_t is the true root with $f(x_t) = 0$ we have:

$$f(x_t) = 0 = f(x_i) + f'(x_i)(x_t - x_i) + \frac{f''(\xi)}{2!}(x_t - x_i)^2$$

for some ξ in the interval between x_i and x_t. Subtracting the above two equations yields:

$$0 = f'(x_i)(x_t - x_{i+1}) + \frac{f''(\xi)}{2!}(x_t - x_i)^2$$

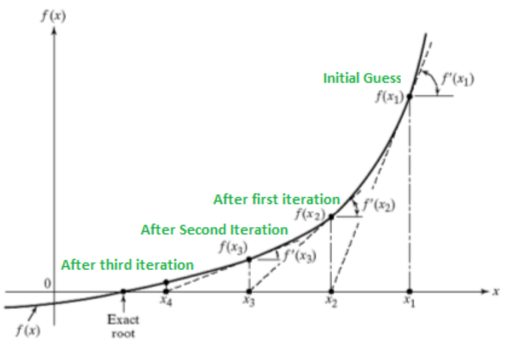

Figure 6: Newton Raphson method

Since $E_{i+1} = x_t - x_{i+1}$ and $E_i = x_t - x_i$ then:

$$E_{i+1} = -\frac{f''(\xi)}{2!f'(x_i)} E_i^2$$

If the method is converging, we have $f''(\xi) \approx f''(x_t)$ and $f'(\xi) \approx f'(x_t)$ therefore:

$$E_{i+1} \propto E_i^2$$

Therefore, the error is squared after each iteration, i.e., the number of correct decimal places approximately doubles with each iteration. This behaviour is called quadratic convergence.

6.3.3 Secant method

The secant method is an alternative to the Newton-Raphson method by replacing the derivative $f'(x)$ with its finite-difference approximation. The secant method thus does not require the use of derivatives especially when $f(x)$ is not explicitly defined. In certain situations, the secant method is preferable over the Newton-Raphson method even though its rate of convergence is slightly less than that of the Newton-Raphson method. Consider the problem of finding the root of the function $f(x) = 0$. Starting with the Newton-Raphson equation and utilizing the following approximation for the derivative $f'(x_i)$:

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

the estimate for iteration $i + 1$ can be computed as:

$$x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$$

Obviously, the secant method requires two initial guesses x_0 and x_1.

6.4 Comparison between bracketing and open methods

Open methods do not rely on having the root squeezed between two values, but rather rely on an initial guess and then apply an iterative process to get better estimates for the root. Open methods are usually faster in convergence if they converge, but they don't always converge. For multi-dimensions, i.e., for solving multiple nonlinear equations, open methods are easier to implement than bracketing methods which cannot be easily extended to multi-dimensions. Bracketing methods are usually slow, but reliable, i.e., if we are given an interval with a root inside it, the bracketing methods will definitely converge.

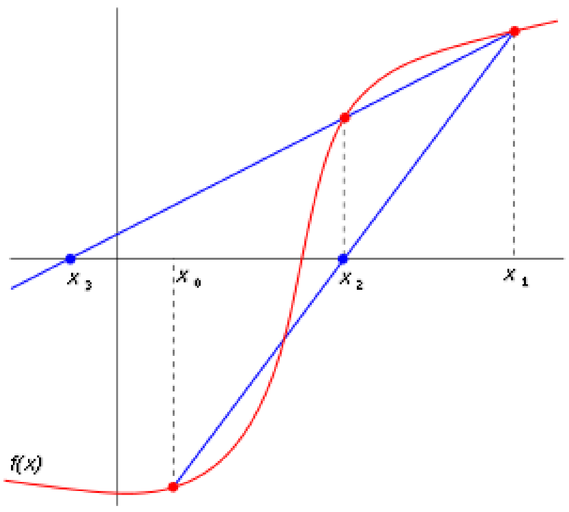

Figure 7: Secant method

7 Optimization methods

7.1 1D unconstrained optimization

7.1.1 Golden section search method

The golden-section search is a technique for finding an extremum (minimum or maximum) of a function inside a specified interval. For a strictly unimodal function with an extremum inside the interval, it will find that extremum, while for an interval containing multiple extrema (possibly including the interval boundaries), it will converge to one of them. If the only extremum on the interval is on a boundary of the interval, it will converge to that boundary point. The method operates by successively narrowing the range of values on the specified interval, which makes it relatively slow, but very robust. The technique derives its name from the fact that the algorithm maintains the function values for four points whose three interval widths are in the ratio $\phi : 1 : \phi$ where ϕ is the golden ratio.

