MA637 - Numerical Analysis and Computing Lecture Notes

Abhinav Jha
Indian Institute of Technology, Gandhinagar
Winter Semester 2024-2025

Preface

These notes are designed to provide a structured and comprehensive understanding of the course content. They will cover key topics, concepts, and computational techniques that are fundamental to numerical analysis. Please note that this is the first iteration (Version 0.1.0) of the notes and hence there is a chance that some of the content is incorrect. If you find some flaws, please email me at `abhinav.jha@iitgn.ac.in`.

Contents

Chapter 1

Interpolation

Interpolation has various definitions depending on the search engine. For example, *Wikipedia* states,

"Interpolation is a type of estimation, a method of constructing (finding) new data points based on the range of a discrete set of known data points."

Blackphoto says,

"It is a technique used by digital scanners, cameras, and printers to increase the size of an image in pixels by averaging the colour and brightness values of surrounding pixels."

One can see such an example in image processing. A rather famous (or infamous) example is the *Ecce Homo* painting (see Fig. 1 (left)). This is a fresco painting painted in 1930 by the Spanish painter Elías García Martínez depicting Jesus Christ. With wear and tear, the painting got degraded, and in 2012, an 81-year-old lady, Cecilia Giménez "tried" to restore it (see Fig. 1 (right)); as we can see, it is not very good, and hence it was named *Ecce Mono.* We can get much better results with modern image processing techniques (which inherently use a form of interpolation).

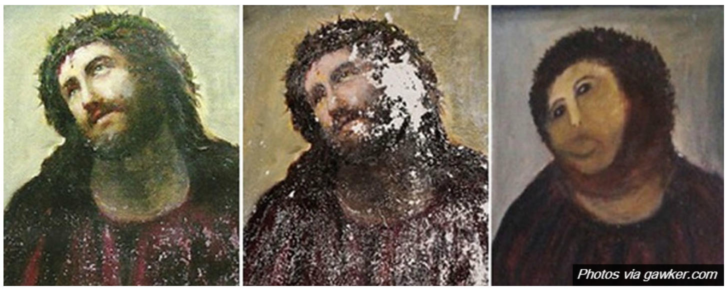

Figure 1.1: Elías García Martínez, Ecce Homo: The leftmost photograph, taken in 2010, shows some initial flaking of the paintwork. The central photograph was taken in July 2012, just a month before the attempted restoration, showing the extent of damage and deterioration. The rightmost photograph documents the artwork following Giménez's efforts to repair it.

In interpolation, we try to approximate general functions by a "simple" class of functions. In analysis, a powerful result connects the continuous functions and polynomial approximation:

the Weierstrass Approximation theorem given by Karl Weierstrass.

Figure 1.2: Karl Weierstrass: 31 October 1815-19 February 1897

Theorem 1.1. *[1, Theorem 5.4.14]* **(Weierstrass Approximation Theorem)** *Let $f \in \mathcal{C}[a,b]$. Then for each $\varepsilon > 0$ there exists a polynomial $p(x)$ with the property that*

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

This theorem is important because polynomials have excellent differentiation and integration properties as their derivatives and integrals are polynomials. Another interpretation of Theorem 1 is that given a continuous function on a closed and bounded interval, there exists a polynomial, i.e. , as "close" to the given function as desired.

But in analysis, there exists one more kind of polynomial approximation, and that is the Taylor's theorem

Theorem 1.2. *[1, Theorem 6.4.1]* **(Taylor's Theorem)** *Suppose $f \in \mathcal{C}^n[a,b]$ and $f^{(n+1)}$ exists on $[a,b]$ and $x_0 \in [a,b]$. For every $x \in [a,b]$ there exists a number $\xi(x) \in [x_0, x]$ with*

$$f(x) = P_n(x) + R_n(x),$$

where $P_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$ and $R_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x - x_0)^{n+1}$.

Figure 1.3: Brook Taylor: 18 August 1685-29 December 1731

There are two issues here:

1. We need to know the higher derivatives of $f(x)$.

2. This is a *local* approximation, i.e., the approximation is excellent near x_0 but we need certain global approximation. For example, if we do the Taylor series expansion for $\exp(x)$ around zero, then it becomes worse as we move away from zero (see Fig. 1.4).

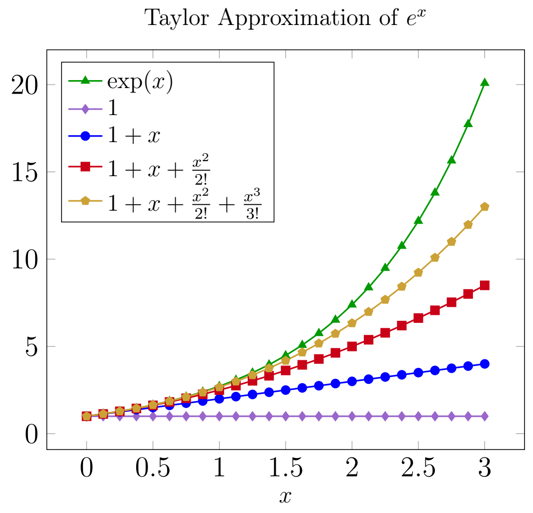

Figure 1.4: Taylor polynomials for exponential function approximated at $x = 0$.

However, it should be noted that the Taylor theorem is still a powerful result whose main purpose is the derivation of numerical techniques and error estimation.

1.1 Polynomial Interpolation

Suppose we have a finite set of data points f_i associated with parameters x_i. We want to depict these data points as a function $f(x)$ with the property that $f(x_i) = f_i$. This is clearly not well-defined since there are many such functions. But if we restrict to finite-dimensional spaces (such as polynomials), then we can define such functions, or to be more precise, the process is well-defined.

We first start with the idea of polynomial interpolation. Polynomials representing an unknown functional dependence of the discrete set of data points are called *interpolants*. The main problem that we want to tackle with interpolation is:

Problem: Given a set of $(n+1)$ data points say $\{(x_i, f_i)\}_{i=0}^n$ find a polynomial $p_n(x)$ of degree n satisfying

$$p_n^{\mathbb{V}}(x_i) = f_i \qquad \text{for all} \quad i = 0, 1, \ldots, n.$$

Now the general form of a polynomial $p_n^{\mathbb{V}}(x)$ is given by

$$p_n^{\mathbb{V}}(x) = \sum_{i=0}^{n} c_i x^{n-i} := c_0 x^n + c_1 x^{n-1} + \cdots + c_n,$$

for coefficients $c_i \in \mathbb{R}$. Since each polynomial of degree n can be determined by $(n+1)$ coefficients, we can re-write the above problem as solving the following system of equations:

$$c_0 x_i^n + c_1 x_i^{n-1} + \cdots + c_{n-1} x_i + c_n = f_i \qquad i = 0, 1, 2, \ldots, n,$$

or in the matrix form

$$\mathbf{V}\mathbf{c} = \mathbf{f} \tag{1.1}$$

where

$$\mathbf{V} = \begin{bmatrix} x_0^n & x_0^{n-1} & \dots & x_0 & 1 \\ x_1^n & x_1^{n-1} & \dots & x_1 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_n^n & x_n^{n-1} & \dots & x_n & 1 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix}, \quad \text{and} \quad \mathbf{f} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{bmatrix}.$$

This system has a unique solution if $\mathbf{V}$ is invertible [5, Theorem 3.10], which is equivalent to saying that $\det(\mathbf{V}) \neq 0$. This matrix $\mathbf{V}$ is called as the *Vandermonde matrix* and it's determinant is given by

$$\det(\mathbf{V}) = \prod_{i=0}^{n-1} \prod_{j=i+1}^{n} (x_i - x_j).$$

Figure 1.5: Alexandre-Théophile Vandermonde: 28 February 1735 – 1 January 1796

This determinant is non-zero if we have distinct points. Hence, from now on, we assume we have $(n+1)$ distinct points.

The algorithm for using the polynomial interpolation using Vandermonde matrix for finding solution at a given point x_{eval} is given in Algorithm 1.

Note that we have introduced the notation $p_n^{\mathbb{V}}(x)$ only to denote the polynomial $p_n(x)$ computed using the Vandermonde matrix.

1.1.1 Drawbacks

Even though Eq. (1.1) has a perfect mathematical solution, computationally, it is not that good. The reason being Vandermonde matrices are ill-conditioned[1] (we will do conditioning of a system in the following chapters). The matrix $\mathbf{V}$ has a large condition number leading to inaccurate solutions.

To understand why the Vandermonde matrix is ill-conditioned for large n, we can plot x^k for $0 \leq k \leq n$ in $[0, 1]$ (see Fig. 1.6). Even though x^k are distinct for larger k, they tend to look the same. As a result, it is harder to identify projections of a particular polynomial $p_n^{\mathbb{V}}(x)$ into the nearly collinear basis of monomials x^k for large k.

[1]**Ill-Conditioned System**: In numerical analysis, the condition number of a function quantifies the extent to which the output can change in response to small variations in the input. It measures a function's sensitivity to input changes or errors, indicating how much an input error can propagate into the output. A problem with a low condition number is said to be *well-conditioned* while a problem with a high condition number is said to be *ill-conditioned*.

Algorithm 1 Vandermonde Interpolation

```
Given: Data sets $\{(x_i, f_i)\}_{i=0}^{n}$, Evaluation point $x_{\text{eval}}$.
Find: Interpolated polynomial $p_n^{\mathbb{V}}(x_{\text{eval}})$.
  Step 1: Compute Vandermonde Matrix
  Initialize an empty Vandermonde matrix $\mathbf{V}$ of size $(n+1) \times (n+1)$
  for $i = 0$ to $n$ do
    for $j = 0$ to $n$ do
      $\mathbf{V}_{i,j} = x_i^{(n-j)}$
    end for
  end for

  Step 2: Solve the System of Linear Equations
  Solve the system $\mathbf{V} \cdot \mathbf{c} = \mathbf{f}$ to get coefficient vector $\mathbf{c}$

  Step 3: Evaluate the Vandermonde Polynomial $p_n^{\mathbb{V}}(x)$ at $x_{\text{eval}}$
  Initialize $p_n^{\mathbb{V}}(x_{\text{eval}}) = 0$
  for $i = 0$ to $n$ do
    $p_n^{\mathbb{V}}(x_{\text{eval}}) = p_n^{\mathbb{V}}(x_{\text{eval}}) + c_i \cdot x_{\text{eval}}^{(n-i)}$
  end for

  return $p_n^{\mathbb{V}}(x_{\text{eval}})$
```

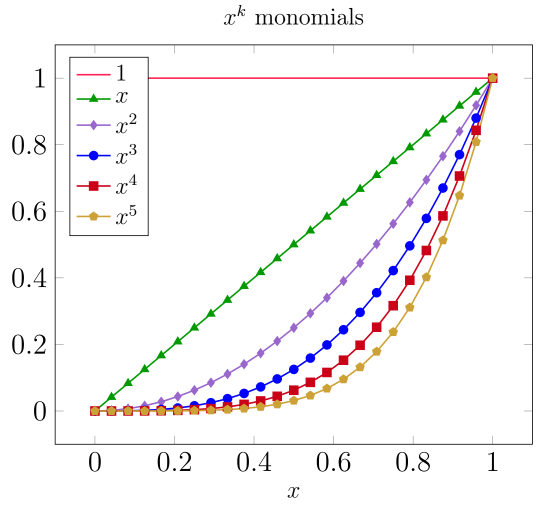

Figure 1.6: Monomial basis x^k for $k = 0, 1, \ldots, 5$.

1.2 Lagrange Interpolation

After examining how unstable polynomial interpolation is, we need to develop more stable methods. One of the most known methods is the Lagrange interpolation. The formula was first published by Waring in 1779, rediscovered by Euler in 1783, and published by Lagrange in 1795 (Jeffreys & Jeffreys, 1988).

Let us start with a basic example of two points (x_0, f_0) and (x_1, f_1), then we define

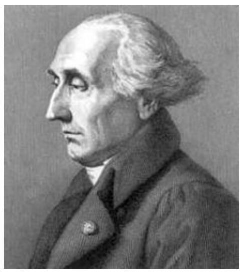

Figure 1.7: Joseph-Louis Lagrange: 25 January 1736-10 April 1813

functions:

$$\mathbb{L}_0(x) = \frac{x - x_1}{x_0 - x_1} \qquad \text{and} \qquad \mathbb{L}_1(x) = \frac{x - x_0}{x_1 - x_0}. \tag{1.2}$$

Then, a linear interpolating polynomial passing through the above points is given by:

$$p_1(x) = \mathbb{L}_0(x)f_0 + \mathbb{L}_1(x)f_1 = \frac{x - x_1}{x_0 - x_1}f_0 + \frac{x - x_0}{x_1 - x_0}f_1,$$

as $\mathbb{L}_0(x_0) = 1; \mathbb{L}_0(x_1) = 0; \mathbb{L}_1(x_0) = 0;$ and $\mathbb{L}_1(x_1) = 1$, we have $p_1(x_0) = f_0$ and $p_1(x_1) = f_1$. This polynomial is called the *Lagrange linear interpolating polynomial*. In fact, this is a unique polynomial. Fig. 1.8 shows $\mathbb{L}_0(x)$ and $\mathbb{L}_1(x)$ for $x_0 = 0$ and $x_1 = 1$.

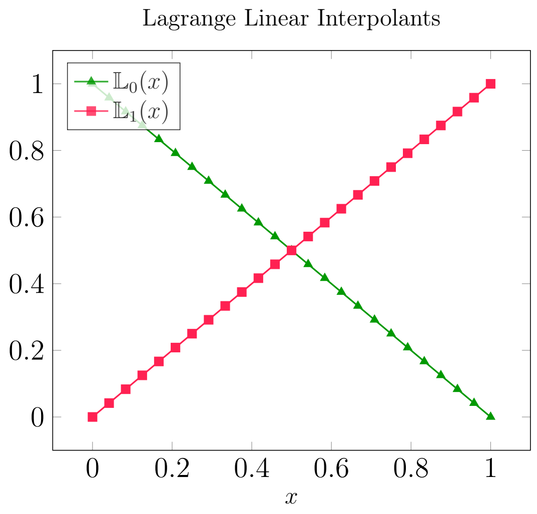

Figure 1.8: Lagrange linear interpolating polynomials for $x_i = \{0, 1\}$.

What happens if we generalize this concept, i.e., we have $\{(x_i, f_i)\}_{i=0}^n$? In this case we first need to construct for each $i = 0, 1, \ldots, n$ a function $\mathbb{L}_{n,i}(x)$ with the property that

$$\mathbb{L}_{n,i}(x_k) = \delta_{ik} \qquad \text{for} \quad k = 0, \ldots, n.$$

Based on Eq. (1.2) the general form should look like:

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

Then, we can define the polynomial as

$$p_n^{\mathbb{L}}(x) = \sum_{i=0}^{n} f_i \mathbb{L}_{n,i}(x), \tag{1.3}$$

where we have used the notation $p_n^{\mathbb{L}}(x)$ to denote the interpolating polynomial obtained by the Lagrange interpolation. If the degree of the polynomial is clear we can write $\mathbb{L}_{n,i}(x)$ as $\mathbb{L}_i(x)$. We call $\mathbb{L}_{n,i}(x)$ as the n^{th} *Lagrange interpolating polynomial* (see Fig. 1.9). One can compute the Lagrange interpolating polynomial using algorithm 2.

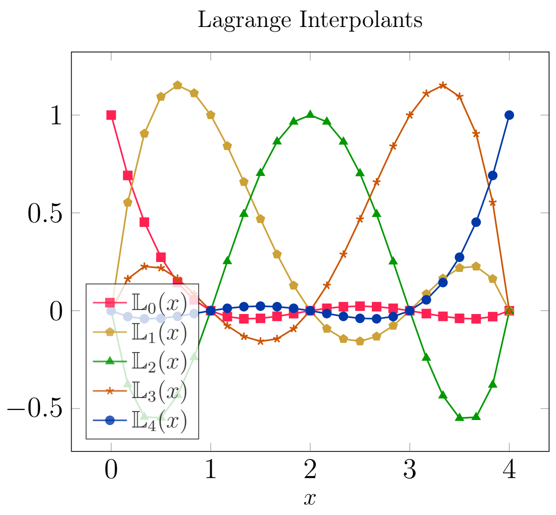

Figure 1.9: Lagrange interpolating polynomials defined over $x_i = 0, 1, 2, 3, 4$.

We have certain remarks for the Lagrange interpolation:

1. We note that in Eq. (1.3) $p_n^{\mathbb{L}}(x)$ maps the linear space $\mathbb{R}^{n+1}$ to the space of polynomials $\mathbb{P}_n$ which is a linear map.

2. We can extend the Lagrange interpolant to any continuous function $f(x)$ by

$$p_n^{\mathbb{L}} f(x) = \sum_{i=0}^{n} f(x_i)\mathbb{L}_i(x).$$

3. The operator $p_n^{\mathbb{L}}(x)$ is a projection, i.e., $p_n^{\mathbb{L}} q = q$ for all $q \in \mathbb{P}_n$.

Now we present a theorem that tells us about the error obtained using Lagrange interpolation.

