University of Central Lancashire

Numerical Analysis (MA2852) Project

Raheem Munir G20821418

Contents

1

Introduction

Numerical analysis is a branch of mathematics that deals with the development of methods and techniques to solve mathematical problems through the use of numerical approximations. Real-world problems are often too complex to yield exact solutions or may be constrained by computational limitations. Hence, numerical analysis provides invaluable tools to obtain approximate solutions that are sufficiently accurate for practical applications.

In this report, we will delve into the world of numerical analysis by exploring various numerical methods and approximation techniques. We will examine how these methods can be implemented using the popular Python programming language, which offers a rich ecosystem of numerical libraries and tools. Numerical methods play a critical role in solving mathematical problems that cannot be solved analytically or whose exact solutions are computationally infeasible. These methods involve using numerical approximations to obtain approximate solutions that are sufficiently accurate for practical use. We will explore different types of numerical methods, such as root finding, linear and nonlinear equation solving, interpolation, solving polynomials and numerical integration.

2

Procedure A: Solutions of nonlinear equations

2.1 Introduction to Laguerre's Method

In the mid 19th century, French mathematician Eduoard Laguerre developed a root-finding algorithm which was used to find complex roots of polynomials. The method, known as Laguerre's Method, was primarily known for its efficiency and accuracy in finding roots in roots of polynomials and it is considered a "sure fire" method. According to **Burden, R. L. and Faires, J. D.** [1] it gives cubic convergence and also approximates complex roots. This method has various applications in other fields, including computer graphics, physics, and cryptography.

Laguerre's method begins with an initial approximation for a root, and then iteratively refines this approximation using a formula that incorporates the polynomial's coefficients and derivatives. With each iteration, the algorithm gets closer to the true root by improving the estimate based on the formula, and this process continues until a desired level of accuracy is attained.

2.2 Newton-Raphson's Method

Another iterative method used for finding the roots/zeros of a function is called the Newton-Raphson method. Developed in the 17th century by mathematician Joseph Raphson, the method still remains to this day as one of the most commonly used methods to finding roots due to its efficiency and accuracy. Isaac Newton, mathematical genius, developed a very similar formula decades before Joseph but we generally consider Joseph Raphson's method to be the superior one out of the two as it is more simpler than Newton's.

Let us suppose we are seeking to find the root of a continuous and differentiable function, denoted as $f(x)$, and we have an initial approximation for the root near a point $x = x_0$. Then by Newton-Raphson method we can find a great approximation for the root using

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{2.1}$$

We can then repeat this as many times. To get the next value we simply just use this formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.2}$$

The process is repeated until the result converges to a desired level of accuracy.

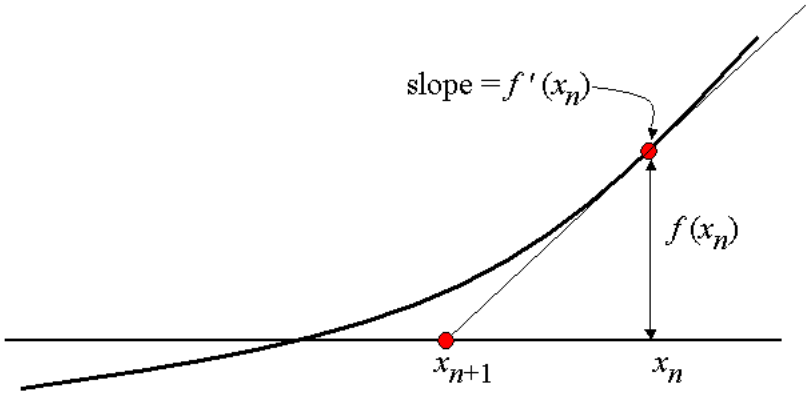

Figure 2.1: Visual representation of the Newton-Raphson method

Here is a great example by **Mehtre, Vishal V and Sharma, Shikhar** [6] where we can visualise a geometric representation of the method.

2.2.1 Newton-Raphson in Python

Here is a quickly created procedure for Newton-Rapson's method where we will be able to compare the results to the Laguerre iterative method:

```python
import math

def newton_raphson(f, f_prime, x0, tol=1e-6, max_iter=100):

    x = x0
    num_iter = 0

    while abs(f(x)) > tol and num_iter < max_iter:
        x = x - f(x) / f_prime(x)
        num_iter += 1

    return x, num_iter


# Define the function whose root needs to be found
def f(x):
    return x**3 - x - 1

# Define the derivative of the function f
def f_prime(x):
    return 3*x**2 - 1

# Initial guess for the root
x0 = 1.0

# Call the Newton-Raphson method
root, num_iter = newton_raphson(f, f_prime, x0)

# Print the result
print("Approximate root:", root)
print("Number of iterations:", num_iter)
```

Figure 2.2: Quick demonstration of how Newton-Raphson can be implemented in python

For this code, we took a function f whose roots needed to be found. We then used f_prime for the derivative of our function f. Obviously we need an initial guess which we called $x0$ and a tolerance level coded as tol. Finally another argument for the maximum number of iterations, we called this max_iter.

