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 0.0.1) 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).

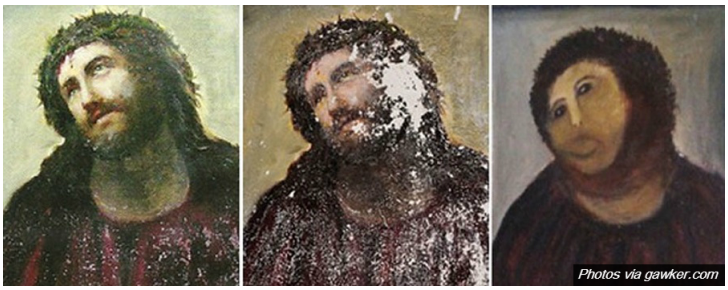

Figure 1.1: Elías García Martínez, Ecce Homo: The leftmost photograph, taken in 2010, shows some initial flaking of the paintwork. The central photograph was taken in July 2012, just a month before the attempted restoration, showing the extent of damage and deterioration. The rightmost photograph documents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

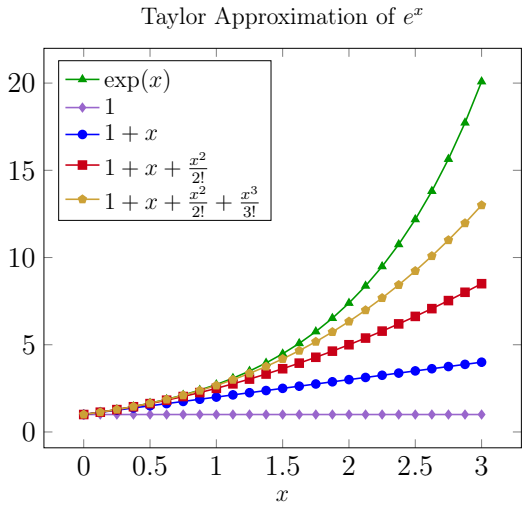

Figure 1.4: Taylor polynomials for exponential function approximated at $x = 0$.

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 [4, 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

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

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_{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**
    $\mathbf{V}[i][j] = x_i[i]^{(n-j)}$
  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_{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 $p_n^{\mathbb{V}}(x_{\text{eval}})$

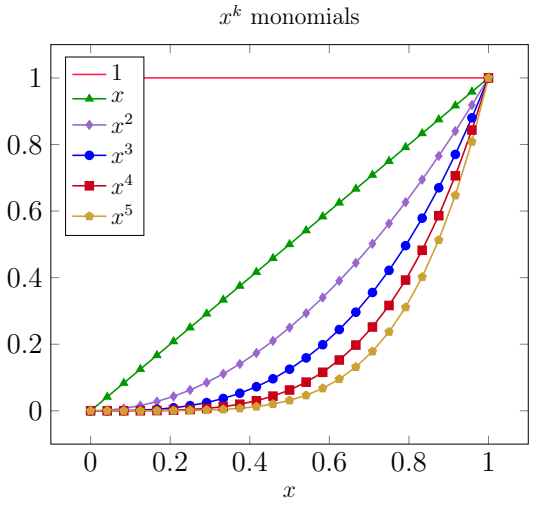

Figure 1.6: Monomial basis x^k for $k = 0, 1, \ldots, 5$.

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

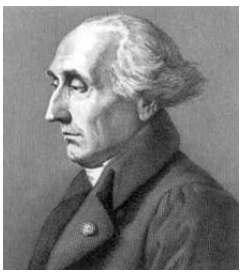

Figure 1.7: Joseph-Louis Lagrange: 25 January 1736-10 April 1813

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 $x_1 = 1$.

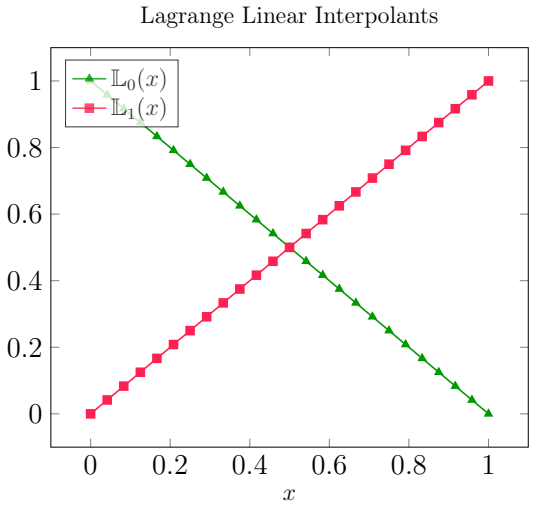

Figure 1.8: Lagrange linear interpolating polynomials for $x_i = \{0, 1\}$.