Unlike finding a zero, where two function evaluations with opposite sign are sufficient to bracket a root, when searching for a minimum, three values are necessary. The golden-section search is an efficient way to progressively reduce the interval locating the minimum. The key is to observe that regardless of how many points have been evaluated, the minimum lies within the interval defined by the two points adjacent to the point with the least value so far evaluated. The diagram above illustrates a single step in the technique for finding a minimum. The functional values of $f(x)$ are on the vertical axis, and the horizontal axis is the x parameter. The value of $f(x)$ has already been evaluated at the three points: x_1, x_2, and x_3. Since f_2 is smaller than either f_1 or f_3, it is clear that a minimum lies inside the interval from x_1 to x_3. The next step in the minimization process is to "probe" the function by evaluating it at a new value of x, namely x_4. It is most efficient to choose x_4 somewhere inside the largest interval, i.e. between x_2 and x_3. From the diagram, it is clear that if the function yields f_{4a}, then a minimum lies between x_1 and x_4, and the new triplet of points will be x_1, x_2, and x_4. However, if the function yields the value f_{4b}, then a minimum lies between x_2 and x_3, and the new triplet of points will be $x_2, x_4, and x_3$. Thus, in either case, we can construct a new narrower search interval that is guaranteed to contain the function's minimum.

The golden-section search chooses the spacing between these points in such a way that these points have the same proportion of spacing as the subsequent triple x_1, x_2, x_4 or x_2, x_4, x_3. By maintaining the same proportion of spacing throughout the algorithm, we avoid a situation in

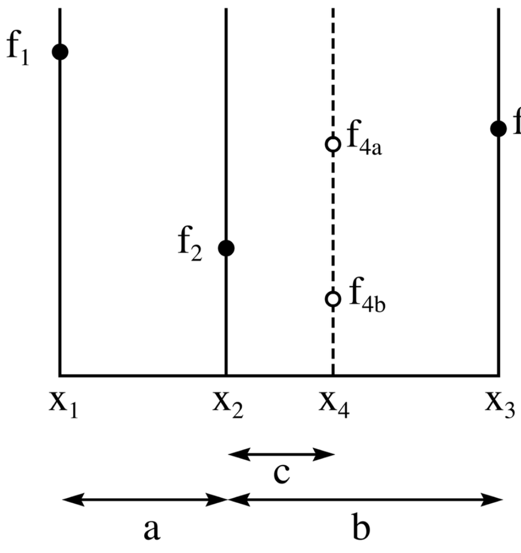

Figure 8: Golden Section Search method

which x_2 is very close to x_1 or x_3 and guarantee that the interval width shrinks by the same constant proportion in each step. Mathematically, to ensure that the spacing after evaluating $f(x_4)$ is proportional to the spacing prior to that evaluation, if $f(x_4)$ is f_{4a} and our new triplet of points is x_1, x_2, and x_4, then we want $\frac{c}{a} = \frac{a}{b}$. However, if $f(x_4)$ is f_{4b} and our new triplet of points is x_2, x_4, and x_3, then we want $\frac{c}{b-c} = \frac{a}{b}$. Eliminating c from these two simultaneous equations yields $\left(\frac{b}{a}\right)^2 - \frac{b}{a} = 1$, or $\frac{b}{a} = \varphi$, where ϕ is the golden ratio: $\varphi = \frac{1+\sqrt{5}}{2} = 1.618033988\ldots$

8 Ordinary Differential Equations

8.1 Introduction

Ordinary Differential Equations (ODEs) are equations that relate the derivatives of one or more smooth functions with an independent variable $x \in \mathbb{R}$.