Theorem 1.3. *Suppose $\{x_0, x_1, \ldots, x_n\}$ are distinct numbers in the interval $[a, b]$ and $f \in \mathcal{C}^{n+1}[a, b]$. Then for each $x \in [a, b]$ there exists a number $\xi(x) \in (a, b)$ with*

$$f(x) = p_n^{\mathbb{L}}(x) + \frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{i=0}^{n} (x - x_i), \tag{1.4}$$

where $p_n^{\mathbb{L}}(x)$ is given by Eq. (1.3).

Proof. Note that if $x = x_k$ then $f(x_k) = p_n^{\mathbb{L}}(x_k)$ for any $k = 0, 1, \ldots, n$. Hence Eq. (1.4) is trivial for any $\xi(x) \in (a, b)$.

Suppose $x \neq x_k$ for any $k = 0, 1, \dots, n$ then define a function g for t in $[a, b]$ as

$$g(t) = f(t) - p_n^{\mathbb{L}}(t) - \left[f(x) - p_n^{\mathbb{L}}(x)\right] \prod_{i=0}^{n} \frac{(t - x_i)}{(x - x_i)}.$$

Since $f \in \mathcal{C}^{n+1}[a, b]$ and $p_n^{\mathbb{L}} \in \mathcal{C}^{\infty}[a, b]$ we have $g \in \mathcal{C}^{n+1}[a, b]$.

Theorem 1.4. *[3, Theorem 1.10]* **(Generalized Rolle's Theorem)**

Suppose $f \in \mathcal{C}[a, b]$ is n-times differentiable on (a, b). If $f(x) = 0$ at $(n+1)$ distinct points $a \leq x_0 < x_1 < \cdots < x_n \leq b$ then there exists a number $c \in (x_0, x_n)$ $(\subset (a, b))$ such that $f^{(n)}(c) = 0$.

For $t = x_k$ for any k, we have

$$g(x_k) = f(x_k) - p_n^{\mathbb{L}}(x_k) = 0.$$

Moreover $g(x) = 0$. Thus $g \in \mathcal{C}^{n+1}[a, b]$ with $(n + 2)$ distinct zeros. By Generalized Rolle's theorem 1.2 there exists a $\xi \in (a, b)$ for which $g^{(n+1)}(\xi) = 0$. So,

$$0 = g^{(n+1)}(\xi) = f^{(n+1)}(\xi) - p_n^{\mathbb{L}(n+1)}(\xi) - \left[f(x) - p_n^{\mathbb{L}}(x)\right] \frac{d^{n+1}}{dt^{n+1}} \left[\prod_{i=0}^{n} \frac{(t - x_i)}{(x - x_i)}\right]_{t=\xi}. \tag{1.5}$$

Now, $p_n^{\mathbb{L}}(x)$ is a polynomial of degree at most n. Hence, $p_n^{\mathbb{L}(n+1)}(x) = 0$. Also, $\prod_{i=0}^{n} \frac{(t-x_i)}{(x-x_i)}$ is a polynomial of degree $(n + 1)$ with leading coeffecient being $\frac{1}{\prod_{i=0}^{n}(x-x_i)}$. Hence,

$$\frac{d^{n+1}}{dt^{n+1}} \left(\prod_{i=0}^{n} \frac{(t - x_i)}{(x - x_i)}\right) = \frac{(n + 1)!}{\prod_{i=0}^{n}(x - x_i)}.$$

Hence, Eq. (1.5) becomes

$$f^{(n+1)}(\xi) - \left[f(x) - p_n^{\mathbb{L}}(x)\right] \frac{(n + 1)!}{\prod_{i=0}^{n}(x - x_i)} = 0 \qquad \Rightarrow \qquad f(x) = p_n^{\mathbb{L}}(x) + \frac{f^{(n+1)}(\xi)}{(n + 1)!} \prod_{i=0}^{n}(x - x_i).$$

□

Note that this error term is similar to Taylor's theorem, but it has information on all the points instead of the error being concentrated along one point.

1.2.1 Drawbacks

Lagrange interpolant suffers from certain drawbacks. The first one is regarding its *computational complexity*[2]. For the evaluation of an unknown point x, we will check the computational

[2]**Computational Complexity**: Computational complexity measures how hard it is for a computer to solve a problem as the size of the problem increases. It tells us how much time and resources are needed to find a solution.

Algorithm 2 Lagrange Interpolation

```
Given: Data sets $\{(x_i, f_i)\}_{i=0}^{n}$, Evaluation point $x_{\text{eval}}$.
Find: Interpolated polynomial $p_n^{\mathbb{L}}(x_{\text{eval}})$.
  Step 1: Compute Lagrange Basis Polynomials $\mathbb{L}_i(x)$
  for $i = 0$ to $n$ do
    $\mathbb{L}_i(x_{\text{eval}}) = 1$
    for $j = 0$ to $n$ do
      if $j \neq i$ then
        $\mathbb{L}_i(x_{\text{eval}}) = \mathbb{L}_i(x_{\text{eval}}) \times \frac{x_{\text{eval}} - x_j}{x_i - x_j}$
      end if
    end for
  end for
  ----------------------------------------
  Step 2: Compute Lagrange Polynomial $p_n^{\mathbb{L}}(x)$ at $x_{\text{eval}}$
  Initialize $p_n^{\mathbb{L}}(x_{\text{eval}}) = 0$
  for $i = 0$ to $n$ do
    $p_n^{\mathbb{L}}(x_{\text{eval}}) = p_n^{\mathbb{L}}(x_{\text{eval}}) + f_i \times \mathbb{L}_i(x_{\text{eval}})$
  end for
  ----------------------------------------
  return $p_n^{\mathbb{L}}(x_{\text{eval}})$
```

complexity. An individual Lagrange interpolating polynomial of degree n looks like

$$\mathbb{L}_i(x) = \prod_{j=0, j\neq i}^{n} \frac{(x - x_j)}{(x_i - x_j)},$$

and then $p_n^{\mathbb{L}}(x) = f_0 + \mathbb{L}_0(x) + f_1\mathbb{L}_1(x) + \cdots + f_n\mathbb{L}_n(x)$. For the computation of each $\mathbb{L}_i(x)$ we need $\mathcal{O}(n)$ multiplications. As we have $(n+1)$ points, we need $\mathcal{O}\left(n^2 + n\right)$ operations. The final operation for computing of $p_n^{\mathbb{L}}(x)$ is of multiplication and addition and hence a total of $\mathcal{O}(n)$ operations. Therefore, in totality, we need $\mathcal{O}(n^2)$ operations, which is not very nice as, generally, we prefer to have linear ($\mathcal{O}(n)$) complexity.

Apart from the above drawback, another major drawback is that if we want to add a new point, say (x_{n+1}, f_{n+1}), then we need to perform new computations from scratch.

But there are advantages as well; for example, the computation of $\{\mathbb{L}_i(x)\}_{i=0}^{n}$ is independent of $f(x_k)$. Another one is that it does not depend on the arrangement of nodes.

1.2.2 Runge Phenomena

In 1901, Carl David Tolmé Runge observed that while approximating

$$f(x) = \frac{1}{1 + 25x^2}, \quad x \in [-1, 1],$$

using polynomial approximation, there are large errors at the endpoints of the interval while using equally spaced points (see Fig. 1.11). This is what is called as the *Runge phenomena* and the above function is called the *Runge function*.

Figure 1.10: Carl David Tolmé Runge: 30 August 1856-3 January 1927

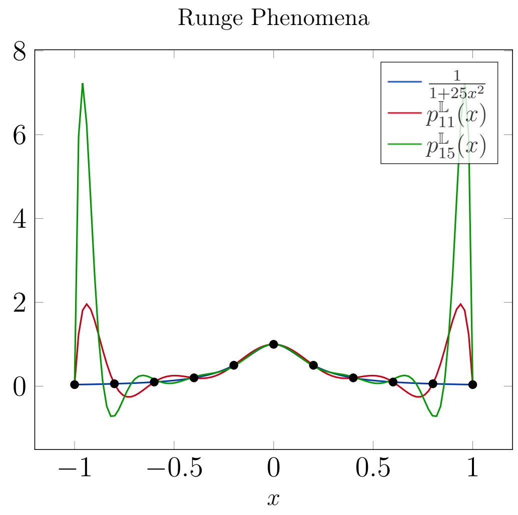

Figure 1.11: Runge phenomena for the function $1/(1+25x^2)$. $p_{11}^{\mathbb{L}}(x)$ refers to an approximation computed using 11 points (the dots refer to $\{x_i\}_{i=0}^{10}$), $p_{15}^{\mathbb{L}}(x)$ refers to approximation using 15 points.

Let us look at the interpolation error and try to understand this phenomenon. In Theorem 1.3 it was seen that

$$f(x) - p_n^{\mathbb{L}}(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i) \quad \text{for} \quad \xi \in (-1, 1).$$

Thus, we have

$$\max_{-1\leq x\leq 1} |f(x) - p_n^{\mathbb{L}}(x)| \leq \max_{-1\leq x\leq 1} \left| \frac{f^{(n+1)}(\xi)}{(n+1)!} \right| \max_{-1\leq x\leq 1} \prod_{i=0}^{n} |x - x_i|.$$

Now, it can be shown (although not very easily) that $\max_{-1\leq x\leq 1} \prod_{i=0}^{n} |x - x_i| \leq h^{n+1} n!$ where $h = 2/n$ and we suppose that the $(n+1)^{\text{th}}$ derivative of $f(x)$ can be bounded by M_{n+1} which in turn can be bounded by $5^{n+1}(n+1)!$ (see this PDF). Hence in total

$$\lim_{n\to\infty} \left(\max_{-1\leq x\leq 1} |f(x) - p_n^{\mathbb{L}}(x)| \right) \leq \lim_{n\to\infty} \left(\left(\frac{10}{n} \right)^{n+1} n! \right) = \infty.$$

To mitigate this problem, one idea is to use a non-uniform grid with points accumulated at the endpoints. If one is interested, I suggest this excellent review paper by Berrut and Trefethen [2].

1.3 Newton Divided Difference Interpolation

We noticed in Sec. 1.2 that Lagrange interpolation suffers from $\mathcal{O}(n^2)$ evaluation computational complexity. Now, we have another interpolating method that overcomes this and is referred to as the *Newton Divided Differences.*

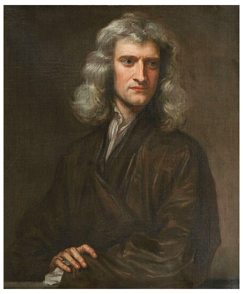

Figure 1.12: Isaac Newton: 4 January 1643-31 March 1727

Let $p_n(x)$ be a polynomial interpolating the data points $\{(x_i, f_i)\}_{i=0}^{n}$. Another way of expressing such a polynomial is

$$p_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_n \prod_{i=0}^{n-1}(x - x_i), \tag{1.6}$$

for appropriate constants $\{a_i\}_{i=0}^{n}$. Now, the question is, how do we determine these coefficients? At $x = x_0$ we have $p_n(x_0) = f_0$. Hence, $y_0 = a_0$. Similarly at $x = x_1$, $p_n(x_1) = f_1$ which implies

$$a_1 = \frac{f_1 - a_0}{x_1 - x_0} = \frac{f_1 - f_0}{x_1 - x_0}.$$

Now, we can continue in this manner and compute each a_i. For this, we introduce the divided difference (DD) notation. The zeroth divided difference of a function $f(x)$ with respect to x_i is denoted by $f[x_i] = f(x_i) = f_i$. For the rest, we define them in a recursive way.

- 1st DD of $f(x)$ with respect to x_i and x_{i+1} is

$$f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i}. \tag{1.7}$$

- 2nd DD of $f(x)$ with respect to $x_i,\ x_{i+1}$ and x_{i+2} is

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i}. \tag{1.8}$$

- k^{th} DD of $f(x)$ with respect to $x_i,\ x_{i+1}, \ldots, x_{i+k}$ is

$$f[x_i, x_{i+1}, \ldots, x_{i+k-1}, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}. \tag{1.9}$$

- n^{th} DD of $f(x)$ with respect to $x_0,\ x_1, \ldots, x_n$ is

$$f[x_0, x_1, \ldots, x_n] = \frac{f[x_1, \ldots, x_n] - f[x_0, \ldots, x_{n-1}]}{x_n - x_0}. \tag{1.10}$$

Hence, we can rewrite the polynomial $p_n(x)$ defined in Eq. (1.6) as

$$p_n^{\mathbb{N}}(x) = f[x_0] + \sum_{k=1}^{n} f[x_0, x_1, \ldots, x_k](x - x_0)(x - x_1)\ldots(x - x_{k-1}). \tag{1.11}$$

We have introduced the notation $p_n^{\mathbb{N}}(x)$ to identify the Newton DD polynomial.

For simplicity, let us look at the DD table we obtain for 4 points (see Table 1.1)

$\boldsymbol{x}$	$\boldsymbol{f(x)}$ = 0th DD	1st DD	2nd DD	3rd DD
x_0	$f[x_0]$			
		$f[x_0, x_1] = \frac{f[x_1]-f[x_0]}{x_1-x_0}$		
x_1	$f[x_1]$		$f[x_0, x_1, x_2] = \frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0}$	
		$f[x_1, x_2] = \frac{f[x_2]-f[x_1]}{x_2-x_1}$		$f[x_0, x_1, x_2, x_3] = \frac{f[x_1,x_2,x_3]-f[x_0,x_1,x_2]}{x_3-x_0}$
x_2	$f[x_2]$		$f[x_1, x_2, x_3] = \frac{f[x_2,x_3]-f[x_1,x_2]}{x_3-x_1}$	
		$f[x_2, x_3] = \frac{f[x_3]-f[x_2]}{x_3-x_2}$		
x_3	$f[x_3]$			

Table 1.1: Divided difference table for four points x_0, x_1, x_2, x_3.

The Lagrange interpolating polynomial has a polynomial basis as $\mathbb{L}_{n,i}(x)$, we can consider the Newton DD as another method with a basis defined by $\omega_i(x) = \prod_{k=0}^{i-1}(x - x_k)$ for $i \geq 1$ and $\omega_0(x) = 1$ (see Fig. 1.13).

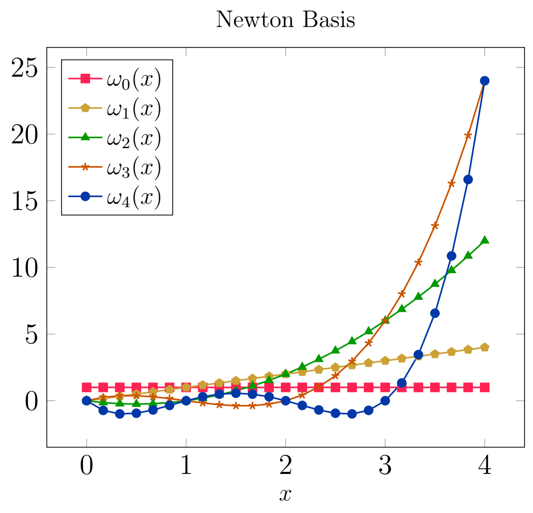

Figure 1.13: Newton basis polynomials defined over $x_i = 0, 1, 2, 3, 4$.

Now, we try to establish a relation between the DD and the derivatives of f. First, we recall the mean value theorem

Theorem 1.5. *[1, Theorem 6.2.4]* **(Mean Value Theorem)** *If $f \in \mathcal{C}[a, b]$ and $f(x)$ is differentiable in (a, b) then there exists a $c \in (a, b)$ such that*

$$f'(c) = \frac{f(b) - f(a)}{b - a}.$$

Now, if we apply the MVT on the interval $[x_i, x_{i+1}]$ then there exists a $\xi \in (x_i, x_{i+1})$ such that

$$f'(\xi) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} = f[x_i, x_{i+1}].$$

In fact, we can generalize this concept.

Theorem 1.6. *Suppose that $f \in \mathcal{C}^n[a,b]$ and $x_0, x_1, \ldots, x_n$ are distinct numbers in $[a,b]$. Then there exists a $\xi \in (a,b)$ with*

$$f[x_0, x_1, \ldots, x_n] = \frac{f^{(n)}(\xi)}{n!}.$$

Proof. Let $g(x) = f(x) - p_n^{\mathbb{N}}(x)$. Since, $f(x_i) = p_n^{\mathbb{N}}(x_i)$ at $i = 0, 1, \ldots, n$. Then $g(x)$ has $(n+1)$ distinct zeros in $[a,b]$. So by generlized Rolle's theorem 1.2 there exists a $\xi \in (a,b)$ with $g^{(n)}(\xi) = 0$, so

$$0 = f^{(n)}(\xi) - p_n^{\mathbb{N}(n)}(\xi).$$

Since, $p_n^{\mathbb{N}}(x)$ is polynomial of degree n with leading coefficient $f[x_0, x_1, \ldots, x_n]$, we have

$$p_n^{\mathbb{N}(n)}(x) = n! f[x_0, x_1, \ldots, x_n] \qquad \text{for all} \quad x.$$

Hence,

$$f[x_0, x_1, \ldots, x_n] = \frac{f^{(n)}(\xi)}{n!}.$$

□

Next, we give a result which gives an explicit representation of the Newton DD formula.

Theorem 1.7. *For distinct points $x_0, \ldots, x_n$, the n^{th} coefficient of the Newton interpolation satisfies*

$$f[x_0, x_1, \ldots, x_n] = \sum_{k=0}^{n} \frac{f(x_k)}{\prod_{i \neq k}(x_k - x_i)},$$

where $f[x_0] = f(x_0)$ in the case $n = 0$.

