Methods of Numerical Analysis

Anwesha Ghosh
Ashoka University, Sonipat
anwesha.ghosh_ug2023@ashoka.edu.in

October 10, 2024

Abstract

This exposition explores key methods in numerical analysis, focusing on root-finding techniques like bisection, false position, fixed-point iteration, Newton–Raphson, and secant, as well as interpolation methods including Lagrange, Newton, and spline interpolation. The project includes developing an independent Python library that implements these methods without external modules, incorporating basic operations like numerical differentiation. Through theoretical discussion and practical implementation, this work highlights the importance of numerical analysis in solving complex mathematical problems.

Introduction

Numerical analysis is a branch of mathematics that focuses on developing and analyzing algorithms for solving problems involving continuous variables. These problems often arise in scientific and engineering contexts where exact solutions are difficult or impossible to obtain analytically. Numerical methods provide a way to approximate these solutions with a high degree of accuracy, using computational techniques that can handle the complexities of real-world data.

In this paper, we explore several key methods in numerical analysis, focusing on root-finding and interpolation techniques. Root-finding methods, such as bisection method, false position method, fixed point iteration method, Newton–Raphson method and secant method are essential for solving equations where the exact roots are not easily determined. These methods vary in their convergence rates and computational efficiency, offering a range of options depending on the nature of the problem at hand.

Interpolation methods, including Lagrange, Newton, and spline interpolation, are crucial for constructing functions that approximate data points. These techniques enable us to estimate values within the range of known data, providing a powerful tool for data analysis and prediction.

The content of this paper is from the book [1]. Throughout the paper, we will discuss the theoretical foundations of these methods and visualize them. Additionally, Python programs accompanying each method illustrate how these algorithms can be efficiently coded and executed, bridging the gap between theory and practice in numerical analysis. This project aims to create an independent Python library without relying on any external modules, encompassing not only these numerical methods but also defining basic mathematical operations such as numerical differentiation.

1 Defining Basic Mathematical Operations and Functions

Trigonometric Functions

```python
def sin(x):
    # Taylor series approximation for sin(x)
    result = 0
    term = x
    n = 1
    while abs(term) > 1e-15:
        result += term
        term *= -x * x / (2 * n * (2 * n + 1))
        n += 1
    return result

def cos(x):
    # Taylor series approximation for cos(x)
    result = 0
    term = 1
    n = 0
    while abs(term) > 1e-15:
        result += term
        term *= -x * x / (2 * n * (2 * n - 1)) if n > 0 else -x * x / 2
        n += 1
    return result

def tan(x):
    return sin(x) / cos(x) if cos(x) != 0 else 'undefined'

def csc(x):
    return 1 / sin(x) if sin(x) != 0 else 'undefined'

def sec(x):
    return 1 / cos(x) if cos(x) != 0 else 'undefined'

def cot(x):
    return 1 / tan(x) if tan(x) != 0 else 'undefined'
```

Listing 1: Defining Trigonometric Functions

Factorial

```python
def factorial(n):
    factorial = 1
    for i in range(n):
        factorial *= i+1
    return factorial
```

Listing 2: Code for Computing Factorial

Numerical Differentiation

```
def numerical_differentiation(f, x, h=1e-5):
    return (f(x + h) - f(x)) / h
```

Listing 3: Code for Numerical Differentiation

2 Methods to Find Roots of Algebraic and Transcendental Equations

A problem frequently encountered by engineers, scientists and statisticians is finding roots of equations of the form:

$$f(x) = 0$$

This is fairly easy to solve if $f(x)$ is a quadratic, cubic or biquadratic expression. However, if $f(x)$ is a polynomial of higher degree or contains some transcendental functions, there seem to be no algebraic methods present to solve them. Thus, we must find some methods to approximate the roots of such equations.
This section goes into further detail of such methods and also, presents an algorithm in python to compute the roots.

2.1 Bisection Method

Theorem 2.1 (Intermediate value theorem)**.** *If f(x) is continuous in $a \leq x \leq b$, and if $f(a)$ and $f(b)$ are of opposite signs, then f(ξ) = 0 for at least one number ξ such that $a \leq \xi \leq b$.*

The first method we will examine is based on Theorem 2.1 stated above. To apply the bisection method for finding roots for an expression $f(x)$:

1. Let $f(a)$ be positive and $f(b)$ be negative.
2. The root lies between a and b, and let its approximate value be given by $x_0 = (a+b)/2$, that is, bisecting the two points.
3. If $f(x_0) = 0$ then x_0 is the root of $f(x) = 0$. Otherwise, the root either lies between x_0 and b, or between x_0 and a depending on whether $f(x_0)$ is positive or negative.
4. The new interval $[a_1, b_1]$ whose length is given by $|a - b|/2$. We bisect this again at x_1, and the interval procured will be exactly half the length of the previous one.
5. Repeat this process till the latest interval is as small as desired, say ε.