8.2 Classification of ODE's

Depending on the boundary conditions, an ODE can be classified as either an Initial Value Problem (IVP) or a Boundary Value Problem (BVP).

- **Initial Value Problem:** An initial value problem is an ODE given with initial conditions of the dependent variable and its derivative at a particular value of the independent variable. This usually applies to dynamic systems whose independent variable is time. For example, Newton's second law of motion
$$m\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = F(x,t)$$
is an initial value problem because the initial value at $t = 0$ of the displacement $x(0)$ and the velocity $x'(0)$ are required to reach a solution. The initial values lead to a particular path that is a function of time $x(t)$.

- **Boundary Value Problem:** A boundary value problem is an ODE given with boundary conditions at different points. This usually applies to static systems whose independent variable is position. For example, the Euler-Bernoulli beam deflection equation

$$YI\frac{\mathrm{d}^4y}{\mathrm{d}x^4} = q$$

is a boundary value problem. The boundary conditions of the displacement y, the rotation y', and the third and fourth derivatives, are usually given at boundary points on the beam which will then dictate the equilibrium deflection of the beam as a function of position $y(x)$.

8.3 Numerical methods of solving ODE IVPs

8.3.1 Euler's method

The Euler method is one of the simplest methods for solving first-order IVPs. Consider the following IVP:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = F(x,t)$$

Assuming that the value of the dependent variable x (say x_i) is known at an initial value t_i, then, we can use a Taylor approximation to estimate the value of x at $t = t_{i+1}$, namely $x(t_{i+1})$ with $h = t_{i+1} - t_i$:

$$x(t_{i+1}) = x_i + \left.\frac{\mathrm{d}x}{\mathrm{d}t}\right|_{t_i}(h) + \mathcal{O}(h^2)$$

Substituting the differential equation into the above equation yields:

$$x(t_{i+1}) = x_i + F(x_i, t_i)(h) + \mathcal{O}(h^2)$$

Therefore, as an approximation, an estimate for $x(t_{i+1})$ can be taken as x_{i+1} as follows:

$$x(t_{i+1}) \approx x_{i+1} = x_i + F(x_i, t_i)(h)$$

Using this estimate, the local truncation error is thus proportional to the square of the step size with the constant of proportionality related to the second derivative of x, which is the first derivative of the given IVP:

$$E_{\text{local}} = \mathcal{O}(h^2)$$

If the errors from each interval are added together, with $n = \frac{L}{h}$ being the number of intervals and L the total length $t_n - t_0$, then, the total error is:

$$E = \mathcal{O}(h^2)\frac{L}{h} = \mathcal{O}(h)$$

8.3.2 Runge-Kutta method

The Runge-Kutta methods developed by the German mathematicians C. Runge and M.W. Kutta are essentially a generalization of all the previous methods, like Euler's method. Consider the following IVP:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = F(x,t)$$

Assuming that the value of the dependent variable x (say x_i) is known at an initial value t_i, then, the Runge-Kutta methods employ the following general equation to calculate the value of x at $t = t_{i+1}$, namely x_{i+1} with $h = t_{i+1} - t_i$:

$$x_{i+1} = x_i + h\phi$$

where ϕ is a function of the form:

$$\phi = \alpha_1 k_1 + \alpha_2 k_2 + \alpha_3 k_3 + \cdots + \alpha_n k_n$$

α_j are constants while $k_j = F(x,t)$ evaluated at points within the interval $[x_i, x_{i+1}]$ and have the form:

$$\begin{aligned}
k_1 &= F(x_i, t_i)\\
k_2 &= F(x_i + q_{11}k_1h, t_i + p_1h)\\
k_3 &= F(x_i + q_{21}k_1h + q_{22}k_2h, t_i + p_2h)\\
&\vdots\\
k_n &= F(x_i + q_{(n-1),1}k_1h + q_{(n-1),2}k_2h + \cdots + q_{(n-1),(n-1)}k_{n-1}h, t_i + p_{n-1}h)
\end{aligned}$$

The constants α_j, and the forms of k_j are obtained by equating the value of x_{i+1} obtained using the Runge-Kutta equation to a particular form of the Taylor series. The k's are recurrence relationships, meaning k_1 appears in k_2, which appears in k_3, and so forth. This makes the method efficient for computer calculations. The error in a particular form depends on how many terms are used. The general forms of these Runge-Kutta methods could be implicit or explicit.