Proof. Using the representation Eq. (1.11) of the interpolant the n^{th} derivative of $p_n^{\mathbb{N}}(x)$ is given by to $f[x_0, \ldots, x_n]\omega_n^{(n)}(x)$ where $\omega_n(x) = \prod_{i=0}^{n-1}(x - x_i)$.

Now, the polynomial $p_n^{\mathbb{N}}(x)$ is just another representation of $p_n^{\mathbb{L}}(x)$. Hence their n^{th} derivatives must match.

For the Lagrange polynomial the n^{th} derivative is $\sum_{k=0}^{n} f(x_k)\mathbb{L}_k^{(n)}(x)$ (see Eq. (1.3)). Hence,

$$f[x_0, x_1, \ldots, x_n]\omega_n^{(n)}(x) = \sum_{k=0}^{n} f(x_k)\mathbb{L}_k^{(n)}(x).$$

Now, the k^{th} Lagrange interpolating polynomial is given by

$$\mathbb{L}_k(x) = \prod_{j\neq k}\left(\frac{x-x_j}{x_k-x_j}\right),$$

and it's n^{th} derivative $\mathbb{L}_k^{(n)}(x) = \frac{n!}{\prod_{j\neq k}(x_k-x_j)}$ since x^n is the leading term. Since the leading term in $\omega_n(x)$ is x^n,we get $\omega_n^{(n)}(x) = n!$. Cancelling out these factorial term we get the expression. □

The algorithm for computing the Newton DD interpolation is provided in Algorithm 3.

Algorithm 3 Newton Interpolation

```
Given: Data sets $\{(x_i, f_i)\}_{i=0}^n$, Evaluation point $x_{\text{eval}}$.
Find: Interpolated polynomial $p_n^{\mathbb{N}}(x_{\text{eval}})$.
  Step 1: Construct Divided Difference Table
  Initialize DD as a zero matrix of size $(n+1)\times(n+1)$.
  for $i = 0$ to $n$ do
    $\text{DD}_{i,0} = f_i$
  end for
  for $j = 1$ to $n$ do
    for $i = 0$ to $n-j$ do
      Compute:
        $$\text{DD}_{i,j} = \frac{\text{DD}_{i+1,j-1} - \text{DD}_{i,j-1}}{x_{i+j}-x_i}.$$
    end for
  end for
  ---------------------------------------------
  Step 2: Evaluate Newton Polynomial $p_n^{\mathbb{N}}(x)$ at $x_{\text{eval}}$
  Initialize $p_n^{\mathbb{N}}(x_{\text{eval}}) = \text{DD}_{0,0}$.
  for $k = 1$ to $n$ do
    Initialize $\omega = 1$.
    for $j = 0$ to $k-1$ do
      $\omega = \omega \times (x_{\text{eval}} - x_j)$
    end for
    Update the interpolated value:
      $$p_n^{\mathbb{N}}(x_{\text{eval}}) = p_n^{\mathbb{N}}(x_{\text{eval}}) + \text{DD}_{0,k}\cdot\omega.$$
  end for
  ---------------------------------------------
  return $p_n^{\mathbb{N}}(x_{\text{eval}})$.
```

1.3.1 Computational Complexity

We can rewrite the Newton interpolation as

$$\begin{aligned} p_n^{\mathbb{N}}(x) &= a_0 + a_1(x-x_0) + a_2(x-x_0)(x-x_1) + \dots + a_n(x-x_0)(x-x_1)\dots(x-x_{n-1}) \\ &= a_0 + (x-x_0)\left[a_1 + (x-x_1)\left\{a_2 + \dots + (x-x_{n-2})\left\{a_{n-1} + a_n(x-x_{n-1})\right\}\right\}\right]. \end{aligned}$$

We notice that each term requires one multiplication and one addition for evaluation, and we have n points. Hence, we require $2n$ operations, which is of $\mathcal{O}(n)$, whereas for Lagrange, we have $\mathcal{O}(n^2)$.

Another advantage of the Newton interpolation over Lagrange interpolation is that it is easy to update the DD table whenever we have a new data set as it does not require new computation only a modification of the DD table.

1.3.2 Forward Difference Formula

Suppose we have an equal spacing of points; then we can rewrite Newton's formula in a better way. Let $h = x_{i+1} - x_i$ for all $i = 0, 1, \dots, n-1$ and $x = x_0 + sh$. Then we can rewrite Eq. (1.11) as

$$\begin{aligned} p_n^{\mathbb{N}}(x) &= f[x_0] + (x-x_0)f[x_0,x_1] + (x-x_0)(x-x_1)f[x_0,x_1,x_2] + \dots \\ &\quad + (x-x_0)\dots(x-x_{n-1})f[x_0,x_1,\dots,x_n] \\ &= f[x_0] + shf[x_0,x_1] + s(s-1)h^2 f[x_0,x_1,x_2] + \dots \\ &\quad + s(s-1)\dots(s-n+1)h^n f[x_0,x_1,\dots,x_n] \\ &= f[x_0] + \sum_{k=1}^{n} s(s-1)\dots(s-k+1)h^k f[x_0,x_1,\dots,x_k]. \end{aligned}$$

Using the binomial coefficient notation

$$^{s}\mathrm{C}_k = \frac{s(s-1)\dots(s-k+1)}{k!},$$

we can express

$$p_n^{\mathbb{N}}(x) = p_n^{\mathbb{N}}(x_0 + sh) = f[x_0] + \sum_{k=1}^{n} {}^{s}\mathrm{C}_k k! h^k f[x_0,x_1,\dots,x_k].$$

Let us use the Δ notation for forward difference, i.e, $\Delta f(x_0) = f(x_1) - f(x_0)$. Similarly for higher differences we use the notation $\Delta^2 f(x_0) = \Delta f(x_1) - \Delta f(x_0)$, then we can rewrite the divided differences as

$$\begin{aligned} f[x_0,x_1] &= \frac{f[x_1]-f[x_0]}{x_1-x_0} = \frac{\Delta f(x_0)}{h} \\ f[x_0,x_1,x_2] &= \frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0} = \frac{1}{2h}\frac{\Delta f(x_1)-\Delta f(x_0)}{h} = \frac{\Delta^2 f(x_0)}{2!h^2} \\ &\vdots \\ f[x_0,x_1,\dots,x_n] &= \frac{\Delta^n f(x_0)}{n!h^n}. \end{aligned}$$

Hence,

$$p_n^{\mathbb{N}}(x) = f(x_0) + \sum_{k=1}^{n} {}^{s}\mathrm{C}_k \Delta^k f(x_0).$$

1.4 Hermite Interpolation

Definition 1.8. Let $\{x_0, x_1, \dots, x_n\}$ be $(n+1)$ distinct points in $[a,b]$ and for $i = 0, 1, \dots, n$ let m_i be a non-negative integer. Suppose that $f \in \mathcal{C}^m[a,b]$ where $m = \max_{0\le i\le n} m_i$. The *osculating polynomial* approximating $f(x)$ is the polynomial $p(x)$ of least degree such that

$$\frac{d^k p(x_i)}{dx^k} = \frac{d^k f(x_i)}{dx^k}, \quad \text{for} \quad i = 0, 1, \dots, n \quad \text{and} \quad k = 0, 1, \dots, m_i.$$

Not when $n = 0$ the osculating polynomial approximating f is the m_0^{th} Taylor polynomial for f at x_0. When $m_i = 0$ for all i then the osculating polynomial is the n^{th} Lagrange polynomial interpolating f at $x_0, x_1, \dots, x_n$.

Hermite Polynomials

If $m_i = 1$ for all $i = 0, 1, \dots, n$ then we get the Hermite polynomials. For a given function f these polynomials agree with f at $x_0, x_1, \dots, x_n$. In addition they agree with there derivatives as well.

Figure 1.14: Charles Hermite: 24 December 1822- 14 January 1901

Theorem 1.9. *If $f \in \mathcal{C}^1[a,b]$ and $x_0, x_1, \dots, x_n \in [a,b]$ are distinct, the unique polynomial of least degree agreeing with f and f' at $x_0, x_1, \dots, x_n$ is the Hermite polynomial of degree at most $2n+1$ given by*

$$p^{\mathbb{H}}_{2n+1}(x) = \sum_{j=0}^{n} f(x_j)\mathbb{H}_{n,j}(x) + \sum_{j=0}^{n} f'(x_j)\hat{\mathbb{H}}_{n,j}(x),$$

where for $\mathbb{L}_{n,j}(x)$ denoting the j^{th} Lagrange coefficient polynomial of degree n we have

$$\mathbb{H}_{n,j}(x) = [1 - 2(x - x_j)\mathbb{L}_{n,j}(x_j)']\,\mathbb{L}^2_{n,j}(x) \qquad \text{and} \qquad \hat{\mathbb{H}}_{n,j}(x) = (x - x_j)\mathbb{L}^2_{n,j}(x).$$

Moreover, if $f \in \mathcal{C}^{2n+2}[a,b]$ then

$$f(x) = p^{\mathbb{H}}_{2n+1}(x) + \frac{(x-x_0)^2(x-x_1)^2\dots(x-x_n)^2}{(2n+2)!} f^{(2n+2)}\left(\xi(x)\right),$$

for some unknown $\xi(x) \in (a,b)$.

Proof. We know that $\mathbb{L}_{n,j}(x_i) = \delta_{ij}$. Hence when $i \neq j$ $\mathbb{H}_{n,j}(x_i) = 0$ and $\hat{\mathbb{H}}_{n,j}(x_i) = 0$ whereas for each i

$$\mathbb{H}_{n,i}(x_i) = \left[1 - 2(x_i - x_i)\mathbb{L}'_{n,i}(x_i)\right]\mathbb{L}^2_{n,i}(x_i) = 1 \qquad \text{and} \qquad \hat{\mathbb{H}}_{n,i}(x_i) = (x_i - x_i)\mathbb{L}^2_{n,i}(x_i) = 0.$$

Hence, we can say $\mathbb{H}_{n,j}(x_i) = \delta_{ij}$ and $\hat{\mathbb{H}}_{n,j}(x_i) = 0$ for all i,j. As a consequence

$$p^{\mathbb{H}}_{2n+1}(x_i) = \sum_{j=0}^{n} f(x_j)\mathbb{H}_{n,j}(x_i) + \sum_{j=0}^{n} f'(x_j)\hat{\mathbb{H}}_{n,j}(x_i) = f(x_i),$$

so $p^{\mathbb{H}}_{2n+1}$ agrees with f at $x_0, x_1, \dots, x_n$.

Now we need to show that they match at the derivatives as well, i.e., $p^{\mathbb{H}'}_{2n+1}$ and f' match at x_i. We will tackle this be differentiating both the terms $\mathbb{H}_{n,j}$ and $\hat{\mathbb{H}}_{n,j}$.

The derivative of $\mathbb{H}_{n,j}(x)$ is given by

$$\begin{aligned}\mathbb{H}'_{n,j}(x) &= \left[1 - 2(x - x_j)\mathbb{L}'_{n,j}(x_j)\right] 2\mathbb{L}_{n,j}(x)\mathbb{L}'_{n,j}(x) + \mathbb{L}^2_{n,j}(x)\left[-2\mathbb{L}'_{n,j}(x_j)\right]\\ &= 2\mathbb{L}_{n,j}(x)\left[\left\{1 - 2(x-x_j)\mathbb{L}'_{n,j}(x_j)\right\}\mathbb{L}'_{n,j}(x) - \mathbb{L}_{n,j}(x)\mathbb{L}'_{n,j}(x_j)\right].\end{aligned}$$

As $\mathbb{L}_{n,j}(x_i) = \delta_{ij}$ we get that at $i \neq j$, $\mathbb{H}'_{n,j}(x_i) = 0$. At $i = j$ we have

$$\begin{aligned}\mathbb{H}'_{n,i}(x_i) &= \left[1 - 2(x_i - x_i)\mathbb{L}'_{n,i}(x_i)\right] 2\mathbb{L}_{n,i}(x_i)\mathbb{L}'_{n,i}(x_i) + \mathbb{L}^2_{n,i}(x_i)\left[-2\mathbb{L}'_{n,i}(x_i)\right]\\ &= \left[2\mathbb{L}'_{n,i}(x_i) - 2\mathbb{L}'_{n,i}(x_i)\right] = 0.\end{aligned}$$

Now for the second term the derivative is given by

$$\begin{aligned}\hat{\mathbb{H}}'_{n,j}(x) &= (x - x_j)2\mathbb{L}_{n,j}(x)\mathbb{L}'_{n,j}(x) + \mathbb{L}^2_{n,j}(x)\\ &= \mathbb{L}_{n,j}(x)\left[2(x-x_j)\mathbb{L}'_{n,j}(x) + \mathbb{L}_{n,j}(x)\right].\end{aligned}$$

At $x = x_i$ we have $\mathbb{L}_{n,j}(x_i) = \delta_{ij}$. Hence $\hat{\mathbb{H}}'_{n,j}(x_i) = 0$ if $i \neq j$ and at $i = j$

$$\begin{aligned}\hat{\mathbb{H}}'_{n,i}(x_i) &= (x_i - x_i)2\mathbb{L}_{n,i}(x_i)\mathbb{L}'_{n,i}(x_i) + \mathbb{L}^2_{n,i}(x_i)\\ &= 1.\end{aligned}$$

Hence $\hat{\mathbb{H}}'_{n,j}(x_i) = \delta_{ij}$. Therefore

$$p^{\mathbb{H}'}_{2n+1}(x_i) = \sum_{j=0}^{n} f(x_j)\mathbb{H}'_{n,j}(x_i) + \sum_{j=0}^{n} f'(x_j)\hat{\mathbb{H}}'_{n,j}(x_i) = f'(x_i).$$

Therefore $p^{\mathbb{H}}_{2n+1}$ agrees with f and $p^{\mathbb{H}'}_{2n+1}$ with f' at $x_0, x_1, \dots, x_n$. So,we have existence of a polynomial that agrees with f and f' at $\{x_i\}_{i=0}^n$.

For the uniqueness we will use the method of contradiction. Suppose there exists another polynomial of least degree say $q(x)$ such that

$$q(x_i) = f(x_i) \qquad \text{and} \qquad q'(x_i) = f'(x_i) \qquad \forall \;\; i.$$

Now consider the polynomial $D(x) = p^{\mathbb{H}}_{2n+1}(x) - q(x)$ of degree at most $(2n + 1)$. Obviously

$$D(x_i) = 0 \quad \text{and} \quad D'(x_i) = 0 \qquad \forall \;\; i$$

Hence x_i are distinct roots of multiplicity two. Therefore we have $2n + 2$ roots, which is a only possible if $D(x) = 0$. Hence, we get $p^{\mathbb{H}}_{2n+1}(x) = q(x)$ leading to a contradiction.

For showing the error term we will use the same strategy as in theorem 1.3., if $x = x_i$ for some i then we can choose $\xi(x)$ arbitrary.

Suppose $x \neq x_i$ for any i, then define

$$g(t) = f(t) - p^{\mathbb{H}}_{2n+1}(t) - \left[f(x) - p^{\mathbb{H}}_{2n+1}(x)\right] \prod_{i=0}^{n} \frac{(t - x_i)^2}{(x - x_i)^2}.$$

Now $g(x) = 0$ and $g(x_i) = 0$ for all i. Hence $g(t)$ has distinct $n + 2$ roots in $[a, b]$. Hence, by Rolle's theorem $g'(t)$ has $n + 1$ distinct roots between $x_0, x_1, \dots, x_n$ and x, say $\xi_0, \xi_1, \dots, \xi_n$.

Now taking the derivative of $g(t)$ with respect to t we get

$$\begin{aligned} g'(t) &= f'(t) - p^{\mathbb{H}'}_{2n+1}(t) - \frac{\left[f(x) - p^{\mathbb{H}}_{2n+1}(x)\right]}{\prod_{i=0}^{n}(x - x_i)^2} \frac{d}{dt}\left(\prod_{i=0}^{n}(t - x_i)^2\right)\\ &= f'(t) - p^{\mathbb{H}'}_{2n+1}(t) - \frac{\left[f(x) - p^{\mathbb{H}}_{2n+1}(x)\right]}{\prod_{i=0}^{n}(x - x_i)^2} \frac{d}{dt}\left((t - x_0)^2(t - x_1)^2 \dots (t - x_n)^2\right)\\ &= f'(t) - p^{\mathbb{H}'}_{2n+1}(t) - \frac{\left[f(x) - p^{\mathbb{H}}_{2n+1}(x)\right]}{\prod_{i=0}^{n}(x - x_i)^2}\Big(2(t - x_0)(t - x_1)^2 \dots (t - x_n)^2\\ &\quad +(t - x_0)^2 2(t - x_1) \dots (t - x_n)^2 + \dots + (t - x_0)^2(t - x_1)^2 \dots 2(t - x_n)\Big)\\ &= f'(t) - p^{\mathbb{H}'}_{2n+1}(t) - 2\frac{\left[f(x) - p^{\mathbb{H}}_{2n+1}(x)\right]}{\prod_{i=0}^{n}(x - x_i)^2} \sum_{k=0}^{n}(t - x_k) \prod_{j=0, j\neq k}^{n} (t - x_j)^2 \end{aligned}$$

At $t = x_i$ for any i we have $g'(x_i) = 0$ for $i = 0, 1, \dots, n$. Hence, $g'(t)$ has $2n + 2$ roots. Using the generalized Rolle's theorem on $g'(t)$ and then following the same pattern as in Theorem 1.3 we get the result. $\square$

Theorem 1.9 gives all the details about the Hermite polynomials but it is computationally expensive as we need to compute the Lagrange polynomials and it's derivatives.