NOTE At every interval, the width is reduced by a factor of one-half. Thus, we get closer and closer to the root.

6. The nth interval will be $[a_n, b_n]$ of length $|a-b|/2^n$. Thus,

$$\frac{|a-b|}{2^n} \leq \varepsilon \tag{2.1}$$

which can be simplified to,

$$n \geq \frac{(log_e(|a-b|/\varepsilon)}{log_e 2} \tag{2.2}$$

Here, Equation 2.2 gives us the number of iterations necessary to get an accurate root.

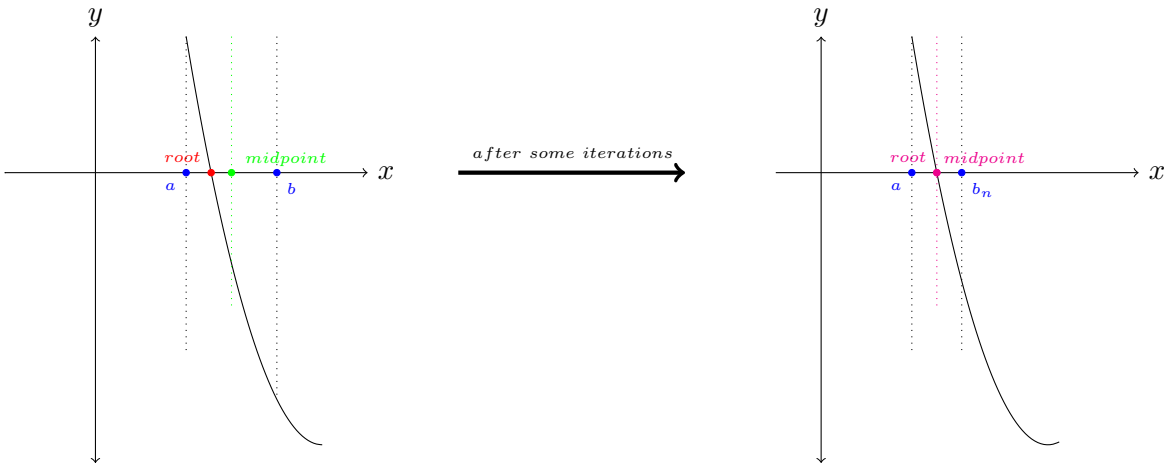


Figure 1: Bisection Method

NOTE The bisection method always succeeds. If there is more than one root present, this method finds one of them. The following is a simple program to apply the bisection method.

```
def bisection_method(f_str, a, b): # enter function in the form of string

     def f(x):
         f = eval(f_str)
         return f

    fa = f(a)
    fb = f(b)
    if fa*fb > 0:
        print("f(a) and f(b) must have different signs.")
        return None


    for _ in range(10000):
        c = (a + b)/2
        fc = f(c)

        if fc == 0:
```

```
            break
        if fa*fc > 0:
            a, fa = c, fc
        if fb*fc > 0:
            b, fb = c, fc
    return c
```

Listing 4: Code for Bisection Method

2.2 The Method of False Position

This is the oldest root-finding method for nonlinear equations of the form $f(x) = 0$. It is similar to the bisection method and secant method. This is also known as the *regula-falsi method* or the *method of chords*.

To apply the method of false position on an equation $f(x)$:

1. Choose two points a and b such that $f(a)$ and $f(b)$ are of opposite signs, that is, $f(a) \cdot f(b) < 0$. The root must lie between these points.

2. We consider the equation of the chord joining the two points $[a, f(a)]$ and $[b, f(b)]$, which is given by

$$\frac{y - f(a)}{x - a} = \frac{f(b) - f(a)}{b - a}. \tag{2.3}$$

3. The chord, represented by Equation 2.3 cuts the x *axis* at some point, let's say c. Now, to find the value of c, we need to solve Equation (2.3) at $y = 0$. This gives us the following equation:

$$c = a - \frac{f(a)}{f(b) - f(a)}(b - a) = \frac{af(b) - bf(a)}{f(b) - f(a)}, \tag{2.4}$$

   which forms our *first approximation*.