Name of method	α	k_i	**Global error** E
Explicit Euler's method	$\alpha_1 = 1$	$k_1 = F(x_i, t_i)$	$\mathcal{O}(h)$
Heun's method	$\alpha_1 = \frac{1}{2}$ $\alpha_2 = \frac{1}{2}$	$k_1 = F(x_i, t_i)$ $k_2 = F(x_i + hk_1, t_i + h)$	$\mathcal{O}(h^2)$
Midpoint method	$\alpha_1 = 0$ $\alpha_2 = 1$	$k_1 = F(x_i, t_i)$ $k_2 = F(x_i + \frac{h}{2}k_1, t_i + \frac{h}{2})$	$\mathcal{O}(h^2)$
Fourth order Runge Kutta method	$\alpha_1 = \frac{1}{6}$ $\alpha_2 = \frac{1}{3}$ $\alpha_3 = \frac{1}{3}$ $\alpha_4 = \frac{1}{6}$	$k_1 = F(x_i, t_i)$ $k_2 = F\left(x_i + \frac{h}{2}k_1, t_i + \frac{h}{2}\right)$ $k_3 = F\left(x_i + \frac{h}{2}k_2, t_i + \frac{h}{2}\right)$ $k_4 = F(x_i + hk_3, t_i + h)$	$\mathcal{O}(h^4)$

8.4 Examples

8.4.1 Lotka-Volterra equations

The Lotka-Volterra equations, also known as the Lotka-Volterra predator-prey model, are a pair of first-order nonlinear differential equations, frequently used to describe the dynamics of biological systems in which two species interact, one as a predator and the other as prey. The populations change through time according to the pair of equations:
$\frac{dx}{dt} = \alpha x - \beta xy,$
$\frac{dy}{dt} = \delta xy - \gamma y,$
where

- the variable x is the population density of prey (for example, the number of rabbits per square kilometre);
- the variable y is the population density of some predator (for example, the number of foxes per square kilometre);
- $\frac{dy}{dt}$ and $\frac{dx}{dt}$ represent the instantaneous growth rates of the two populations;
- t represents time;

- The prey's parameters, α and β, describe, respectively, the maximum prey per capita growth rate, and the effect of the presence of predators on the prey growth rate.
- The predator's parameters, γ, δ, respectively describe the predator's per capita death rate, and the effect of the presence of prey on the predator's growth rate.
- All parameters are positive and real.

The solution of the differential equations is deterministic and continuous. This, in turn, implies that the generations of both the predator and prey are continually overlapping.

9 Partial Differential Equations

9.1 Methods for solving PDEs

9.1.1 Jacobi Relaxation method

Consider the Poisson equation in 2D:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

where $u(x, y)$ is the unknown function and $f(x, y)$ is a given source term.

To solve this equation numerically, we discretize the 2D domain into a grid with uniform spacing h in both the x and y directions. The grid points are denoted as (x_i, y_j), where $x_i = i \cdot h$ and $y_j = j \cdot h$. We also introduce the notation $u_{i,j} \approx u(x_i, y_j)$, which represents the approximate value of u at grid point (i, j).

The Jacobi method updates the values of $u_{i,j}$ iteratively using the formula:

$$u_{i,j}^{(k+1)} = \frac{1}{4}\left(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)}\right) - \frac{h^2}{4} f_{i,j}$$

where $u_{i,j}^{(k)}$ is the value of u at grid point (i, j) in the k iteration, and $f_{i,j} = f(x_i, y_j)$ is the value of the source term at grid point (i, j). Repeat this process until the solution converges to the desired accuracy.