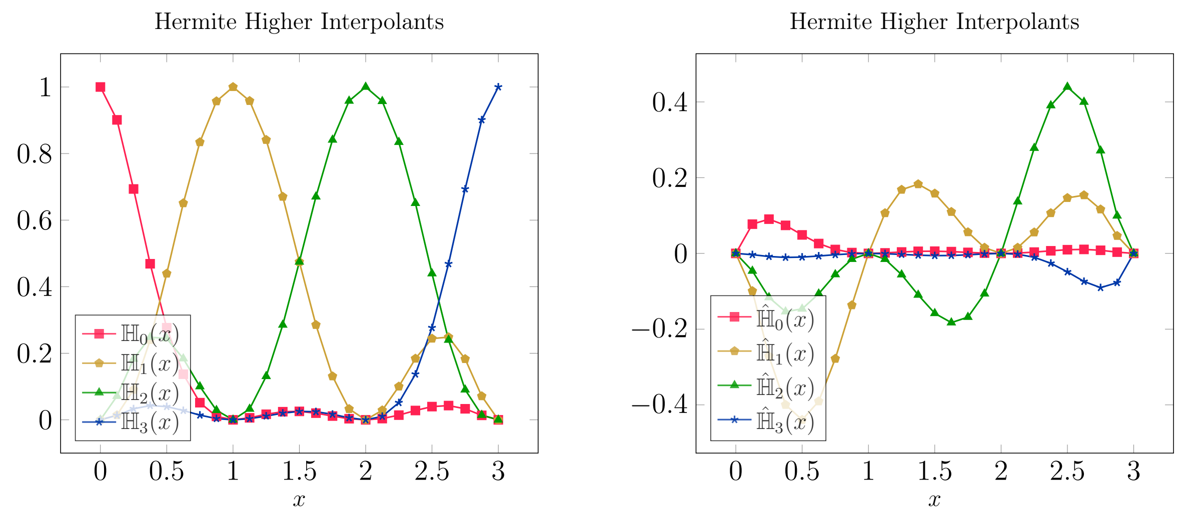

Figure 1.15: Hermite interpolating polynomials defined over $x_i = 0, 1, 2, 3$.

1.4.1 Hermite Polynomials using Divided Difference

For computing the Hermite polynomials using the Newton DD. We will use the relation between the n^{th} DD and the n^{th} derivative of $f(x)$ as in Theorem 1.6.

Suppose we have $(n+1)$ distinct points $\{x_i\}_{i=0}^{n}$, we define a new sequence $\{z_i\}_{i=0}^{2n+1}$ by

$$z_{2i} = z_{2i+1} = x_i \qquad \text{for each} \quad i = 0, 1, \dots, n,$$

i.e., $z_0 = z_1 = x_0$, $z_2 = z_4 = x_1$, and so on. Then we can construct the DD table using these values. Since $z_{2i} = z_{2i+1}$ we cannot define $f[z_{2i}, z_{2i+1}]$. However from Theorem 1.6 we can make a reasonable substitution that

$$f[z_{2i}, z_{2i+1}] = f'(z_{2i}) = f'(x_i).$$

Hence we can use the derivative entries for the undefined DD.

The remaining entries of the DD are defined in the same manner and we get the Hermite polynomial as

$$p^{\mathbb{H}}_{2n+1}(x) = f[z_0] + \sum_{k=1}^{2n+1} f[z_0, z_1, \dots, z_k](x - z_0)(x - z_1) \dots (x - z_{k-1}).$$

For an example let us consider a data set of two points x_0 and x_1. Then the DD table is given by table 1.2.

The algorithm for Hermite interpolation is given in algorithm 4.

z	$f(z)$	1st DD	2nd DD	3rd DD
$z_0 = x_0$	$f[z_0] = f(x_0)$			
		$f[z_0, z_1] = f'(x_0)$		
$z_1 = x_0$	$f[z_1] = f(x_0)$		$f[z_0, z_1, z_2] = \frac{f[z_1,z_2]-f[z_0,z_1]}{z_2-z_0}$	
		$f[z_1, z_2] = \frac{f[z_2]-f[z_1]}{z_2-z_1}$		$f[z_0, z_1, z_2, z_3] = \frac{f[z_1,z_2,z_3]-f[z_0,z_1,z_2]}{z_3-z_0}$
$z_2 = x_1$	$f[z_2] = f(x_1)$		$f[z_1, z_2, z_3] = \frac{f[z_2,z_3]-f[z_1,z_2]}{z_3-z_1}$	
		$f[z_2, z_3] = f'(x_1)$		
$z_3 = x_1$	$f[z_3] = f(x_1)$			

Table 1.2: Divided difference table for two points and the Hermite polynomial

Algorithm 4 Hermite Interpolation

```
Given: Data sets ${(x_i, f_i, f'_i)}_{i=0}^n$, Evaluation point $x_{\text{eval}}$.
Find: Interpolated polynomial $p^{\mathbb{H}}_{2n+1}(x_{\text{eval}})$.
  Step 1: Create $z_i$ and $f(z_i)$ arrays
  Construct $\{z_i\}_{i=0}^{2n+1}$ and $\{f(z_i)\}_{i=0}^{2n+1}$
  for $i = 0$ to $n$ do
    $z_{2i} = z_{2i+1} = x_i, \qquad f(z_{2i}) = f(z_{2i+1}) = f_i$
  end for
  ----------------------------------------
  Step 2: Construct Divided Difference Table
  Initialize DD as a zero matrix of size $(2n + 2) \times (2n + 2)$.
  for $i = 0$ to $2n + 1$ do
    $\mathrm{DD}_{i,0} = f(z_i)$
  end for
  for $j = 1$ to $2n + 1$ do
    for $i = 0$ to $2n + 1 - j$ do
      if $j = 1$ and $i\%2 = 0$ then
        $\mathrm{DD}_{i,j} = f'_{\frac{i}{2}}$
      else
        Compute:
          $$\mathrm{DD}_{i,j} = \frac{\mathrm{DD}_{i+1,j-1} - \mathrm{DD}_{i,j-1}}{z_{i+j} - z_i}.$$
      end if
    end for
  end for
  ----------------------------------------
  Step 3: Evaluate Hermite Polynomial $p^{\mathbb{H}}_{2n+1}(x)$ at $x_{\text{eval}}$
  Initialize $p^{\mathbb{H}}_{2n+1}(x) = \mathrm{DD}_{0,0}$.
  for $k = 1$ to $2n + 1$ do
    Initialize $\omega = 1$.
    for $j = 0$ to $k - 1$ do
      $\omega = \omega \times (x_{\text{eval}} - z_j)$
    end for
    Update the interpolated value:
      $$p^{\mathbb{H}}_{2n+1}(x_{\text{eval}}) = p^{\mathbb{H}}_{2n+1}(x_{\text{eval}}) + \mathrm{DD}_{0,k} \cdot \omega.$$
  end for
  ----------------------------------------
  return $p^{\mathbb{H}}_{2n+1}(x_{\text{eval}})$.
```

1.5 Spline Interpolation

Both Lagrange and Newton interpolation methods suffer from the Runge phenomenon, where oscillations occur at the edges of the interval, especially with high-degree polynomials. This issue arises because these methods rely on a single global polynomial, meaning that every data point influences the entire approximation. This "global approximation" can lead to poor performance for non-uniform or large datasets.

An alternative approach is to divide the interval into smaller sub-intervals and use piecewise polynomial approximation. This strategy, known as local interpolation, reduces the influence of distant data points, resulting in more stable and accurate approximations.

Given a set of points $\{(x_i, f_i)\}_{i=0}^{n}$ we can use piecewise-linear interpolation that consists of joining set of data points using straight lines (see Fig. 1.16). An immediate disadvantage of such an interpolation is that the approximating polynomial is not differentiable at the nodal points which geometrically mean the function is not "smooth".

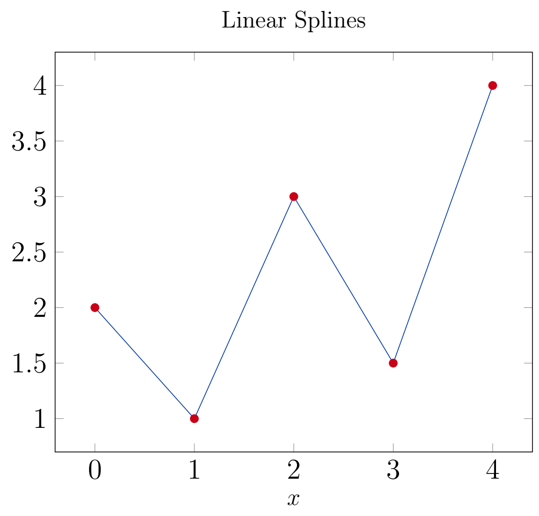

Figure 1.16: Linear spline defined over $x_i = 0, 1, 2, 3, 4$.

To address the limitations of linear interpolation, we use *splines*, which are piecewise polynomials of higher degree. The term "spline" was introduced by Isaac Jacob Schoenberg in the 1930s, inspired by drafting tools called "flat splines". These tools were used to draw smooth curves on paper before the advent of computer-aided design. A spline curve behaves like a flexible beam, ensuring continuity in both slope and curvature.

1.5.1 Cubic Splines

The most common piecewise polynomial approximation uses the cubic polynomials between each successive pair of nodes and is called *cubic spline interpolation* (see Fig. 1.18) .

Figure 1.17: Isaac Jacob Schoenberg: 21 April 1903-21 February 1990

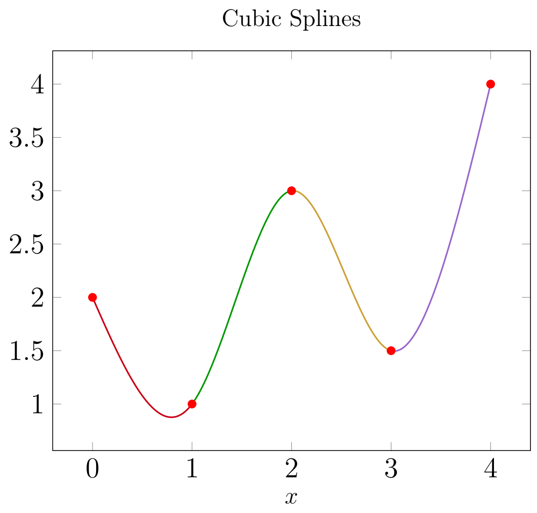

Figure 1.18: Cubic spline defined over $x_i = 0, 1, 2, 3, 4$.

Definition 1.10. Given a function f defined on $[a, b]$ and a set of nodes $a = x_0 < x_1 < \cdots < x_n = b$ (called *knots*), a cubic spline interpolant $p^{\mathbb{S}}$ for f is a function that satisfies the following conditions:

a) $p^{\mathbb{S}}(x)$ is a cubic polynomial, whose restriction on the interval $[x_j, x_j + 1]$ is denoted by $p^{\mathbb{S}}_j(x)$ for each $j = 0, 1, \dots, n-1$.

b) $p^{\mathbb{S}}_j(x_j) = f(x_j)$ and $p^{\mathbb{S}}_j(x_{j+1}) = f(x_{j+1})$ for $j = 0, 1, \dots, n-1$.

c) $p^{\mathbb{S}}_{j+1}(x_{j+1}) = p^{\mathbb{S}}_j(x_{j+1})$ for $j = 0, 1, \dots, n-2$ (implied by **b)**.

d) $p^{\mathbb{S}'}_{j+1}(x_{j+1}) = p^{\mathbb{S}'}_j(x_{j+1})$ for $j = 0, 1, \dots, n-2$.

e) $p^{\mathbb{S}''}_{j+1}(x_{j+1}) = p^{\mathbb{S}''}_j(x_{j+1})$ for $j = 0, 1, \dots, n-2$.

f) One of the following sets of boundary conditions is satisfied:

i) $p^{\mathbb{S}''}(x_0) = p^{\mathbb{S}''}(x_n) = 0$ (*natural* (or free) boundary).

ii) $p^{\mathbb{S}'}(x_0) = f'(x_0)$ and $p^{\mathbb{S}'}(x_n) = f'(x_n)$ (*clamped* boundary).

When the free boundary condition occurs the spline is called *natural spline* . In general clamped boundary conditions lead to more accurate results but it includes the information about the derivative of the function which is not easily available.

Notice that we have n intervals and on each interval we have 4 unknowns. Hence we have a total of $4n$ unknowns.

Construction of Cubic Splines

Let $[a,b]$ be an interval. We divide this interval into n subintervals denoted $[x_j, x_{j+1}]$ for any $j = 0,1,2,\dots,n-1$, then for each subinterval we define a cubic polynomial as

$$p_j^{\mathbb{S}}(x) = a_j + b_j(x-x_j) + c_j(x-x_j)^2 + d_j(x-x_j)^3.$$

Since, $p_j^{\mathbb{S}}(x) = f(x_j)$ we get $a_j = f(x_j)$.

Now from condition **c)** we have

$$\begin{aligned} p_{j+1}^{\mathbb{S}}(x_{j+1}) &= p_j^{\mathbb{S}}(x_{j+1}) \\ a_{j+1} &= a_j + b_j(x_{j+1}-x_j) + c_j(x_{j+1}-x_j)^2 + d_j(x_{j+1}-x_j)^3, \end{aligned}$$

for $j = 0,1,2,\dots,n-2$. Let us denote $x_{j+1}-x_j$ by h_j for $j = 1,2,\dots,n-1$. If we define $a_n = f(x_n)$, then we get the relation

$$a_{j+1} = a_j + b_jh_j + c_jh_j^2 + d_jh_j^3, \qquad \text{for } \ j = 0,1,\dots,n-1. \tag{1.12}$$

We also note that

$$p_j^{\mathbb{S}'}(x) = b_j + 2c_j(x-x_j) + 3d_j(x-x_j)^2.$$

Substituting $x = x_j$, we get $p_j^{\mathbb{S}'}(x_j) = b_j$ for each $j = 0,1,\dots,n-2$. Defining $b_n = p^{\mathbb{S}'}(x_n)$ we get the relation

$$b_{j+1} = b_j + 2c_jh_j + 3d_jh_j^2 \qquad \text{for } \ j = 0,1,\dots,n-1. \tag{1.13}$$

Now $p_j^{\mathbb{S}''}(x) = 2c_j + 6d_j(x-x_j)$ and hence $p_j^{\mathbb{S}''}(x_j) = 2c_j$. Defining $p^{\mathbb{S}''}(x_n) = 2c_n$, from condition **e)** we get

$$2c_{j+1} = 2c_j + 6d_jh_j \qquad \text{for } \ j = 0,1,\dots,n-1. \tag{1.14}$$

Solving for d_j in Eq. (1.14) we get $d_j = \frac{c_{j+1}-c_j}{3h_j}$ and substituting this back in Eq. (1.12) and Eq. (1.13) we get

$$\begin{aligned} a_{j+1} &= a_j + b_jh_j + c_jh_j^2 + \frac{(c_{j+1}-c_j)}{3h_j}h_j^3 \\ &= a_j + b_jh_j + \frac{(2c_j + c_{j+1})}{3}h_j^2. \end{aligned} \tag{1.15}$$

$$b_{j+1} = b_j + 2c_jh_j + h_j(c_{j+1}-c_j). \tag{1.16}$$

From Eq. (1.15) we get for b_j

$$b_j = \frac{a_{j+1}-a_j}{h_j} - \frac{h_j}{3}(2c_j + c_{j+1}). \tag{1.17}$$

Substituting Eq. (1.17) into Eq. (1.16) we get

$$\begin{aligned} \frac{a_{j+2}-a_{j+1}}{h_{j+1}} - \frac{h_{j+1}}{3}(2c_{j+1}+c_{j+2}) &= \frac{a_{j+1}-a_j}{h_j} - \frac{h_j}{3}(2c_j+c_{j+1}) + h_j(c_j+c_{j+1}) \\ \frac{a_{j+2}-a_{j+1}}{h_{j+1}} - \frac{a_{j+1}-a_j}{h_j} &= \frac{c_jh_j}{3} + \frac{2c_{j+1}h_j}{3} + \frac{2c_{j+1}h_{j+1}}{3} + \frac{c_{j+2}h_{j+1}}{3} \\ c_jh_j + 2c_{j+1}(h_j+h_{j+1}) + c_{j+2}h_{j+1} &= \frac{3(a_{j+2}-a_{j+1})}{h_{j+1}} - \frac{3(a_{j+1}-a_j)}{h_j}, \end{aligned}$$

for $j = 0, 1, 2, \ldots, n-2$.