4. If $f(c) \cdot f(a) < 0$, then the root lies between a and c, and we replace b by c in Equation 2.4, which gives us the new approximation. Otherwise, replace a by c and generate the required approximation.

5. The above process is repeated till we find the root of desired accuracy.

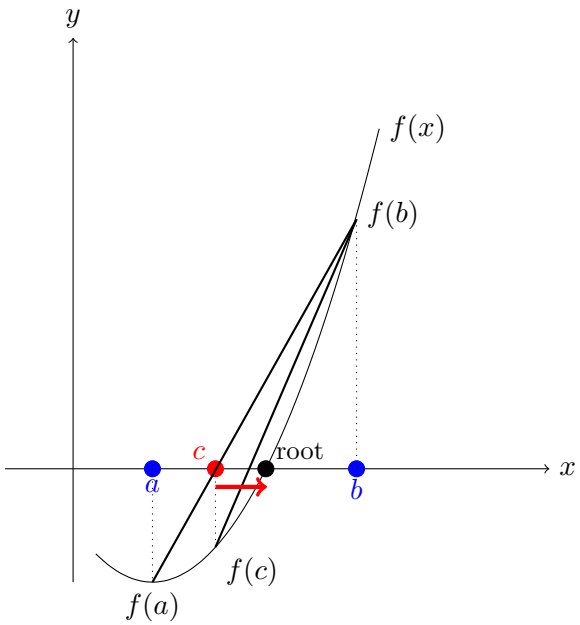


Figure 2: Method of False Position (Regula-Falsi)

The following is a python program to apply the method of false position:

```
def false_position_method(func_str, a, b, error_accept): # input function as string, interval points [a,b] and user's acceptable level of error

    def f(x):

        f = eval(func_str) # evaluates function entered in form of string as an "equation"
        return f

    i = 0
    c_before = 0
    c = (a * f(b) - b * f(a)) / (f(b) - f(a)) # calculates secant line
    error = abs(c - c_before)

    while error > error_accept:
        c_after = (a * f(b) - b * f(a)) / (f(b) - f(a))

        if f(a) * f(b) >= 0:
            print("Either no root present, or more than one root present. False Position method is invalid.")
            quit()

        elif f(c_after) * f(a) < 0:
            error = abs(c_after - b)
            b = c_after
            i = i + 1

        elif f(c_after) * f(b) < 0:
```

```
            error = abs(c_after - a)
            a = c_after
            i = i + 1

        else:
            print("Something went wrong.")
            quit()

    print(f"The error remaining is {error}, after {i} iterations")
    print(f"The root can be approximately found at {c_after}.")
    print(f"The lower root boundary, a, is {a} and the upper root boundary, b, is
    {b}")
```

Listing 5: Code for the Method of False Position

2.3 Fixed Point Iteration Method

Definition 1 (Fixed Point)**.** *A point, say s, is called a fixed point if it satisfies the equation $x = g(x)$.*

In the fixed point iteration method, any equation of the form $f(x) = 0$ can be converted algebraically into the form $x = g(x)$ and then iterated using the recursion relation

$$x_{i+1} = g(x_i), \quad i = 0, 1, 2, \ldots \tag{2.5}$$

with some initial guess x_0.

To apply the method for finding the root of the equation $f(x) = 0$

1. Rewrite $f(x) = 0$ in the form
$$x = g(x) \tag{2.6}$$
2. Assume the root to be at x_0.
3. Substitute x_0 in Equation 2., we get the first approximation
$$x_1 = g(x_0). \tag{2.7}$$
4. After successive substitutions, we get the approximations
$$x_2 = g(x_1), \quad x_3 = g(x_2) \ldots, \quad x_n = g(x_{n-1}). \tag{2.8}$$
5. This sequence may not always converge to a definite number. But if it does converge to a definite number ξ, then ξ will be a root of the equation $x = g(x)$.

   To show this, let
   $$x_{n+1} = g(x_n) \tag{2.9}$$
   be the relation between the nth and the $(n+1)$th approximations. As n increases, $x_{n+1} \to \xi$ and if $g(x)$ is a continuous function, then $g(x_n) \to g(\xi)$. Now, in the limit, we obtain
   $$\xi = g(\xi), \tag{2.10}$$
   which shows that ξ is a root of the equation $x = g(x)$.