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.

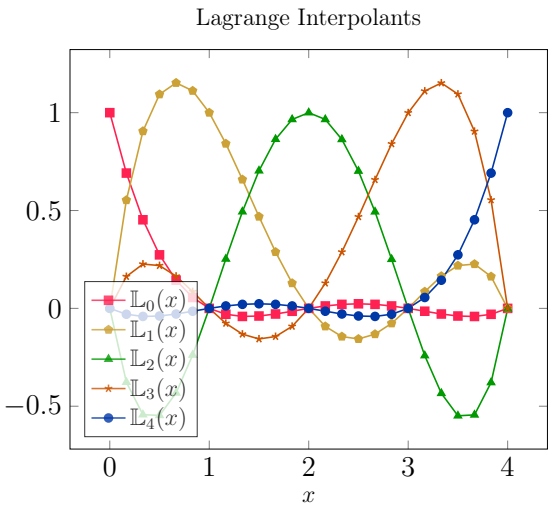

Figure 1.9: Lagrange interpolating polynomials defined over $x_i = 0, 1, 2, 3, 4$.

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., $p_n q = q$ 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 $k = 0, 1, \ldots, n$ 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_{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_{eval}
Initialize $p_n^{\mathbb{L}}(x_{\text{eval}}) = 0$
for $i = 0$ **to** n **do**
  $p_n^{\mathbb{L}}(x_{\text{eval}}) = p_n^{\mathbb{L}}(x_{\text{eval}}) + f(x_i) \times \mathbb{L}_i(x_{\text{eval}})$
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

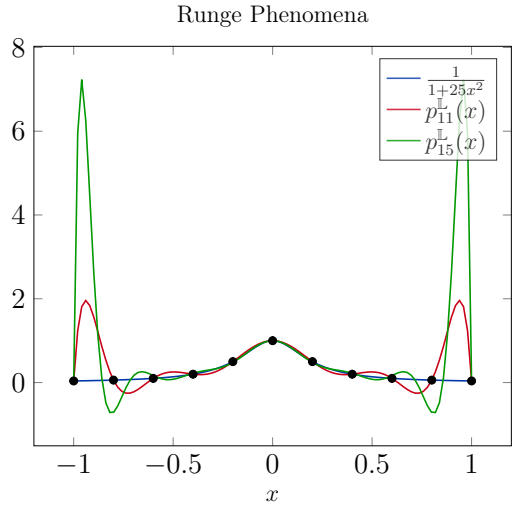

Figure 1.11: Runge phenomena for the function $1/(1+25x^2)$. $p_{11}^{\mathbb{L}}(x)$ refers to an approximation computed using 11 points (the dots refer to $\{x_i\}_{i=0}^{10}$), $p_{15}^{\mathbb{L}}(x)$ refers to approximation using 15 points.

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

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

Now, it can be shown (although not very easily) that $\max_{-1 \le x \le 1} \prod_{i=0}^{n} |x - x_i| \le h^{n+1} n!$ 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

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

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].

Chapter 2

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 2.1.

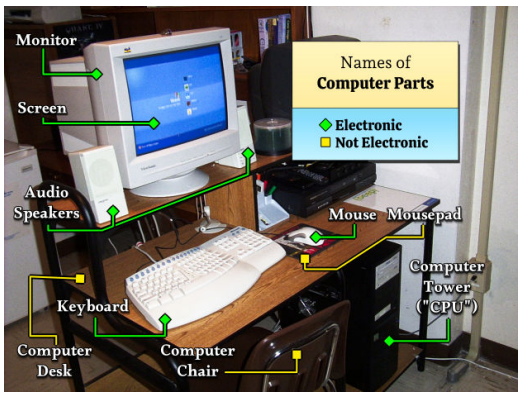

Figure 2.1: Parts of a computer from the early 2000s.

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.

2.1 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.

2.1.1 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:

```
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:

```
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;
}
```

2.1.2 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 ${}^{2}C_{k}$, the factorial function can be reused without rewriting it.

2.2 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.

2.3 Introduction to Computing Using Python

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

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.

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 2.1: 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.

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.

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 2.2: 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.

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] 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.