For simplicity we do the shifting of the index by 1. Hence, finally we get the system of equation as

$$c_{j-1}h_{j-1} + 2c_j(h_{j-1} + h_j) + c_{j+1}h_j = \frac{3}{h_j}(a_{j+1} - a_j) - \frac{3}{h_{j-1}}(a_j - a_{j-1}), \tag{1.18}$$

for $j = 1, 2, \ldots, n-1$. The system of equations given by Eq. (1.18) involves the unknown $\{c_i\}_{i=0}^{n}$ as the values of $\{h_j\}_{j=0}^{n}$ and $\{a_j\}_{j=0}^{n}$ are known. Hence, if we can compute $\{c_j\}$ then we can compute $\{b_j\}_{j=0}^{n-1}$ using Eq. (1.17) and $\{d_j\}_{j=0}^{n-1}$ from Eq. (1.14). Hence after these computations we can compute $\{p_j^{\mathbb{S}}(x)\}_{j=0}^{n-1}$. So if Eq. (1.18) has an unique solution then we are done.

Theorem 1.11. *If f is defined at $a = x_0 < x_1 < \cdots < x_n = b$, then f has a unique natural spline interpolant $p^{\mathbb{S}}(x)$ on the nodes $x_0, x_1, \ldots, x_n$, i.e., a spline interpolant that satisfies the natural boundary condition $p^{\mathbb{S}''}(a) = p^{\mathbb{S}''}(b) = 0$.*

Proof. Let us consider $p_0^{\mathbb{S}}(x)$, which is given by $p_0^{\mathbb{S}}(x) = a_0 + b_0(x-x_0) + c_0(x-x_0)^2 + d_0(x-x_0)^3$. Now, $p_0^{\mathbb{S}''}(x) = 2c_0 + 6d_0(x-x_0)$ and at $x = x_0$, we have $p_0^{\mathbb{S}''}(x_0) = 2c_0 = 0$ which implies $c_0 = 0$. Similarly $c_n = 0$.

Let us look at Eq. (1.18)

$$h_{j-1}c_{j-1} + 2(h_{j-1} + h_j)c_j + h_j c_{j+1} = \frac{3}{h_j}(a_{j+1} - a_j) - \frac{3}{h_{j-1}}(a_j - a_{j-1}),$$

for $j = 1, 2, \ldots, n-1$. If we substitute for each j we get a system of equation

$$\mathbf{S}\mathbf{c} = \mathbf{v},$$

where

$$\mathbf{S} = \begin{bmatrix} 1 & 0 & 0 & 0 & \ldots & \ldots & 0 & 0 \\ h_0 & 2(h_0+h_1) & h_1 & 0 & \ldots & \ldots & 0 & 0 \\ 0 & h_1 & 2(h_1+h_2) & h_2 & \ldots & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & \ldots & \ldots & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & 0 & \ldots & \ldots & \ldots & 0 & 0 & 1 \end{bmatrix},$$

$$\mathbf{v} = \begin{bmatrix} 0 \\ \frac{3}{h_1}(a_2 - a_1) - \frac{3}{h_0}(a_1 - a_0) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n - a_{n-1}) - \frac{3}{h_{n-2}}(a_{n-1} - a_{n-2}) \\ 0 \end{bmatrix}, \quad \text{and} \quad \mathbf{c} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix}.$$

We use the following theorem to show that the matrix $\mathbf{S}$ is invertible.

Theorem 1.12. (Strictly Diagonal Dominant Matrix)

A strictly diagonally dominant matrix $\mathbf{A}$ *is nonsingular.*

We notice that our matrix $\mathbf{S}$ is strictly diagonally dominant[3] and hence it is invertible, which leads to an unique solution. □

The algorithm for natural spline interpolation can be found in Algorithm 5 We have a similar result for the clamped spline interpolation.

Theorem 1.13. *If* f *is defined at* $a = x_0 < x_1 < \cdots < x_n = b$ *and differentiable at* a *and* b *then* f *has a unique clamped spline interpolant* $p^{\mathbb{S}}(x)$ *on the nodes* $x_0, x_1, \ldots, x_n$*,i.e., a spline interpolant that satisfies the clamped boundary condition* $p^{\mathbb{S}'}(a) = f'(a)$ *and* $p^{\mathbb{S}'}(b) = f'(b)$*.*

Now we present a result regarding the error bound of the spline interpolation but we will not delve into it's proof as the proof requires a lot of technicalities from Numerical Analysis which is out of scope of this lecture.

Theorem 1.14. *Let* $f \in \mathcal{C}^4[a,b]$ *with* $M = \max_{a\le x\le b} |f^{(4)}(x)|$*.If* $p^{\mathbb{S}}(x)$ *is the unique clamped cubic spline interpolant to* f *with respect to the nodes* $a = x_0 < x_1 < \cdots < x_n = b$ *then for all* $x \in [a,b]$

$$|f(x) - p^{\mathbb{S}}(x)| \le \frac{5M}{384} \max_{0\le j\le n-1} (x_{j+1} - x_j)^4 .$$

A fourth order error bound also exist for the case of natural boundary splines, but they are more difficult to express. An alternative to the natural boundary condition is the *not-a-knot* condition, it states that $p^{\mathbb{S}'''}(x)$ has to be continuous at x_1 and x_{n-1}.

1.5.2 B-Splines

So far, we have focused on a specific type of spline function called cubic splines. A natural question arises: can we generalize this to splines of other degrees? The answer to this is yes. A generalization of the cubic splines is the basis splines or B-splines.

Let $\{x_i\}_{i=0}^n$ be the data points (or knots); then we define the zeroth degree B-spline as

$$B_{j,0}(x) = \begin{cases} 1, & x \in [x_j, x_{j+1}), \\ 0, & \text{else}, \end{cases}$$

[3]**Strictly Diagonally Dominant Matrix**: A matrix $\mathbf{A} = \{a_{ij}\}_{i=1,j=1}^n$ is said to be strictly diagonally dominant if

$$|a_{ii}| > \sum_{j\ne i} |a_{ij}| \qquad \forall i.$$

Algorithm 5 Cubic Natural Spline Interpolation

```
Given: Data sets $\{(x_i, f_i)\}_{i=0}^{n}$, Evaluation point $x_{\text{eval}}$.
Find: Interpolated polynomial $p^{\mathbb{S}}(x_{\text{eval}})$.
  Step 1: Compute $h_i$ arrays
  Construct $\{h_i\}_{i=0}^{n-1}$,
  for $i = 0$ to $n - 1$ do
    $h_i = x_{i+1} - x_i$
  end for
  ------------------------------------------------
  Step 2: Construct S and v
  Initialize $\mathbf{S}$ as a zero matrix of size $(n+1) \times (n+1)$ and $\mathbf{v}$ as $(n+1)$.
  for $i = 0$ to $n$ do
    if $i = 0$ or $i = n$ then
      $\mathbf{S}_{ii} = 1$ and $\mathbf{v}_i = 0$
    else
      $\mathbf{S}_{ii} = 2(h_{i-1} + h_i)$
      $\mathbf{S}_{i,i-1} = h_{i-1}$
      $\mathbf{S}_{i,i+1} = h_i$
      $\mathbf{v}_i = \frac{3}{h_i}(f_{i+1} - f_i) - \frac{3}{h_{i-1}}(f_i - f_{i-1})$
    end if
  end for
  ------------------------------------------------
  Step 3: Solve $\mathbf{Sc} = \mathbf{v}$
  Solve the system $\mathbf{Sc} = \mathbf{v}$ to get coefficient vector $\mathbf{c}$.
  ------------------------------------------------
  Step 4: Locate $x_{\text{eval}}$
  loc = 0
  for $i = 0$ to $n$ do
    if $x_{\text{eval}} \leq x_i$ then
      loc = $i - 1$
      break
    end if
  end for
  ------------------------------------------------
  Step 5: Compute $b_{\text{loc}}$ and $d_{\text{loc}}$
  $b_{\text{loc}} = \frac{f_{\text{loc}+1} - f_{\text{loc}}}{h_{\text{loc}}} - \frac{h_{\text{loc}}}{3}(2\mathbf{c}_{\text{loc}} + \mathbf{c}_{\text{loc}+1})$
  $d_{\text{loc}} = \frac{\mathbf{c}_{\text{loc}+1} - \mathbf{c}_{\text{loc}}}{3h_{\text{loc}}}$
  ------------------------------------------------
  Step 6: Evaluate Spline Polynomial $p^{\mathbb{S}}(x)$ at $x_{\text{eval}}$
  $p^{\mathbb{S}}(x_{\text{eval}}) = f_{\text{loc}} + b_{\text{loc}}(x_{\text{eval}} - x_{\text{loc}}) + \mathbf{c}_{\text{loc}}(x_{\text{eval}} - x_{\text{loc}})^2 + d_{\text{loc}}(x_{\text{eval}} - x_{\text{loc}})^3$
  ------------------------------------------------
  return $p^{\mathbb{S}}(x_{\text{eval}})$
```

for $j = 0, 1, \ldots, n-1$. In Fig. 1.19 we have the zero degree spline $B_{0,0}(x)$ for $x_j = 0$. Higher-degree splines are constructed recursively using lower-degree splines as follows:

$$B_{j,k}(x) = \frac{x - x_j}{x_{j+k} - x_j} B_{j,k-1}(x) + \frac{x_{j+k+1} - x}{x_{j+k+1} - x_{j+1}} B_{j+1,k-1}(x) \qquad k \geq 1.$$

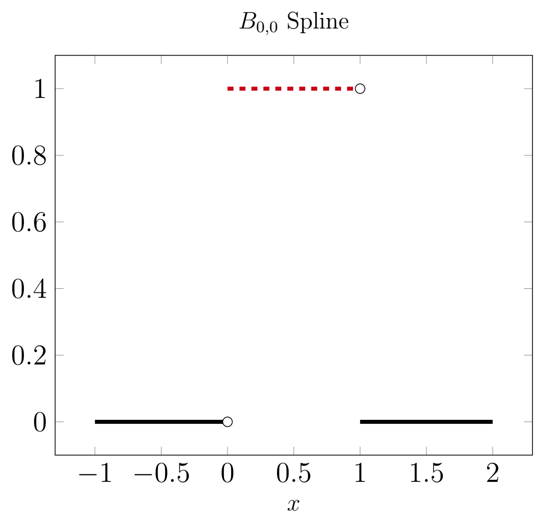

Figure 1.19: Zeroth degree B-spline $B_{0,0}(x)$ for $x_i = 0$.

Although not obvious, one can see that $B_{j,k}(x)$ has one more continuous derivative than $B_{j,k-1}(x)$. Thus while $B_{j,0}(x)$ is discontinuous, $B_{j,1}(x)$ is continuous, $B_{j,2}(x) \in \mathcal{C}^1(\mathbb{R})$, and $B_{j,3}(x) \in \mathcal{C}^2(\mathbb{R})$.

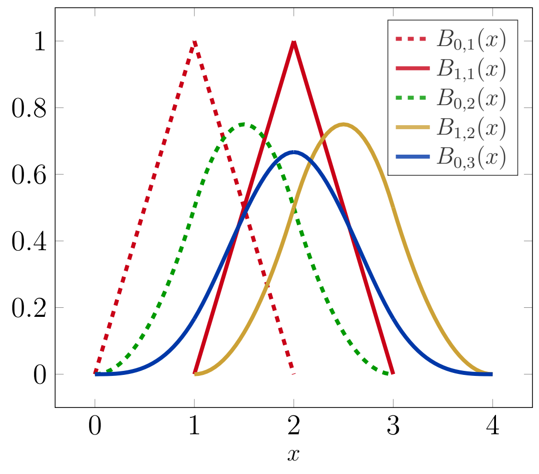

Figure 1.20: Higher degree B-spline polynomial for $x_i = 0$.

As the degree of the B-splines increases, they become more smooth, but the support of $B_{j,k}(x)$ also increases. Based on these results, we can make the following observations:

1. $B_{j,k}(x) \in \mathcal{C}^{k-1}(\mathbb{R})$ (Continuity).

2. $B_{j,k}(x) = 0$ if $x \notin (x_j, x_{j+k+1})$ (Compact Support) [4].

[4]**Compact Support**: Let $f : X \to \mathbb{R}$ be a real-valued function whose domain is an arbitrary set X. The support of f written as $\text{supp}(f)$, is the set of points in X where f is non-zero, i.e., $\text{supp}(f) = \{x \in X : f(x) \neq 0\}$. If $\text{supp}(f)$ is a compact set, then the support is referred to as compact support

3. $B_{j,k}(x) > 0$ for $x \in (x_j, x_{j+k+1})$ (Positivity).

Note: We can notice from Fig. 1.20 that as the degree of the function increases, the support of the function increases as well. Hence, we might get points outside $[x_0, x_n]$. To develop the method, we include additional points beyond the original domain as follows:

$$\cdots < x_{-2} < x_{-1} < x_0 < x_1 < \cdots < x_n < x_{n+1} < \ldots.$$

Let $p_k^{\mathbb{S}}(x)$ denote the spline of piecewise polynomial in $\mathbb{P}_k$. Then we have the following two conditions:

1. $p_k^{\mathbb{S}}(x_i) = f_i$ for $i = 0, 1, \ldots, n$.
2. $p_k^{\mathbb{S}} \in \mathcal{C}^{k-1}[x_0, x_n]$ for $k \geq 1$.

Notice that we have an abuse of notation here. In the previous section we used $p_j^{\mathbb{S}}$ to denote the restriction to $[x_j, x_{j+1}]$, whereas here $p_k^{\mathbb{S}}$ denote a spline of degree k.

Let $c_{j,k}$ denote the unknown coefficients, then

$$p_k^{\mathbb{S}}(x) = \sum_j c_{j,k} B_{j,k}(x).$$

Now the question remains on what values of j the summation applies. For the greatest flexibility, we take j for which

$$B_{j,k}(x) \neq 0 \quad \text{for some } x \in [x_0, x_n].$$

Now, for $k \geq 1, B_{j,k}(x)$ has support of (x_j, x_{j+k+1}) and hence

$$p_k^{\mathbb{S}}(x) = \sum_{j=-k}^{n-1} c_{j,k} B_{j,k}(x), \quad k > 1.$$

The inclusion of negative indices for j arises due to the support of the B-spline at boundary knots, particularly at x_0. For the B-spline $B_{j,k}(x)$ to contribute at x_0, its support must include x_0. Since the support of $B_{j,k}(x)$ spans from x_j to x_{j+k+1}, and the last point of this support is x_1 when considering x_0, we require $j + k + 1 = 1$. Solving for j, this gives $j = -k$, which explains the inclusion of "ghost points" $x_{-1}, x_{-2}, \ldots, x_{-k}$ in the extended knot sequence. At the other boundary, x_n, the support extends back to x_{n-1}, ensuring that $B_{j,k}(x)$ contributes only within the domain of the spline. To include all valid intervals in the original knot sequence, the upper bound for j is $j \leq n - 1$. Thus, the range of j is determined as $-k \leq j \leq n - 1$, ensuring that the spline remains well-defined and accounts for boundary contributions at x_0 and x_n.

Hence we have $n + k$ (include $k = 0$) coefficients (unknowns) that satisfy the $n + 1$ interpolation condition,

$$p_k^{\mathbb{S}}(x_i) = f_i = \sum_{j=-k}^{n-1} c_{j,k} B_{j,k}(x_i) \quad i = 0, 1, \ldots, n.$$

The system becomes underdetermined for higher degrees k, meaning there are more unknowns than equations. As we observed in the cubic interpolation, we might need to impose more conditions to get a system of equations.

Chapter 2

System of Equations

In many applications of science and engineering, solving a system of equations is essential. One prominent example arises in Operations Research, where traffic flow modelling involves solving such systems. The foundational work of Ford and Fulkerson [4] introduced the maximum flow problem, which significantly advanced the theory and applications of system-solving techniques.

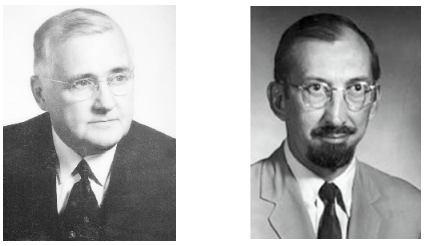

Figure 2.1: Lester Randolph Ford Jr. (23 September 1927–26 February 2017, left) and Delbert Ray Fulkerson (14 August 1924–10 January 1976, right).

Another important application arises in the discretisation of differential equations, a technique widely used in civil, mechanical, and electrical engineering. When differential equations are discretised, the resulting system of equations often takes the form of a *band matrix*[1]. Depending on the choice of polynomial approximation used in the discretisation, the resulting band matrix can be tridiagonal, pentadiagonal, or a more general band matrix. Efficiently solving these systems is crucial to obtaining solutions to the differential equations.

In this chapter, we first introduce *direct methods* for solving systems of equations, followed by iterative methods. Direct methods aim to find the exact solution theoretically in a finite number of steps, though practical computations are subject to round-off errors, which must be carefully managed to ensure accuracy.

[1]**Band Matrix:** A matrix $\{a_{ij}\}_{i,j=1}^n$ is called a band matrix if all elements outside a certain diagonal band are zero. The band is determined by:

$$a_{ij} = 0 \quad \text{if} \quad j < i - k_1 \quad \text{or} \quad j > i + k_2; \quad k_1, k_2 \geq 0,$$

where k_1 and k_2 are the lower and upper bandwidths, respectively. Special cases include diagonal matrices ($k_1 = k_2 = 0$) and tridiagonal matrices ($k_1 = k_2 = 1$).