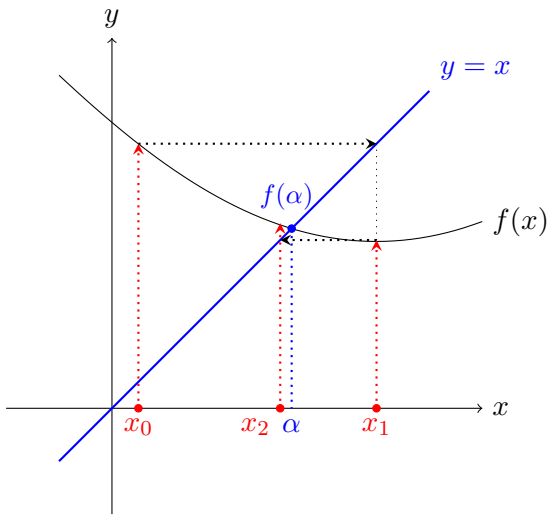


Figure 3: Fixed Point Iteration Method

```python
def fixed_point_iteration(x_0, error_accept, maximum_step):

    x_0 = float(x_0)
    e = float(error_accept)
    N = int(maximum_step)

    step = 1
    flag = 1
    condition = True

    while condition:
        x_1 = g(x_0)
        print('Iteration-%d, x1 = %0.6f and f(x1) = %0.6f' % (step, x_1, f(x_1)))
        x_0 = x_1

        step = step + 1

        if step > N:
            flag = 0
            break

        condition = abs(f(x_1)) > e

    if flag==1:
        print('\nRequired root is: %0.8f' % x_1)
    else:
        print('\nNot Convergent.')
```

Listing 6: Code for Fixed Point Iteration Method

2.4 Newton–Rhaphson Method

The Newton–Raphson method, named after Isaac Newton and Joseph Rhaphson is a root-finding method used to improve the result obtained by the methods shown so far.

To apply the Newton–Rhaphson method on an equation $f(x)$:

1. Assume the root to be at x_0. Let $x_1 = x_0 + h$ be the correct root such that $f(x_1) = 0$.
2. Rewriting $f(x_0 + h)$ using Taylor's series, we get
$$f(x_0) + hf'(x_0) + \frac{h^2}{2!}f''(x_0) + \cdots = 0. \tag{2.11}$$
3. Now, neglecting the second and higher-order derivatives, we have
$$f(x_0) + hf'(x_0) = 0,$$
which gives
$$h = -\frac{f(x_0)}{f'(x_0)}. \tag{2.12}$$
4. Equation 2.12 is a better approximation than x_0, therefore, given by x_1, where
$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{2.13}$$
5. Following the pattern above in Equation 2.13, we form successive approximations given by x_2, x_3, …, x_{n+1}, where
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \tag{2.14}$$
which is the *Newton–Raphson formula*.

NOTE The idea behind the Newton–Rhaphson method is to use the tangent line to the curve $y = f(x)$ at the point $(x_n, f(x_n))$ to approximate where the curve crosses the x *axis*. The x *intercept* of this tangent line becomes the next approximation x_{n+1}.

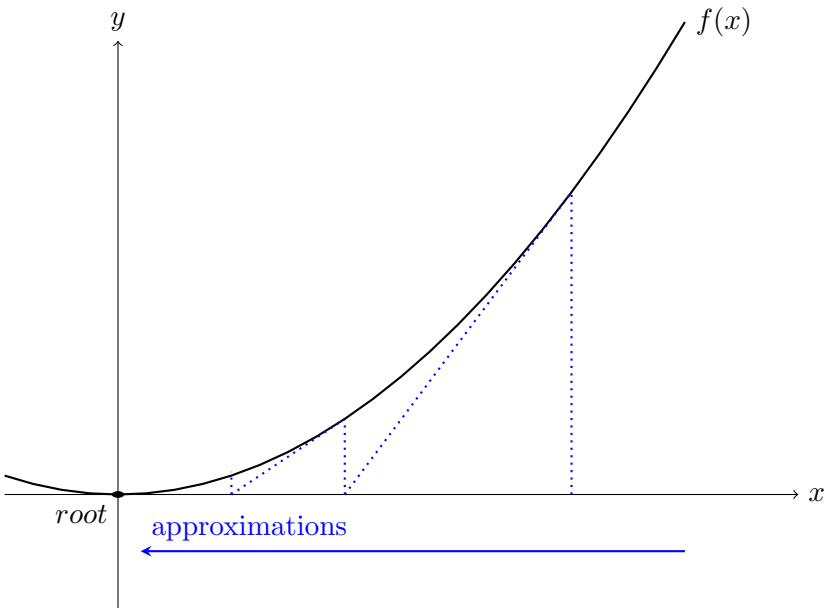


Figure 4: Newton-Rhapson Method

```
def newton_rhapson_method(func_str, derivfunc, x, n): # inputs the function, derivative of the function, initial guess and number of iterations

    def f(x):
        f = eval(func_str)
        return f

    def df(x):
        df = eval(derivfunc)
        return df

    for iteration in range(1, n+1):

        i = x - (f(x)/df(x))
        x = i

    print(f"Root found at {x} after {n} iterations.")
```