9.1.2 Gauss-Siedel method

Consider the Poisson equation in 2D:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

where $u(x, y)$ is the unknown function and $f(x, y)$ is a given source term.

To solve this equation numerically, we discretize the 2D domain into a grid with uniform spacing h in both the x and y directions. The grid points are denoted as (x_i, y_j), where $x_i = i \cdot h$ and $y_j = j \cdot h$. We also introduce the notation $u_{i,j} \approx u(x_i, y_j)$, which represents the approximate value of u at grid point (i, j).

The Gauss-Seidel method is an iterative technique that updates the values of $u_{i,j}$ until they converge to the solution. The update formula for the $k+1$ iteration at the grid point (i, j) is given by:

$$u_{i,j}^{(k+1)} = \frac{1}{4}\left(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k+1)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k+1)}\right) - \frac{h^2}{4} f_{i,j}$$

where $u_{i,j}^{(k)}$ is the value of u at grid point (i, j) in the k iteration, and $f_{i,j} = f(x_i, y_j)$ is the value of the source term at grid point (i, j).

The Gauss-Seidel method can be enhanced using successive over-relaxation (SOR) by introducing a relaxation parameter ω such that $0 < \omega < 2$. The update formula with relaxation becomes:

$$u_{i,j}^{(k+1)} = (1-\omega)u_{i,j}^{(k)} + \frac{\omega}{4}\left(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k+1)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k+1)}\right) - \frac{\omega h^2}{4} f_{i,j}$$

To solve the Poisson equation using the Gauss-Seidel method, follow these steps:

1. Discretize the 2D domain into a grid with spacing h.
2. Initialize the values of $u_{i,j}$ at each grid point.
3. Iterate the update formula for each grid point until convergence, using either the standard Gauss-Seidel method ($\omega = 1$) or the SOR method with a chosen ω.
4. Repeat the iterations until the solution converges to the desired accuracy.

The Jacobi Relaxation method and the Gauss-Seidel method are both iterative techniques used to solve systems of linear equations that arise from discretizing PDEs. However, they differ in how they update the solution variables during each iteration. In the Jacobi method, all solution variables are updated simultaneously using values from the previous iteration, and these updated values are used for the next iteration. On the other hand, the Gauss-Seidel method updates the solution variables one by one as soon as their updated values are available. This means that the Gauss-Seidel method uses the most recent available information to update the variables, potentially leading to faster convergence compared to the Jacobi method.

9.1.3 Implicit methods (eg: Crank-Nicolson method)

Consider the 1D heat equation:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

where $u(x,t)$ is the temperature distribution, α is the thermal diffusivity, and t and x represent time and spatial coordinates, respectively.

To solve this equation numerically, we discretize both time and space using finite difference approximations. Let u_i^n represent the approximation of u at the grid point (x_i, t_n), where $x_i = i \cdot \Delta x$ and $t_n = n \cdot \Delta t$. The Crank-Nicolson method updates u at each time step as follows:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{\alpha}{2}\left(\frac{u_{i+1}^{n+1} - 2u_i^{n+1} + u_{i-1}^{n+1}}{\Delta x^2} + \frac{u_{i+1}^{n} - 2u_i^{n} + u_{i-1}^{n}}{\Delta x^2}\right)$$

where Δt and Δx are the time and spatial step sizes, respectively.

The Crank-Nicolson method is an implicit method, which means that the equation at each time step is a system of linear equations involving u^{n+1}. This method is unconditionally stable and leads to better accuracy compared to explicit methods like the Forward Euler method. It takes into account the average of values at time steps n and $n+1$, providing a more accurate approximation of the solution.

9.2 Example: Schrodinger equation

The time-dependent Schrödinger equation in one dimension is given by:

$$\frac{\partial \Psi(x,t)}{\partial t} = \frac{i\hbar}{2m}\frac{\partial^2 \Psi(x,t)}{\partial x^2} - \frac{i}{\hbar}V(x)\Psi(x,t),$$

where $\Psi(x,t)$ is the wave function, $\hbar$ is the reduced Planck's constant, m is the mass of the particle, and $V(x)$ is the potential energy function.