2.1 Gaussian Elimination

The most fundamental method that one studies in linear algebra for solving the system of equation is the Gaussian elimination. Even though the method has been developed independently in ancient China and early modern Europe, it became popular due to Gauss and hence it was named Gaussian elimination by George Forsythe.

Figure 2.2: Carl Friedrich Gauss: 30 April 1777-23 February 1855.

Suppose we have a system of n equations for n variables of the form

$$\begin{aligned} \mathrm{R}_1 \quad &: a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &&= b_1 \\ \mathrm{R}_2 \quad &: a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &&= b_2 \\ &\qquad\qquad\vdots && \;\vdots \\ \mathrm{R}_n \quad &: a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &&= b_n. \end{aligned} \tag{2.1}$$

In Eq. (2.1) the matrix $\{a_{ij}\}_{i,j=1}^n$ and the vector $\{b_i\}_{i=1}^n$ are given and the $\{x_i\}_{i=1}^n$ are the unknowns.

These system of equations follow certain rules due to which we can transform it into a "simpler" system of equations, i.e., easy to solve. We recall three properties that helps us to achieve this:

1. **Scalar Multiplication:** $\mathrm{R}_i \mapsto \lambda \mathrm{R}_i$ for $\lambda \in \mathbb{R}$.
2. **Scalar Multiplication and Adding:** $\mathrm{R}_i \mapsto \mathrm{R}_i + \lambda \mathrm{R}_j$ for some $j = 1, \ldots, n$ and $j \neq i$.
3. **Transposition:** $\mathrm{R}_i \leftrightarrow \mathrm{R}_j$ for $i \neq j$.

We can represent the system of equation presented in Eq. (2.1) as a $n \times (n+1)$ matrix $[\mathbf{A}, \mathbf{b}]$ and this is called as the *augmented matrix* and is given by

$$[\mathbf{A}, \mathbf{b}] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \ldots & a_{1n} & b_1 \\ a_{21} & a_{22} & \ldots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} & b_n \end{array}\right]. \tag{2.2}$$

The line is shown so as to represent the separation between $\mathbf{A}$ and $\mathbf{b}$. The idea of the Gaussian elimination with backward substitution is to reduce the system provided in Eq. (2.2) to an upper triangular matrix and then perform backward substitution.

Let us for the uniformity of the notation denote b_i by $a_{i,n+1}$ for $i = 1, 2, \dots, n$ then

$$\tilde{\mathbf{A}} = [\mathbf{A}, \mathbf{b}] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & a_{1,n+1} \\ a_{21} & a_{22} & \dots & a_{2n} & a_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & a_{n,n+1} \end{array}\right]. \tag{2.3}$$

Provided $a_{11} \neq 0$ we perform the operations corresponding to

$$\mathrm{R}_j \mapsto \mathrm{R}_j - \frac{a_{j1}}{a_{11}}\mathrm{R}_1 \quad \text{for} \quad j = 2, 3, \dots, n,$$

to eliminate the coefficient x_1 in each of the rows. Once the coefficients of x_1 are cancelled, we do the same for other rows and follow a sequential procedure for $i = 2, 3, \dots, n-1$ and perform the operation

$$\mathrm{R}_j \mapsto \mathrm{R}_j - \frac{a_{ji}}{a_{ii}}\mathrm{R}_i \quad \text{for} \quad j = i+1, i+2, \dots, n.$$

The resulting matrix has the form

$$\tilde{\tilde{\mathbf{A}}} = [\mathbf{A}, \mathbf{b}] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & a_{1,n+1} \\ 0 & \tilde{a}_{22} & \dots & \tilde{a}_{2n} & \tilde{a}_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & \tilde{a}_{nn} & \tilde{a}_{n,n+1} \end{array}\right].$$

This system of equation has the same solution set as Eq. (2.1). But the new system of equation has the form:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= a_{1,n+1} \\ \tilde{a}_{22}x_2 + \dots + \tilde{a}_{2n}x_n &= \tilde{a}_{2,n+1} \\ \vdots \quad &\vdots \quad \vdots \\ \tilde{a}_{nn}x_n &= \tilde{a}_{n,n+1}. \end{aligned}$$

By backward substitution we get

$$x_n = \frac{\tilde{a}_{n,n+1}}{\tilde{a}_{nn}}.$$

Solving the $(n-1)^{\text{th}}$ equation for x_{n-1} and using the value of x_n we get

$$\begin{aligned} \tilde{a}_{n-1,n-1}x_{n-1} + \tilde{a}_{n-1,n}x_n &= \tilde{a}_{n-1,n+1} \\ x_{n-1} &= \frac{\tilde{a}_{n-1,n+1} - \tilde{a}_{n-1,n}x_n}{\tilde{a}_{n-1,n-1}}. \end{aligned}$$

Continuing this process we get

$$x_i = \frac{\tilde{a}_{i,n+1} - \sum_{j=i+1}^{n} \tilde{a}_{ij}x_j}{\tilde{a}_{ii}},$$

for $i = n-1, n-2, \dots, 2, 1$ where for $i = 1$, $\tilde{a}_{1,n+1} = a_{1,n+1}$ and $\tilde{a}_{11} = a_{11}$.

Gaussian elimination can also be seen more precisely by forming a sequence of augmented matrices $\tilde{\mathbf{A}}^{(1)}, \tilde{\mathbf{A}}^{(2)}, \ldots, \tilde{\mathbf{A}}^{(n)}$ where $\tilde{\mathbf{A}}^{(1)}$ is the matrix given in Eq. (2.3) and a general $\tilde{\mathbf{A}}^{(k)}$ matrix for $k = 2, 3, \ldots, n$ is given by

$$\tilde{\mathbf{A}}^{(k)} = \left[\begin{array}{cccccccc|c} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \ldots & a_{1,k-1}^{(1)} & a_{1,k}^{(1)} & \ldots & a_{1,n}^{(1)} & a_{1,n+1}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \ldots & a_{2,k-1}^{(2)} & a_{2,k}^{(2)} & \ldots & a_{2,n}^{(2)} & a_{2,n+1}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & a_{k-1,k-1}^{(k-1)} & a_{k-1,k}^{(k-1)} & \ldots & a_{k-1,n}^{(k-1)} & a_{k-1,n+1}^{(k-1)} \\ 0 & 0 & 0 & \ldots & 0 & a_{k,k}^{(k)} & \ldots & a_{k,n}^{(k)} & a_{k,n+1}^{(k)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 0 & a_{n,k}^{(k)} & \ldots & a_{n,n}^{(k)} & a_{n,n+1}^{(k)} \end{array}\right]. \tag{2.4}$$

where x_{k-1} has been eliminated from $\mathrm{R}_k, \ldots, \mathrm{R}_n$.

In general the matrix entries are given by

$$a_{ij}^{(k)} = \begin{cases} a_{ij}^{(k-1)} & \text{if } \ i = 1, 2, \ldots, k-1 \ \text{ and } \ j = 1, 2, \ldots, n+1, \\ 0 & \text{if } \ i = k, k+1, \ldots, n \ \text{ and } \ j = 1, 2, \ldots, k-1, \\ a_{ij}^{(k-1)} - \frac{a_{i,k-1}^{(k-1)}}{a_{k-1,k-1}^{(k-1)}} a_{k-1,j}^{(k-1)} & \text{if } \ i = k, k+1, \ldots, n \ \text{ and } \ j = k, k+1, \ldots, n+1. \end{cases}$$

This procedure will fail if any of the elements $\{a_{ii}^{(i)}\}$ for $i = 1, 2, \ldots, n$ is zero as

$$\mathrm{R}_i \mapsto \mathrm{R}_i - \frac{a_{i,k}^{(k)}}{a_{k,k}^{(k)}} \mathrm{R}_k$$

cannot be performed or the backward substitution fails.

The system may still have solution but the technique might be altered. For example, consider the augmented matrix

$$\tilde{\mathbf{A}} = \tilde{\mathbf{A}}^{(1)} = \left[\begin{array}{rrrr|r} 1 & -1 & 2 & -1 & -8 \\ 2 & -2 & 3 & -3 & -20 \\ 1 & 1 & 1 & 0 & -2 \\ 1 & -1 & 4 & 3 & 4 \end{array}\right].$$

Performing the operations, $\mathrm{R}_3 \mapsto \mathrm{R}_3 - \mathrm{R}_1$, $\mathrm{R}_4 \mapsto \mathrm{R}_4 - \mathrm{R}_1$, and $\mathrm{R}_2 \mapsto \mathrm{R}_2 - 2\mathrm{R}_1$,

$$\tilde{\mathbf{A}} = \tilde{\mathbf{A}}^{(2)} = \left[\begin{array}{rrrr|r} 1 & -1 & 2 & -1 & -8 \\ 0 & 0 & 1 & -1 & -4 \\ 0 & 2 & -1 & 1 & 6 \\ 0 & 0 & 2 & 4 & 12 \end{array}\right].$$

Here $a_{22}^{(2)}$ is zero and is called the *pivot* element. Hence the procedure cannot proceed. So we search the second column for first non-zero entry after 2nd row. Since, $a_{32}^{(2)} \neq 0$, we perform $\mathrm{R}_2 \leftrightarrow \mathrm{R}_3$ and then proceed.

The above example shows what happens if $a_{kk}^{(k)} = 0$ for some $k = 1, 2, \ldots, n-1$. In this case we follow:

1. The k^{th} column of $\tilde{\mathbf{A}}^{(k-1)}$ is searched from the k^{th} row to the n^{th} row for first non zero entry, $a_{pk}^{(k)} \neq 0$ for $k+1 \leq p \leq n$.
2. Then $\mathrm{R}_p \leftrightarrow \mathrm{R}_k$ is performed to get a temporary matrix $\tilde{\mathbf{A}}^{(k-1)'}$ and then the usual elimination follows.

In the case $a_{pk}^{(k)} = 0$ for each p, then the system does not have a unique solution as two columns are the linearly dependent. Finally, if $a_{nn}^{(n)} = 0$ then the system does not have a unique solution.

The algorithm for the Gaussian elimination is provided in Algorithm 6. Although the algorithm looks like we are creating new matrices $\tilde{\mathbf{A}}^{(i)}$ for $i = 1, 2, \dots, n$ but we can perform all the computation using only one $n \times (n+1)$ matrix for storage.

2.1.1 Computational Complexity

Now we look at the computational complexity of the Gaussian elimination. Generally time taken to perform a multiplication or division is generally more than addition or subtraction. Hence, we count these operations separately.

The arithmetic operations happens in Step 2.3:

1. **Computation of m_{ki}:** Requires division and $(n-i)$ operations.
2. **Multiplication of $m_{ki}\mathrm{R}_i$:** This multiplication happens with the non-zero entries of R_i which is $(n-i) \times (n-i+1)$ as non zero entries is given by $(n-i+1)$ in the R_i^{th} row.
3. **Subtraction for $\mathrm{R}_k - m_{ki}\mathrm{R}_i$:** These will also $(n-i+1) \times (n-i)$ as we subtract the non-zero entries.

The first two are multiplication and division and the last one is addition/subtraction.

Multiplication/Division Complexity

Now, $(n-i) + (n-i) \times (n-i+1) = (n-i) \times (n-i+2)$. Summing i from 1 to $n-1$ we get

$$\begin{aligned}
\sum_{i=1}^{n-1}(n-i)(n-i+2) &= \sum_{i=1}^{n}\left(n^2 - in + 2n - in + i^2 - 2i\right) \\
&= \sum_{i=1}^{n-1}\left(n^2 - 2in + 2n + i^2 - 2i\right) \\
&= \sum_{i=1}^{n-1}(n-i)^2 + 2\sum_{i=1}^{n-1}(n-i) \\
&= \sum_{i=1}^{n-1} i^2 + 2\sum_{i=1}^{n-1} i \\
&= \frac{(n-1)n(2n-1)}{6} + \frac{2n(n-1)}{2} = \frac{2n^3 + 3n^2 - 5n}{6}.
\end{aligned}$$

Algorithm 6 Gauss Elimination Algorithm

```
Given: Matrix $\mathbf{A}$, right hand side $\mathbf{b}$ and dimension $n$.
Find: Solution $\mathbf{x}$.
  Step 1:Create Augmented Matrix $\tilde{\mathbf{A}}$
  Initialize $\tilde{\mathbf{A}}$ as a zero matrix of size $n \times (n+1)$
  for $i = 1$ to $n$ do
      for $j = 1$ to $n+1$ do
          if $j \leq n$ then
              $\tilde{\mathbf{A}}_{ij} = \mathbf{A}_{ij}$
          else
              $\tilde{\mathbf{A}}_{ij} = \mathbf{b}_i$
          end if
      end for
  end for
  ----------------------------------------
  Step 2: Reduce the matrix to Row-Echelon form
  for $i = 1$ to $n-1$ do
      ------------------------------------
      Step 2.1: Check Pivot
      Initialize $p = -1$
      for $q = i$ to $n$ do
          if $\tilde{\mathbf{A}}_{qi} \neq 0$ then
              $p = q$
              break
          end if
      end for
      if $p = -1$ then
          Output("No Unique Solution")
          exit()
      end if
      ------------------------------------
      Step 2.2: Exchange Rows $\mathrm{R}_i \leftrightarrow \mathrm{R}_p$
      if $p \neq i$ then
          $\text{temp} = 0$
          for $j = 1$ to $n+1$ do
              $\text{temp} = \tilde{\mathbf{A}}_{ij}$
              $\tilde{\mathbf{A}}_{ij} = \tilde{\mathbf{A}}_{pj}$
              $\tilde{\mathbf{A}}_{pj} = \text{temp}$
          end for
      end if
      ------------------------------------
      Step 2.3: Matrix Reduction
      $m_{ki} = 0$
      for $k = i+1$ to $n$ do
          $m_{ki} = \tilde{\mathbf{A}}_{ki}/\tilde{\mathbf{A}}_{ii}$
          for $j = i$ to $n+1$ do
              $\tilde{\mathbf{A}}_{kj} = \tilde{\mathbf{A}}_{kj} - m_{ki}\tilde{\mathbf{A}}_{ij}$
          end for
      end for
  end for
  ----------------------------------------
  Step 3: Check for no Solution
  if $\tilde{\mathbf{A}}_{nn} = 0$ then
      Output("No Unique Solution")
      exit()
  end if
  ----------------------------------------
  Step 4: Backward Substitution
  Initialize $x$ as a vector of size $n$
  $x_n = \frac{a_{n,n+1}}{a_{nn}}$
  for $i = n-1$ to $1$ do
      $\text{sum} = 0$
      for $j = i+1$ to $n$ do
          $\text{sum} = \text{sum} + \tilde{\mathbf{A}}_{ij} x_j$
      end for
      $x_i = \frac{\tilde{\mathbf{A}}_{i,n+1} - \text{sum}}{\tilde{\mathbf{A}}_{ii}}$
  end for
  ----------------------------------------
  return $\{x_i\}_{i=1}^{n}$
```

In the above equation we have used the basic identities of summation, namely $\sum_{i=1}^n i^2$ and $\sum_{i=1}^n i$.

Addition/Subtraction Complexity

$$\begin{aligned}
\sum_{i=1}^{n-1}(n-i)(n-i+1) &= \sum_{i=1}^{n-1}\left(n^2-2ni+i^2+n-i\right) \\
&= \sum_{i=1}^{n-1}(n-i)^2+\sum_{i=1}^{n-1}(n-i) \\
&= \sum_{i=1}^{n-1}i^2+\sum_{i=1}^{n-1}i \\
&= \frac{n(n-1)(2n-1)}{6}+\frac{n(n-1)}{2}=\frac{n^3-n}{3}.
\end{aligned}$$

Hence we notice that in Step 2.3 we require $\mathcal{O}(n^3)$ operations.

The next step that require arithmetic operations are the ones in backward substitution, i.e, Step 4. First is in the computation of x_n which requires one division. For the computation of rest of the $\{x_i\}$ we need $(n-i)$ multiplications and one division for each i and $(n-i-1)$ addition for each summation followed by one subtraction.

Multiplication/Division Complexity

$$\begin{aligned}
1+\sum_{i=1}^{n-1}((n-i)+i) &= 1+\left(\sum_{i=1}^{n-1}(n-i)\right)+n-1 \\
&= n+\sum_{i=1}^{n-1}(n-i) \\
&= n+\sum_{i=1}^{n-1}i \\
&= n+\frac{n(n-1)}{2}=\frac{n^2+n}{2}.
\end{aligned}$$

Addition/Subtraction Complexity

$$\sum_{i=1}^{n-1}((n-i-1)+1) = \sum_{i=1}^{n-1}(n-i)=\frac{n^2-n}{2}.$$

Hence in total we require

$$\frac{2n^3+3n^2-5n}{6}+\frac{n^2+n}{2}=\frac{n^3}{3}+n^2-\frac{n}{3},$$

operations for multiplication and division; and

$$\frac{n^3 - n}{3} + \frac{n^2 - n}{2} = \frac{2n^3 + 3n^2 - 5n}{6},$$

for addition and subtraction. Hence we have $\mathcal{O}(n^3)$ computational complexity.