Listing 7: Code for Netwon-Rhapson Method

2.5 Secant Method

We know that the Newton–Rhapson method requires the derivative of the function, which is not always possible. The secant method bypasses this issue by approximating the derivative at x_i using the formula

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

which can be written as

$$f_i' = \frac{f_i - f_{i-1}}{x_i - x_{i-1}}, \tag{2.15}$$

where $f_i = f(x_i)$. Hence, the Newton–Rhapson formula is rewritten as

$$x_{i+1} = x_i - \frac{f_i(x_i - x_{i-1})}{f_i - f_{i-1}} = \frac{x_{i-1}f_i - x_i f_{i-1}}{f_i - f_{i-1}}. \tag{2.16}$$

NOTE The secant method requires two initial approximations to the root.

```python
def secant_method(func_str, x0, x1, n): # enter function in form of string

    def f(x):
        f = eval(func_str)
        return f

    for intercept in range(1,n+1):
        fx0 = f(x0)
        fx1 = f(x1)

        xi = x0 - (fx0/((fx0 - fx1)/(x0 - x1)))
        x0 = x1
        x1 = xi

    print(f"Root found at {xi} after {n} iterations.")
```

Listing 8: Code for Secant Method

3 Root finding for the function $f(x) = \tan\left(\frac{x}{4}\right) - 1$

Let's use the root-finding methods to approximate the solutions of the equation $f(x) = \tan\left(\frac{x}{4}\right) - 1$. This equation has a well-known root at $x = 4$, corresponding to the fact that $\tan\left(\frac{4}{4}\right) = \tan(1) = 1$. The results of applying each of the root-finding methods (Newton-Raphson, Bisection, Secant, and False Position) show that the root converges to approximately **3.141593** with the following iterations:

- **Newton–Raphson Method:** The root is **3.141593**, found in **4 iterations**.
- **Bisection Method:** The root is **3.141594**, found in **16 iterations**.
- **Secant Method:** The root is **3.141593**, found in **6 iterations**.
- **False Position Method:** The root is **3.141591**, found in **22 iterations**.

NOTE In this example, the fixed-point iteration method consistently yields the value of π. This happens because of the rearrangement $x = 4 \cdot \tan^{-1}(1)$, which directly gives fixed point at π. The convergence is not dependent on the choice of the initial guess, making this method inappropriate for computing an approximation of π.

The correct value of π until 20 digits is: 3.14159265358979323 85. Thus, when substituted into the expression $\tan\left(\frac{x}{4}\right)$, the value is approximately **3.141593**, which is an approximation of π. This demonstrates that each method accurately finds the root, confirming the consistency and reliability of these numerical approaches when applied to the function $f(x) = \tan\left(\frac{x}{4}\right) - 1$. Therefore, the result shows that the numerical methods effectively approximate π, but the number of iterations required to get the same accuracy provide a means to compare the methods.