Finite Difference Method (FDM)

The finite difference method is one of the simplest numerical methods for solving partial differential equations. In this method, we discretize both the spatial and temporal domains. Let's consider a spatial domain x divided into N grid points, and a temporal domain t divided into M time steps. The wave function Ψ is approximated at each grid point x_i and time step t_j as $\Psi_{i,j} = \Psi(x_i, t_j)$.

The spatial derivative is approximated using the central difference formula:

$$\frac{\partial^2 \Psi_{i,j}}{\partial x^2} \approx \frac{\Psi_{i+1,j} - 2\Psi_{i,j} + \Psi_{i-1,j}}{\Delta x^2},$$

where Δx is the spatial grid spacing.

The time derivative is approximated using the forward Euler method:

$$\frac{\partial \Psi_{i,j}}{\partial t} \approx \frac{\Psi_{i,j+1} - \Psi_{i,j}}{\Delta t},$$

where Δt is the time step.

Substituting these approximations into the Schrödinger equation, we get a difference equation that relates $\Psi_{i,j+1}$ with $\Psi_{i,j}$ and the values at neighboring grid points. This system of difference equations can be solved iteratively to obtain the time evolution of the wave function.

10 Random Numbers and Monte-Carlo Methods

10.1 Generating Random Numbers

Random numbers play a crucial role in many computational techniques, including Monte Carlo simulations. Pseudo-random numbers are commonly used for simulations as they appear to be random but are generated deterministically using algorithms. One popular method for generating pseudo-random numbers is the linear congruential generator (LCG). It is defined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \mod m \tag{2}$$

where X_n is the current random number, a is a multiplier, c is an increment, and m is the modulus. The initial value X_0 is called the seed. The generated random numbers X_n lie in the range [0, $m-1$] and can be scaled to obtain random numbers in any desired range [a, b] using:

$$\text{Random number in } [a, b] = a + \left(\frac{b-a}{m-1}\right) X_n \tag{3}$$

It is important to choose appropriate values for a, c, and m to ensure the generated sequence has good statistical properties and a long period before repeating.

10.2 Monte Carlo Simulations

Monte Carlo simulations are powerful computational techniques that rely on random sampling to approximate complex systems or perform numerical integration. One common application is to estimate integrals of the form:

$$I = \int_a^b f(x)\, dx \tag{4}$$

The basic idea is to sample random points x_i from the interval [a, b] and compute the average value of $f(x_i)$ over these points. The integral I is then approximated as:

$$I \approx \frac{b-a}{N} \sum_{i=1}^{N} f(x_i) \tag{5}$$

where N is the number of random samples. The accuracy of the approximation improves with increasing N due to the law of large numbers. Monte Carlo simulations can handle high-dimensional integrals and complex systems, which are challenging for traditional numerical methods. They are widely used in physics, finance, optimization, and many other fields.

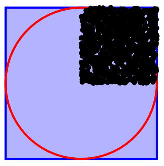

Figure 9: Estimating π using the Monte Carlo method.

Another application of Monte-Carlo simulations is calculating numerical quantities like π. We have a square with side length $2r$ centered at the origin, and a quarter circle with radius r also centered at the origin. We randomly generate points (x, y) within the square. The ratio of the number of points inside the circle to the total number of points generated will give an approximation of $\pi/4$. The more points we use, the better our estimate will be.

The area of the circle is πr^2, and the area of the square is $4r^2$. Thus, the ratio of the areas is:

$$\frac{\pi r^2}{4r^2} = \frac{\pi}{4}.$$

Therefore, to approximate π, we can use the following formula:

$$\pi \approx 4 \times \left(\frac{\text{number of points inside the circle}}{\text{total number of points}}\right).$$

Monte Carlo simulations offer a powerful and flexible toolset for approximating solutions to various problems. By increasing the number of samples, one can achieve higher accuracy in the estimation. However, it's essential to consider the trade-off between accuracy and computational cost, especially for large-scale simulations.