2.1.2 Gauss-Jordan Algorithm

Wilhelm Jordan was a geodesist who extended the basic Gaussian elimination to achieve a full row-reduced echelon form of a matrix. Do not confuse Wilhelm Jordan with Camille Jordan (who gave us Jordan Curve theorem and Jordan Canonical form).

Figure 2.3: Wilhelm Jordan: 1 March 1842-17 April 1899.

This method is a variation of Gaussian elimination where the variable x_i is not only removed from $\mathrm{R}_{i+1}, \mathrm{R}_{i+2}, \ldots, \mathrm{R}_n$ but also from $\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_{i-1}$. Upon this reduction the augmented matrix looks like

$$[\mathbf{A}, \mathbf{b}] = \left[\begin{array}{cccc|c} a_{11}^{(1)} & 0 & \ldots & 0 & a_{1,n+1}^{(1)} \\ 0 & a_{22}^{(2)} & \ldots & 0 & a_{2,n+1}^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & a_{nn}^{(n)} & a_{n,n+1}^{(n)} \end{array}\right].$$

Then the solution can be easily obtained using

$$x_i = \frac{a_{i,n+1}^{(i)}}{a_{ii}^{(i)}}, \qquad \text{for} \quad i = 1, 2, \ldots, n.$$

The Gauss-Jordan algorithm is presented in Algorithm 7.

2.2 Matrix Factorisation

Like polynomial interpolation, which was the basis for developing more efficient algorithms such as Lagrange and Newton divided differences, Gaussian elimination is the foundation for more advanced topics.

Gaussian elimination consists of two steps: the row-reduction step and backward substitution. The former has a computational complexity of $\mathcal{O}(n^3)$, while the latter requires only $\mathcal{O}(n^2)$. This means that if we have a triangular matrix, solving the system requires only $\mathcal{O}(n^2)$ operations.

Algorithm 7 Gauss Jordan Algorithm

```
Given: Matrix $\mathbf{A}$, right hand side $\mathbf{b}$ and dimension $n$.
Find: Solution $\mathbf{x}$.
  Step 1:Create Augmented Matrix $\tilde{\mathbf{A}}$
  Initialize $\tilde{\mathbf{A}}$ as a zero matrix of size $n \times (n+1)$
  for $i = 1$ to $n$ do
      for $j = 1$ to $n+1$ do
          if $j \leq n$ then
              $\tilde{\mathbf{A}}_{ij} = \mathbf{A}_{ij}$
          else
              $\tilde{\mathbf{A}}_{ij} = \mathbf{b}_i$
          end if
      end for
  end for
  ------------------------------------------------------------
  Step 2: Reduce the matrix to Row-Echelon form
  for $i = 1$ to $n-1$ do
      --------------------------------------------------------
      Step 2.1: Check Pivot
      Initialize $p = -1$
      for $q = i$ to $n$ do
          if $\tilde{\mathbf{A}}_{qi} \neq 0$ then
              $p = q$
              break
          end if
      end for
      if $p = -1$ then
          Output("No Unique Solution")
          exit()
      end if
      --------------------------------------------------------
      Step 2.2: Exchange Rows $\mathrm{R}_i \leftrightarrow \mathrm{R}_p$
      if $p \neq i$ then
          temp = 0
          for $j = 1$ to $n+1$ do
              temp = $\tilde{\mathbf{A}}_{ij}$
              $\tilde{\mathbf{A}}_{ij} = \tilde{\mathbf{A}}_{pj}$
              $\tilde{\mathbf{A}}_{pj}$ = temp
          end for
      end if
      --------------------------------------------------------
      Step 2.3: Matrix Reduction
      $m_{ki} = 0$
      for $k = i+1$ to $n$ do
          if $k = i$ then
              continue
          else
              $m_{ki} = \tilde{\mathbf{A}}_{ki}/\tilde{\mathbf{A}}_{ii}$
              for $j = i$ to $n+1$ do
                  $\tilde{\mathbf{A}}_{kj} = \tilde{\mathbf{A}}_{kj} - m_{ki}\tilde{\mathbf{A}}_{ij}$
              end for
          end if
      end for
  end for
  ------------------------------------------------------------
  Step 3: Check for no Solution
  if $\tilde{\mathbf{A}}_{nn} = 0$ then
      Output("No Unique Solution")
      exit()
  end if
  ------------------------------------------------------------
  Step 4: Backward Substitution
  Initialize $x$ as a zero vector of size $n$
  for $i = 1$ to $n$ do
      $x_i = \frac{a_{i,n+1}}{a_{ii}}$
  end for
  ------------------------------------------------------------
  return $\{x_i\}_{i=1}^{n}$
```

2.2.1 LU Decomposition

Suppose that we have $\mathbf{A} = \mathbf{LU}$, meaning that $\mathbf{A}$ has been factored into a lower triangular matrix $(\mathbf{L})$[2] and an upper triangular matrix $(\mathbf{U})$[3]. Then, solving $\mathbf{Ax} = \mathbf{b}$ can be done in two steps:

- Solve $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$.
- Solve $\mathbf{Ux} = \mathbf{y}$ for $\mathbf{x}$.

Both steps require only $\mathcal{O}(n^2)$ operations.

Although several mathematicians introduced LU decomposition, the Polish mathematician Tadeusz Banachiewicz is credited with generalizing the method for arbitrary matrices.

Figure 2.4: Tadeusz Banachiewicz: 13 February 1882 - 17 November 1954.

LU decomposition reduces an $\mathcal{O}(n^3/3)$ problem to an $\mathcal{O}(2n^2)$ problem. This reduction is useful but comes at a cost: the factorization of $\mathbf{A}$ into $\mathbf{L}$ and $\mathbf{U}$ itself requires $\mathcal{O}(n^3/3)$ operations. However, once computed, the factorization can be stored and reused for multiple right-hand-side vectors $\mathbf{b}$.

To proceed with LU decomposition, we assume that $\mathbf{Ax} = \mathbf{b}$ can be solved using Gaussian elimination without row pivoting, i.e., $a_{ii}^{(i)} \neq 0$ for $i = 1, 2, \ldots, n$.

The first step in Gaussian elimination consists of performing, for each $j = 2, 3, \ldots, n$,

$$\mathrm{R}_j \mapsto \mathrm{R}_j - m_{j1}\mathrm{R}_1, \qquad \text{where} \quad m_{j1} = \frac{a_{j1}^{(1)}}{a_{11}^{(1)}}.$$

An equivalent way of viewing this is by multiplying $\mathbf{A}$ on the left by the matrix $\mathbf{M}^{(1)}$, where

$$\mathbf{M}^{(1)} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ -m_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ -m_{n1} & 0 & \cdots & 1 \end{bmatrix}.$$

This is called the *first Gaussian transformation matrix.* The product of this matrix with $\mathbf{A}$ is denoted by $\mathbf{A}^{(2)}$, so that

$$\mathbf{A}^{(2)} = \mathbf{M}^{(1)}\mathbf{A}.$$

[2]**Lower Triangular Matrix:** A matrix $\mathbf{A} = \{a_{ij}\}_{i,j=1}^{n}$ is said to be lower triangular if $a_{ij} = 0$ for $i > j$.
[3]**Upper Triangular Matrix:** A matrix $\mathbf{A} = \{a_{ij}\}_{i,j=1}^{n}$ is said to be upper triangular if $a_{ij} = 0$ for $j > i$.

Similarly, the right-hand side vector is updated as

$$\mathbf{b}^{(2)} = \mathbf{M}^{(1)}\mathbf{b}.$$

Next, we construct $\mathbf{M}^{(2)}$ by replacing the subdiagonal entries in the second column of the identity matrix with the negative of the multipliers

$$m_{j2} = \frac{a_{j2}^{(2)}}{a_{22}^{(2)}}.$$

This process continues until we obtain an upper triangular matrix $\mathbf{A}^{(n)}$, given by

$$\mathbf{A}^{(n)} = \mathbf{M}^{(n-1)}\mathbf{M}^{(n-2)}\cdots\mathbf{M}^{(1)}\mathbf{A}.$$

At this point, we define $\mathbf{U} = \mathbf{A}^{(n)}$ as the upper triangular matrix in the LU factorization.

To compute the lower triangular matrix $\mathbf{L}$, we note that the inverse of each $\mathbf{M}^{(k)}$ matrix is given by

$$\mathbf{L}^{(k)} = \left[\mathbf{M}^{(k)}\right]^{-1} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & m_{k+1,k} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & m_{n,k} & 0 & \cdots & 1 \end{bmatrix}.$$

The lower triangular matrix $\mathbf{L}$ is then obtained as

$$\mathbf{L} = \mathbf{L}^{(1)}\mathbf{L}^{(2)}\cdots\mathbf{L}^{(n-1)}.$$

Since each $\mathbf{L}^{(k)}$ is the inverse of $\mathbf{M}^{(k)}$, we confirm that

$$\mathbf{LU} = \mathbf{A}.$$

Theorem 2.1. (Doolittle LU Decomposition) *If Gaussian elimination can be performed on the system* $\mathbf{Ax} = \mathbf{b}$ *without row interchanges, then the matrix* $\mathbf{A}$ *can be factored as* $\mathbf{A} = \mathbf{LU}$, *where*

$$m_{ji} = \frac{a_{ji}^{(i)}}{a_{ii}^{(i)}},$$

and

$$\mathbf{U} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn}^{(n)} \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ m_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & 1 \end{bmatrix}.$$

The above factorization is the Doolittle method, where $\mathbf{L}$ has ones on its diagonal. Alternatively, if the ones are placed on the diagonal of $\mathbf{U}$, the technique is called Crout's LU decomposition.

Once the LU factorization is obtained, the system $\mathbf{Ax} = \mathbf{LUx}$ is solved efficiently by first computing $\mathbf{y}$ from $\mathbf{Ly} = \mathbf{b}$ using forward substitution and then solving $\mathbf{Ux} = \mathbf{y}$ using backward substitution.

It is important to note that not all square matrices have an LU factorization. For example, the matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

has no LU factorization. Suppose it did; then there would exist $\mathbf{L}$ and $\mathbf{U}$ such that $\mathbf{A} = \mathbf{LU}$. However, this would lead to a contradiction, as one of the factors would necessarily be singular while $\mathbf{A}$ is not.

Next, we note that the LU decomposition is not unique.

Theorem 2.2. *If a matrix has an LU decomposition, then it is not unique.*

Proof. Let $\mathbf{A}$ have an LU decomposition, i.e., $\mathbf{A} = \mathbf{LU}$. Then, we can write

$$\begin{aligned} \mathbf{A} &= \mathbf{LU} \\ &= \mathbf{LDD}^{-1}\mathbf{U} \\ &= (\mathbf{LD})\left(\mathbf{D}^{-1}\mathbf{U}\right), \end{aligned}$$

where $\mathbf{D}$ is any diagonal matrix. Since $\mathbf{LD}$ remains lower triangular and $\mathbf{D}^{-1}\mathbf{U}$ is still upper triangular, we obtain infinitely many LU decompositions of $\mathbf{A}$ by varying $\mathbf{D}$. □

PLU Decomposition

So far, we have assumed that LU decomposition is applicable to systems of equations that do not require pivoting. However, in general, pivoting is necessary. To introduce LU decomposition with pivoting, we first define the permutation matrix.

Definition 2.3. A *permutation matrix* $\mathbf{P} = \{p_{ij}\}_{i,j=1}^{n}$ is an $n \times n$ matrix obtained by rearranging the rows of the identity matrix.

For example, the matrix

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

is a 3×3 permutation matrix where the second and third rows are interchanged. For any 3×3 matrix $\mathbf{A}$, multiplying by $\mathbf{P}$ on the left swaps these two rows:

$$\mathbf{PA} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{31} & a_{32} & a_{33} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}.$$

Let $k_1, k_2, \ldots, k_n$ be a permutation of $1, 2, \ldots, n$. The permutation matrix $\mathbf{P}$ is then defined as:

$$p_{ij} = \begin{cases} 1 & \text{if } \ j = k_i, \\ 0 & \text{otherwise.} \end{cases}$$

This satisfies the following properties:

1. $\mathbf{PA}$ permutes the rows of $\mathbf{A}$:

$$\mathbf{PA} = \begin{bmatrix} a_{k_1 1} & a_{k_1 2} & \cdots & a_{k_1 n} \\ a_{k_2 1} & a_{k_2 2} & \cdots & a_{k_2 n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k_n 1} & a_{k_n 2} & \cdots & a_{k_n n} \end{bmatrix}.$$

2. The inverse of a permutation matrix exists and is given by $\mathbf{P}^{-1} = \mathbf{P}^{\top}$.

In the previous section, we saw that for any nonsingular matrix $\mathbf{A}$, the linear system $\mathbf{Ax} = \mathbf{b}$ can be solved using Gaussian elimination with row interchanges. If the required row interchanges are known beforehand, we can apply them initially, allowing us to use LU decomposition without further row swaps. That is, for any nonsingular matrix $\mathbf{A}$, there exists a permutation matrix $\mathbf{P}$ such that the system

$$\mathbf{PAx} = \mathbf{Pb}$$

can be solved without row interchanges. Consequently, we can factorize $\mathbf{PA}$ as

$$\mathbf{PA} = \mathbf{LU}.$$

Since $\mathbf{P}$ is a permutation matrix, we have $\mathbf{P}^{-1} = \mathbf{P}^{\top}$, which implies

$$\mathbf{A} = \left(\mathbf{P}^{\top}\mathbf{L}\right)\mathbf{U}.$$

While $\mathbf{U}$ remains upper triangular, the matrix $\mathbf{P}^{\top}\mathbf{L}$ may not necessarily be lower triangular unless $\mathbf{P} = \mathbf{I}$.

Based on this, we establish the following lemma.

Lemma 2.4. *Let $\mathbf{A}$ be an $n \times n$ matrix. Then, there exists a permutation matrix $\mathbf{P}$ such that $\mathbf{PA}$ has an LU decomposition, i.e., $\mathbf{PA} = \mathbf{LU}$.*

The next theorem addresses the uniqueness of the LU decomposition.

Theorem 2.5. *Let $\mathbf{A}$ be an $n \times n$ matrix, and let $\mathbf{P}$ be an $n \times n$ permutation matrix such that $\mathbf{PA}$ has an LU decomposition. If $\mathbf{A}$ is invertible, then there exists a unique $n \times n$ lower triangular matrix $\mathbf{L}$ with all diagonal entries equal to 1, and a unique $n \times n$ upper triangular matrix $\mathbf{U}$ such that*

$$\mathbf{PA} = \mathbf{LU}.$$

Proof. The existence of the LU decomposition follows from Lemma 2.4. We now prove the uniqueness.

Suppose $\mathbf{L}$ is not unit lower triangular. Then, we can express the decomposition as

$$\mathbf{PA} = \mathbf{LU}.$$

Rewriting,

$$\mathbf{PA} = \mathbf{LD}^{-1}\mathbf{DU},$$

where $\mathbf{D}$ is a diagonal matrix whose diagonal entries match those of $\mathbf{L}$. Since $\mathbf{A}$ is invertible, $\mathbf{L}$ is also invertible, ensuring that $\mathbf{D}^{-1}$ exists. Defining

$$\mathbf{L}_1 = \mathbf{LD}^{-1}, \quad \mathbf{U}_1 = \mathbf{DU},$$

we obtain a new factorization with $\mathbf{L}_1$ as a unit lower triangular matrix and $\mathbf{U}_1$ as an upper triangular matrix:

$$\mathbf{PA} = \mathbf{L}_1\mathbf{U}_1.$$

Now, suppose there exists another decomposition:

$$\mathbf{PA} = \mathbf{L}_2\mathbf{U}_2,$$

where $\mathbf{L}_2$ is also unit lower triangular. Then, we equate the two decompositions:

$$\mathbf{L}_1\mathbf{U}_1 = \mathbf{L}_2\mathbf{U}_2.$$

Since $\mathbf{A}$ is invertible, both $\mathbf{L}_1$ and $\mathbf{P}$ are invertible, implying that $\mathbf{U}_1 = \mathbf{L}_1^{-1}\mathbf{PA}$ is also invertible.