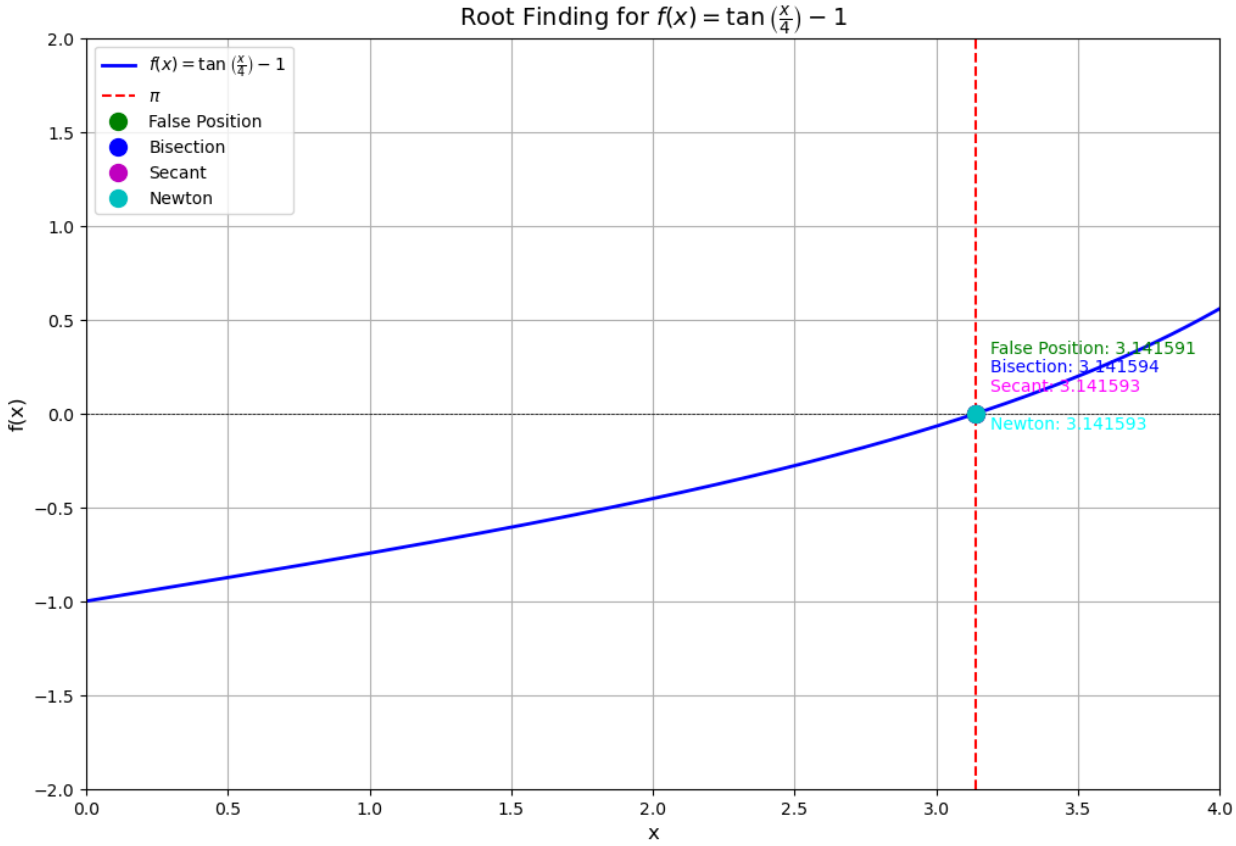


Figure 5: Root finding for the function $f(x) = \tan\left(\frac{x}{4}\right) - 1$

4 Methods for Interpolation

Interpolation is the problem of fitting a smooth curve through a given set of points, generally as the graph of a function.

4.1 Lagrange Interpolation

Lagrange's Interpolation is for problems where the values of the independent variables are unequally spaced.

Theorem 4.1. *(Lagrange Interpolation Formula)*

*Let $x_0, x_1, \ldots, x_n \in I = [a, b]$ be $n+1$ distinct nodes and let $f(x)$ be a continuous real-valued function defined on I. Then, there exists a unique polynomial p_n of degree $\geq n$ (called **Lagrange Formula for Interpolating Polynomial**), given by*

$$p_n(x) = \sum_{k=0}^{n} f(x_k) l_k(x), \quad l_k(x) = \prod_{i=0, i\neq k}^{n} \frac{x - x_i}{x_k - x_i}, k = 0, \ldots, n, \tag{4.1}$$

such that,

$$p_n(x + i) = f(x_i), \quad i = 0, 1, \ldots, n. \tag{4.2}$$

*The function $l_k(x)$ is called the **Lagrange multiplier**.*

The following is a simple python code to apply Lagrange's Interpolation.

```
def lagrange_interpolation():
    n = int(input('Enter number of data points: '))

    x = [0] * n
    y = [0] * n

    print('Enter data for x and y: ')
    for i in range(n):
        x[i] = float(input('x[' + str(i) + ']='))
        y[i] = float(input('y[' + str(i) + ']='))

    xp = float(input('Enter interpolation point: '))
    yp = 0

    for i in range(n):
        p = 1
        for j in range(n):
            if i != j:
                p *= (xp - x[j]) / (x[i] - x[j])
        yp += p * y[i]

    print('Interpolated value at %.3f is %.3f.' % (xp, yp))
```

Listing 9: Code for Lagrange Interpolation

4.2 Newton's Interpolation and Divide Differences

In the previous section, we saw that in the Lagrange's Interpolation formula for a function, if we add a point to the set of nodes to increase the accuracy, we would have to completely recompute all of the $l_i(x)$ functions. Therefore, we cannot express p_{n+1} in terms of p_n, using Lagrange's formula. An alternate form of the polynomial, known as the *Newton form*, avoids this problem, and allows us to easily write p_{n+1} in terms of p_n.