Thus, we obtain

$$\mathbf{L}_2^{-1}\mathbf{L}_1 = \mathbf{U}_2\mathbf{U}_1^{-1}. \tag{2.5}$$

Since:

1. The inverse of a lower (upper) triangular matrix is lower (upper) triangular.
2. The product of lower (upper) triangular matrices remains lower (upper) triangular.

it follows that $\mathbf{L}_2^{-1}\mathbf{L}_1$ is lower triangular, and $\mathbf{U}_2\mathbf{U}_1^{-1}$ is upper triangular. Since $\mathbf{L}_2^{-1}\mathbf{L}_1$ is also unit diagonal, the only possibility is

$$\mathbf{L}_2^{-1}\mathbf{L}_1 = \mathbf{I} \Rightarrow \mathbf{L}_2 = \mathbf{L}_1.$$

Similarly, we obtain $\mathbf{U}_1 = \mathbf{U}_2$, proving uniqueness. □

Algorithm 8 LU Decomposition with Partial Pivoting

```
Given: Matrix $\mathbf{A}$ of size $n \times n$.
Find: Matrices $\mathbf{L}$, $\mathbf{U}$, and $\mathbf{P}$ such that $\mathbf{PA} = \mathbf{LU}$.
  Step 1: Initialize Matrices
  Initialize $\mathbf{L}$ as an $n \times n$ identity matrix.
  Initialize $\mathbf{P}$ as an $n \times n$ identity matrix.
  Initialize $\mathbf{U}$ as $\mathbf{A}$.
  ----------------------------------------
  Step 2: Perform LU Decomposition
  for $i = 1$ to $n - 1$ do
    ----------------------------------------
    Step 2.1: Check Pivot
    Initialize $p = -1$
    for $q = i$ to $n$ do
      if $\mathbf{U}_{qi} \neq 0$ then
        $p = q$
        break
      end if
    end for
    if $p = -1$ then
      Output("Matrix is singular but the LU decomposition still exists!")
      continue
    end if
    ----------------------------------------
    Step 2.2: Exchange Rows for P and U $\mathrm{R}_i \leftrightarrow \mathrm{R}_p$
    if $p \neq i$ then
      temp1 = 0; temp2 = 0.
      for $j = 1$ to $n + 1$ do
        temp1 = $\mathbf{P}_{ij}$; temp2 = $\mathbf{U}_{ij}$
        $\mathbf{P}_{ij} = \mathbf{P}_{pj}$; $\mathbf{A}_{ij} = \mathbf{A}_{pj}$.
        $\mathbf{P}_{pj}$ = temp1; $\mathbf{A}_{pj}$ = temp2.
      end for
    end if
    ----------------------------------------
    Step 2.3: Matrix Reduction
    for $k = i + 1$ to $n$ do
      $m_{ki} = \mathbf{U}_{ki}/\mathbf{U}_{ii}$
      $\mathbf{L}_{ki} = m_{ki}$
      for $j = i$ to $n$ do
        $\mathbf{U}_{kj} = \mathbf{U}_{kj} - m_{ki}\mathbf{U}_{ij}$
      end for
    end for
  end for
  ----------------------------------------
  return $\mathbf{L}, \mathbf{U}, \mathbf{P}$
```

Chapter 3

Computing

The word **computing** has different meanings based on context and definition. According to Wikipedia:

> *Computing is any goal-oriented activity requiring, benefitting from, or creating computing machinery.*

This definition creates a recursive loop, as it uses "computing" to define itself. To break this loop, let us explore what a **computer** is. Wikipedia defines it as:

> *A computer is a machine that can be programmed to automatically carry out sequences of arithmetic or logical operations (computation).*

Here, two terms stand out: *arithmetic* and *logical*. These are fundamental concepts that mathematicians are familiar with. Thus, we have some basic understanding of computing.

In this course, we will not delve deeply into the workings of a computer. It is assumed that students are familiar with components like the keyboard, mouse (or trackpad), CPU, and monitor. For a refresher, see Figure 3.1.

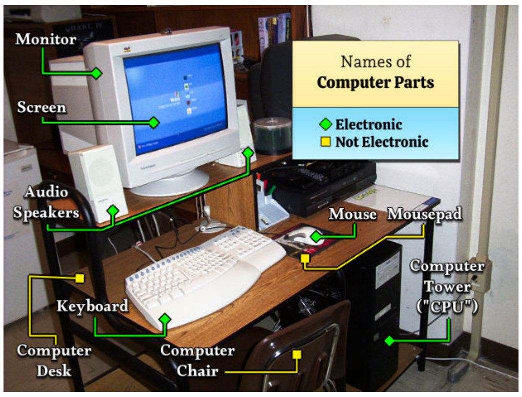

Figure 3.1: Parts of a computer from the early 2000s.

The primary aim of this course is to teach the fundamentals of programming using the Python language. This is not an *Introduction to Python* course. Instead, the focus is on how to think about coding problems, understand common paradigms across languages, adopt good coding practices, maintain code, and debug effectively.

3.1 Good Practices in Coding

Coding is an art form, and like any art, its true audience is those who interact with it (in this case, the users of the code). A good codebase should be **well-documented**. Comments should explain the purpose of each function or variable. The beginning of the code should clearly state its objective.

3.1.1 Variable Initialization and Naming

Variables and functions should have **descriptive names**. For example, if a variable represents the number of oranges, naming it `n_oranges` is much clearer than simply using `n`. Additionally, variables should be initialized to prevent the use of garbage values.

Indentation of the conditional and iterative statements are important as it helps to differentiate different loops (or if-else statements).

In Python, indentation is mandatory, making this practice less error-prone. However, for languages like C++, proper formatting and indentation are crucial. Below we give an example of bad coding vs good coding in C++.

Bad Example:

```cpp
int n_oranges;
for (int i = 0; i < 10; i++)
{
std::cout << i;
for(int j=0;j<2;j++
{
std::cout<<i+j;
}
std::cout<<n_oranges;
}
```

Good Example:

```cpp
int n_oranges = 0;
for (int i = 0; i < 10; i++)
{
    std::cout << i;
    for (int j = 0; j < 2; j++)
    {
        std::cout << i + j;
    }
    std::cout<<n_oranges;
}
```

3.1.2 Reusability and Modularity

Code should be **reusable** and **modular** . For instance, consider a program that computes the Taylor series of a function. Instead of hardcoding the factorial computation in the main function, create a separate function for it. This is called *modularity*. This approach makes the code reusable. If another project requires the computation of ${}^2\mathrm{C}_k$, the factorial function can be reused without rewriting it.

3.2 Testing and Continuous Integration

Testing is a critical aspect of programming to ensure correctness and reliability. Continuous integration ensures that code changes do not break existing functionality.

After writing code, how do we know if it is correct? One effective approach is to verify the solution produced by the code against a pre-existing known solution. For example, if we write code to find the roots of a function, we can test its accuracy by using values with known solutions, such as $x^2 = 2$.

It is always advisable to run the code on multiple test cases to validate its correctness. Once the code is verified, we can create specific *test routines* to ensure its reliability in various scenarios.

3.3 Introduction to Computing Using Python

This course covers various aspects of computing, but we begin with the basics to build a strong foundation.

3.3.1 Variables

In Python, there are three commonly used variable types:

- `int`: Represents integers.
- `str`: Represents strings (text).
- `float`: Represents floating-point numbers (decimals).

Python does not require explicit variable declaration; you can assign values directly. For example:

```
n_oranges = 10  # An integer
price_oranges = 10.4  # A floating-point number
```

To define strings, use double quotes ("):

```
fruit = "oranges"  # A string
```

To check the type of a variable, use the `type()` function:

```
print(type(n_oranges))  # Output: <class 'int'>
```

Another important variable type is the **list**, which can contain multiple values of different types:

```
list_fruits = [n_oranges, price_oranges, fruit]  # A list with mixed
    types
```

While there are more variable types in Python, these four are essential for now.

Note: The `print()` function is used to display information. We will explore more advanced printing techniques later.

3.3.2 Arithmetic Operations

Arithmetic operations are fundamental in any programming language. Python provides the following operations:

Operation	Description	Example
$+$	Addition	$2 + 2 = 4$
$-$	Subtraction	$6 - 2 = 4$
$*$	Multiplication	$2 * 2 = 4$
$/$	Division	$2/2 = 1$
$**$	Exponentiation	$2 * *2 = 4$
$==$	Equality comparison	$2 == 2$
$\%$	Modulus (remainder)	$3\%2 = 1$

Table 3.1: Arithmetic Operations in Python

Additionally, the $! =$ operator means "not equal to," as in $3 \neq 2$.

These operations allow us to build more advanced functions and logic.

3.3.3 Compound Assignment

Python supports shorthand operations for self-assignment:

```
A = 10
A = A + 10  # Equivalent to:
A += 10
```

This shorthand applies to all arithmetic operations (`+=`, `-=`, `*=`, `/=`, etc.).

Note: When performing operations on variables of different types, such as `int` and `float`, Python automatically converts the result to `float`. For example:

```
result = 10 + 10.5  # result is a float (20.5)
```

However, operations combining `str` with `int` or `float` will result in errors:

```
"10" + 10  # This will raise a TypeError
```

Experiment with different cases to understand how Python handles these scenarios. Also, remember that Python follows the BODMAS convention.

3.3.4 Logical Operations

Another important class of operations is **logical operations**. These operations are used when you need to run specific parts of the code based on multiple conditions. For example:

- To check if a number is greater than 5 **and** divisible by 3.
- To check if a number is greater than 5 **or** divisible by 3.

In both cases, you have a logical expression to evaluate. Python provides three logical operators to handle such cases:

(*i*) **and**: Evaluates to `True` if both conditions A and B are true, otherwise `False`.

(*ii*) **or**: Evaluates to `True` if at least one of the conditions A or B is true, otherwise `False`.

(*iii*) **not**: Returns the negation of condition A.

The truth table for these logical operators is shown below:

A	**B**	**A and B**	**A or B**	**not A**
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Table 3.2: Truth table for logical operators.

Logical operations can also involve more than two conditions. For example, suppose you have three conditions: **A**, **B**, and **C**. In such cases, you can group conditions using parentheses to control the order of evaluation. For instance:

- Check **(A and B)** first, and then combine the result with **C**.
- Evaluate **A or (B and C)** to prioritize **B and C**.

This flexibility allows for constructing complex logical expressions tailored to your requirements.

3.4 Conditional Statements

Conditional statements allow executing specific code blocks depending on conditions. In Python, the syntax is as follows:

```
if condition_1:
    execute_1
else:
    execute_2
```

Example: Checking if a number is even or odd:

```
eval_point = 5
if eval_point % 2 == 0:
    print(f"The number {eval_point} is even.")
else:
    print(f"The number {eval_point} is odd.")
```

For multiple conditions, we use `if-elif-else`:

```
eval_point = 5
if eval_point % 2 == 0:
    print(f"The number {eval_point} is divisible by 2.")
elif eval_point % 3 == 0:
    print(f"The number {eval_point} is divisible by 3.")
else:
    print(f"The number {eval_point} is not divisible by 2 or 3.")
```

Note: An `else` statement is usually not necessary for an `if` statement. Suppose we want to check if a number is even we can use

```
eval_point = 5
if eval_point % 2 == 0:
    print(f"The number {eval_point} is divisible by 2.")
```

Here we want to check if the number is even without checking if it odd or not.

Note: Here we have introduced a new way to print. The `print(f"...")` command is printing a formatted string. It prints the characters as well as the variable values defined in the curly braces $\{\cdot\}$.

3.5 Recursive Statements

Recursive Statements allow repetitive execution of code blocks.

3.5.1 For Loop

The syntax for a `for` loop in Python is:

```
for i in range(a, b):
    execute_1
```

Here:

- `i`: The loop iterator.
- `range(a, b)`: Specifies the range of values, starting at `a` and stopping before `b`, i.e., it goes over $\mathtt{a}, \mathtt{a}+1, \ldots, \mathtt{b}-1$.

Example: Summing numbers from 1 to 9:

```
total_sum = 0
for i in range(1, 10):
    print(i)
    total_sum += i
print(f"The summation of 9 points: {total_sum}")
```

3.5.2 Custom Step Size

The default step size of a `for` loop is one. If we want to use a custom step-size then we can use the following modification:

```
for i in range(a, b, step):
    execute_1
```

Example: Summing odd numbers from 1 to 9:

```
total_sum = 0
for i in range(1, 10, 2):
    print(i)
    total_sum += i
print(f"The summation of 10 points with step 2: {total_sum}")
```

3.5.3 Break and Continue

While using loops there can be cases when we want to exit the loop due to some condition. Also we can have cases when we want to skip some iteration. In this case we use `break` and `continue`, respectively.

- `break` : Exits the loop entirely.
- `continue` : Skips the current iteration and moves to the next.

Example: Adding even numbers up to 10 but stopping at 7:

```
total_sum = 0
for i in range(10):
    if i == 7:
        break
    if i % 2 == 1:
        continue
    print(i)
    total_sum += i
print(f"The summation of even numbers: {total_sum}")
```

3.5.4 Nested Loops

A `for` loop can be nested within another `for` loop. For example, to generate multiplication tables:

```
for i in range(1, 5):
    print(f"The table of {i}")
    for j in range(1, 11):
        print(f"{i} x {j} = {i * j}")
```

3.6 Functions

Until now, we have focused on sequential coding. However, to enhance reusability and maintainability, modular coding is essential. Functions enable modular programming by allowing code reuse. The syntax for creating a function is:

```
def function_name(input_1, input_2):
    # Function body
    result = some_operation(input_1, input_2)
    return result
```

Example: Function to check if a number is even:

```
def is_even(number):
        if number % 2 == 0:
                return True
        else:
                return False

value = 20
result = is_even(value)
print(f"The number {value} is even: {result}")
```

Note:

- A function can accept multiple inputs, a single input, or no input at all.
- A function may include multiple `return` statements or omit a `return` entirely, in which case it returns `None` by default.

Example: Function with multiple `return` statements:

```
def analyze_number(number):
    if number > 0:
        return "positive", number
    elif number < 0:
        return "negative", number
    else:
        return "zero", number

result_type, result_value = analyze_number(-5)
print(f"The number {result_value} is {result_type}.")
```

3.7 NumPy Library

In this section we provide an introduction to the NumPy library, a fundamental Python library for mathematical computations.

Importing the Library

To use NumPy in Python, the library must be imported. The standard convention is to import it with the alias `np`:

> *Import the NumPy library as follows:* `import numpy as np`.

3.7.1 Arrays and Matrices

NumPy arrays are versatile tools used as vectors (one-dimensional) or matrices (two-dimensional). For instance:

- A one-dimensional array can be thought of as a row vector, e.g., $[1, 2, 3, 4]$. This kind of array can be created using `temp_array` = `np.array`$([1, 2, 3, 4])$.
- A two-dimensional matrix is defined by nesting arrays, e.g., $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$. This kind of array can be created using `temp_array` = `np.array`$([[1, 2, 3, 4], [5, 6, 7, 8]])$.

Key commands include:

- `np.size(array)`: Returns the total number of elements in an array.
- `np.shape(array)`: Provides the dimensions of an array.

Special Arrays: Zeros and Ones

It is common to initialize arrays with default values such as zeros or ones. This helps avoid uninitialized or garbage values in computations. For example:

- A zero matrix of size 10×10 can be created using `np.zeros((10, 10))`.
- Similarly, a ones matrix of the same size is created with `np.ones((10, 10))`.

Indexing in Arrays and Matrices

In NumPy:

- Indexing starts at 0.
- Negative indexing allows access to elements from the end, e.g., -1 refers to the last element.

For matrices, indexing uses row and column coordinates. For example, the element at row 0, column 0 in a matrix is accessed as [0][0]. Suppose we have the following $n \times n$ matrix

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0\ n-1} \\ a_{10} & a_{11} & \cdots & a_{1\ n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1\ 0} & a_{n-1\ 1} & \cdots & a_{n-1\ n-1} \end{bmatrix},$$

then the $a_{n-1\ n-1}$ entry can be accessed using $\texttt{A}[-1][-1]$ as well as $\texttt{A}[\texttt{n}-1][\texttt{n}-1]$. Similarly, $a_{0\ \texttt{n}-1}$ can be accessed using $\texttt{A}[0][-1]$ and $\texttt{A}[0][\texttt{n}-1]$.

3.7.2 Linspace

To generate arrays with evenly spaced points, the `np.linspace` function is used:

> *For an array between a and b with n elements, use the syntax:*
> `np.linspace(a, b, n).`

This is particularly useful for numerical methods. It it important to note that the end points a, b are included and the spacing between the points is $(b-a)/(n-1)$.

3.7.3 Mathematical Functions

NumPy provides a wide range of mathematical functions, including:

- Trigonometric functions: `np.sin`, `np.cos`, etc.
- Hyperbolic functions: `np.sinh`, `np.cosh`, etc.
- Absolute value: `np.abs`.

For example, the sine and absolute value of $-\pi$ can be computed using `np.sin(-np.pi)` and `np.abs(-np.pi)`. For a comprehensive list of available mathematical routines, refer to the official documentation: `https://numpy.org/doc/stable/reference/routines.math.html`.

List of Algorithms

Index

Bibliography

[1] Robert G. Bartle and Donald R. Sherbert. *Introduction to real analysis*. Second. John Wiley & Sons, Inc., New York, 1992, pp. xii+404. ISBN: 0-471-51000-9.

[2] Jean-Paul Berrut and Lloyd N. Trefethen. “Barycentric Lagrange interpolation”. In: *SIAM Rev.* 46.3 (2004), pp. 501–517. ISSN: 0036-1445,1095-7200. DOI: `10.1137/S0036144502417715`. URL: `https://doi.org/10.1137/S0036144502417715`.

[3] J. Douglas Faires and Richard Burden. *Numerical methods*. Second. With 1 IBM-PC floppy disk (3.5 inch; HD). Brooks/Cole Publishing Co., Pacific Grove, CA, 1998, pp. xii+594. ISBN: 0-534-35187-5.

[4] L. R. Ford Jr. and D. R. Fulkerson. “Maximal flow through a network”. In: *Canadian J. Math.* 8 (1956), pp. 399–404. ISSN: 0008-414X,1496-4279. DOI: `10.4153/CJM-1956-045-5`. URL: `https://doi.org/10.4153/CJM-1956-045-5`.

[5] S.H. Friedberg, A.J. Insel, and L.E. Spence. *Linear Algebra*. Pearson Education, 2014. ISBN: 9780321998897. URL: `https://books.google.co.in/books?id=KyB0DAAAQBAJ`.