The idea behind the Newton form is to write $p_n(x)$ in the following form:

$$p_n(x) = A_0 + A_1(x - x_0) + A_2(x - x_0)(x - x_1) + \cdots + A_n(x - x_0)\cdots(x - x_{n-1}) \tag{4.3}$$

where the coefficients $A_i, i = 0, 1, \ldots, n$ are to be obtained. From the interpolation condition that this polynomial agrees with the function value at the node points, we get

$$A_0 = p_n(x_0) = f(x_0).$$

For $x = x_1$, we have

$$A_1 = \frac{p_n(x_1 - A_0}{x_1 - x_0} = \frac{f(x_1) - f(x_0)}{x_1 - x_0} := f[x_0, x_1].$$

For $x = x_2$, we have

$$A_2 = \frac{p_n(x_2) - p_1(x_2)}{(x_2 - x_0)(x_2 - x_1)} = \frac{f(x_2) - p_1(x_2)}{(x_2 - x_0)(x_2 - x_1)} := f[x_0, x_1, x_2].$$

In this way, we can obtain all the coefficients.

The advantage in the Newton form is that if p_n is already calculated, then p_{n+1} can be written as

$$p_{n+1}(x) = p_n(x) + A_{n+1}(x - x_0)\cdots(x - x_n)$$

This also shows that the coefficient A_{n+1} in the Newton form, as seen in Equation 4.3, for the interpolating polynomial is the leading coefficient. This means, the coefficient of x^{n+1}, in the polynomial p_{n+1} of degree $\geq n + 1$ which agree with $f(x)$ at $x_0, \ldots, x_{n+1}$. The following theorem summarises this.

Theorem 4.2 (**Newton's Interpolation Formula**).
Let p_n be the polynomial that interpolates a continuous function $f(x)$ at $(n + 1)$ distinct nodes $x_i \in I$, for $i = 0, 1, \ldots, n$. Then the polynomial p_{n+1} that interpolates f at $(n + 2)$ distinct nodes $x_i \in I$, for $i = 0, 1, \ldots, n + 1$ is given by

$$p_{n+1}(x) = p_n(x) + f[x_0, x_1, \ldots, x_{n+1}]w_n(x), \tag{4.4}$$

where,

$$f[x_0, x_1, \ldots, x_{n+1}] = \frac{f(x_{n+1}) - p_n(x_{n+1})}{w_n(x_{n+1})}, \quad f[x_0] = f(x_0) \tag{4.5}$$

*is called the $(n + 1)$th **divided difference** of $f(x)$ of points $x_0, x_1, \ldots, x_{n+1}$ with*

$$w_n(x) = \prod_{i=0}^{n}(x - x_i). \tag{4.6}$$

*Equation 4.4 is called the **Newton Formula for Interpolating Polynomial**.*

Newton's Forward Interpolation

```
def newton_forward_interpolation():

    n = int(input("Enter number of values given: "));
    x = [];
    print("Enter the x values:")
    for i in range(n):
        x_val = float(input(f"x[{i}]: "))
        x.append(x_val)

    def u_cal(u, n):

      temp = u;
      for i in range(1, n):
        temp = temp * (u - i);
      return temp;

    # factorial previously defined

    factorial(n)

    # y[][] is used for difference table
    # with y[][0] used for input
    print("Enter the y values:")
    for i in range(n):
        y_val = float(input(f"y[{i}]: "))
        y.append(y_val)

    # Calculating the forward difference table
    for i in range(1, n):
      for j in range(n - i):
        y[j][i] = y[j + 1][i - 1] - y[j][i - 1];

    # Displaying the forward difference table
    for i in range(n):
      print(x[i], end = "\t");
      for j in range(n - i):
        print(y[i][j], end = "\t");
      print("");

    # Value to interpolate at
    value = float(input("value to interpolate at: "));

    # initializing u and sum
    sum = y[0][0];
    u = (value - x[0]) / (x[1] - x[0]);
    for i in range(1,n):
      sum = sum + (u_cal(u, i) * y[0][i]) / fact(i);

    print("\nValue at", value,
      "is", round(sum, 6));

```

```
newton_forward_interpolation()
```

Listing 10: Code for Newton's Forward Interpolation

Newton's Backward Interpolation

```

def newton_backward_interpolation():
    n = int(input("Enter number of values given: "));
    x = [];
    print("Enter the x values:")
    for i in range(n):
        x_val = float(input(f"x[{i}]: "))
        x.append(x_val)

    def u_cal(u, n):
      temp = u
      for i in range(n):
        temp = temp * (u + i)
      return temp

    # factorial previously defined

    factorial(n)


    # y[][] is used for difference table
    # with y[][0] used for input
    print("Enter the y values:")
    for i in range(n):
        y_val = int(input(f"y[{i}]: "))
        y.append(y_val)

    # Calculating the backward  difference table
    for i in range(1, n):
      for j in range(n - i):
        y[j][i] = y[j + 1][i - 1] - y[j][i - 1];

    # Displaying the backward difference table
    for i in range(n):
      for j in range(i + 1):
        print(y[i][j], end="\t")
      print()

    # Value to interpolate at
    value = float(input("value to interpolate at: "));


    sum = y[n - 1][0]
    u = (value - x[n - 1]) / (x[1] - x[0])
    for i in range(1, n):
      sum = sum + (u_cal(u, i) * y[n - 1][i]) / fact(i)

```

```
48     print("\n Value at", value, "is", sum)
```

Listing 11: Code for Newton's Backward Interpolation

5 Deriving the Formula for Summation of k^4

We will derive the formula for the summation $S = \sum_{k=1}^{n} k^4$ using both Lagrange interpolation and Newton interpolation.

5.1 Lagrange Interpolation

First, we will consider the values of k^4 for $k = 1, 2, 3, 4, 5$:

$$\begin{aligned} 1^4 &= 1 \\ 2^4 &= 16 \\ 3^4 &= 81 \\ 4^4 &= 256 \\ 5^4 &= 625 \end{aligned}$$

Next, we define the Lagrange basis polynomials:

$$L_j(x) = \prod_{\substack{0 \le m \le 4 \\ m \neq j}} \frac{x - x_m}{x_j - x_m}.$$

Using $x_0 = 1, x_1 = 2, x_2 = 3, x_3 = 4, x_4 = 5$:

$$S(x) = \sum_{j=0}^{4} y_j L_j(x)$$

where $y_j = j^4$.
The polynomial $S(x)$ can be expanded and simplified as follows:

$$S(x) = 1 \cdot L_0(x) + 16 \cdot L_1(x) + 81 \cdot L_2(x) + 256 \cdot L_3(x) + 625 \cdot L_4(x).$$

Calculating $S(n)$ and simplifying gives:

$$S(n) = \frac{1}{30}n^5 + \frac{1}{6}n^4 + \frac{1}{12}n^3 - \frac{1}{30}n.$$

This leads to the final formula for the summation:

$$\sum_{k=1}^{n} k^4 = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{12} - \frac{n}{30}.$$

5.2 Newton Interpolation

Using the same data points, we construct the Newton interpolation polynomial:

1. **Finite Differences:**

$$\begin{aligned}\Delta^1 &: 15, 65, 175, 369\\ \Delta^2 &: 50, 110, 194\\ \Delta^3 &: 60, 84\\ \Delta^4 &: 24\end{aligned}$$

2. **Newton's Interpolating Polynomial:**

The Newton polynomial is given by:

$$\begin{aligned}P_n(x) = f(x_0) + \frac{\Delta^1}{1!}(x - x_0) + \frac{\Delta^2}{2!}(x - x_0)(x - x_1)\\ + \frac{\Delta^3}{3!}(x - x_0)(x - x_1)(x - x_2) + \frac{\Delta^4}{4!}(x - x_0)(x - x_1)(x - x_2)(x - x_3)\end{aligned}$$

This simplifies to:

$$P_n(n) = \frac{1}{30}n^5 + \frac{1}{6}n^4 + \frac{1}{12}n^3 - \frac{1}{30}n.$$

Thus, both Lagrange and Newton interpolation yield the same result for the summation of k^4:

$$\sum_{k=1}^{n} k^4 = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{12} - \frac{n}{30}.$$

Acknowledgment

This article was written as a part of the LGP–Ashoka University Mathematics Apprenticeship Programme, Summer 2024.

References

[1] S.S. Sastry. *Introductory Methods of Numerical Analysis.* PHI Learning, 2012.