As an example, we used the function $f(x) = x^3 - x - 1$, and we set the tolerance to 10^{-6}. The result we obtained was:

As you can see, we programmed this procedure to give an approximate root and the amount of iterations applied. This will be very useful to us when we compare between this and the Laguerre iterative method.

```
Approximate root: 1.3247181739990537
Number of iterations: 4
```

Figure 2.3: Result from Newton-Raphson with the example conditions applied

2.3 Python Implementation of Laguerre's Method

Here is a procedure on the Laguerre method in python:

```python
import math
from sympy import *
import numpy as np

x = symbols('x')

def lag(f, n, P, tol, iter):
    step = 1                    # Initialise the iteration counter
    P0 = P                      # Set the initial value of P0
    f_diff = diff(f, x)             # Compute the first derivative of f
    f_second_diff = diff(f_diff, x) # Compute the second derivative of f
    fx = lambdify(x, f)             # Convert f to a lambda function
    fx_diff = lambdify(x, f_diff)  # Convert the first derivative of f to a lambda function
    fx_second_diff = lambdify(x, f_second_diff) # Convert the second derivative of f to a lambda function

    for step in range(iter):
        A = (fx_diff(P0)) / (fx(P0)) # Compute A
        B = A ** 2 - (fx_second_diff(P0) / fx(P0)) # Compute B
        if A > 0:
            d = n / ((A) + ((n - 1) * (n * B) - (A ** 2)) ** 0.5) # Compute d for A > 0 case
            if d <= tol:  # Check if d is less than or equal to the tolerance
                break
            else:
                P0 = P0 - d # Update P0 for next iteration
                step = step + 1 # Increment the iteration counter
                if step == 100: # Check if the maximum number of iterations is reached
                    break
        elif A < 0:
            d = n / ((A) - ((n - 1) * (n * B) - (A ** 2)) ** 0.5) # Compute d for A < 0 case
            if d <= tol:  # Check if d is less than or equal to the tolerance
                break
            else:
                P0 = P0 - d # Update P0 for next iteration
                step = step + 1 # Increment the iteration counter
                if step == 100: # Check if the maximum number of iterations is reached
                    break

    print("Approximate root is: ", P0) # Print the final result
    print("Number of iterations: ", step) # Print the number of iterations

lag(5 * x ** 5 + 2.3 * x ** 4 - 2 * x ** 2 + 6 * x - 3, 40, 100, 10 ** -4, 10) # Call the Lag function with input arguments
```

Figure 2.4: Code for the Laguerre iteraion method

The provided code implements Laguerre's method in Python. It defines a function called "lag" that takes five input arguments: f (the function whose root needs to be found), n (a constant), P (the initial guess for the root), tol (the tolerance for convergence), and $iter$ (the maximum number of iterations allowed).

Inside the function, several variables are initialised, such as *step* (the iteration counter), $P0$ (the current guess for the root), and f_diff and f_second_diff (the first and second derivatives of f, respectively). Additionally, lambda functions are created to numerically evaluate f,f_diff, and f_second_diff.

The function then enters a loop that iterates a maximum of *iter* times. Within the loop, A and B are computed using the current value of $P0$ and the lambda functions. It then checks if A is greater than 0 and computes d using a specific formula. If d is small enough (less than or equal to tol), the loop is cut. Otherwise, $P0$ is updated and step is incremented. However, if A is less than 0, a different formula is used to compute d, and the same convergence check is performed. If step reaches the maximum allowed iterations (100 in this case), the loop is stopped.

After the loop has completed its course, the final approximate root ($P0$) and the number of iterations (step) are printed. Finally, the lag function is called with specific input arguments to execute the root-finding algorithm with the given parameters.

2.3.1 The Result we Obtained For the Root

We were asked to find the root of $f(x) = 5x^5 + 2.3x^4 - 2x^2 + 6x - 3 = 0$. Using the Laguerre method we were able to obtain:

```
Approximate root is:  0.5288125965826731
Number of iterations:  10
```

Figure 2.5: Approximate root of the given function

2.4 Laguerre's Method vs Newton-Raphson's Method

Let us now compare which method takes less iterations for finding the root of $f(x) = 5x^5 + 2.3x^4 - 2x^2 + 6x - 3 = 0$. Here is the result when we input this function in the Newton-Raphson method:

```
Approximate root: 0.5287825872494671
Number of iterations: 5
```

Figure 2.6: Computing the same function as Laguerre's method

We get roughly the same approximation of the root for both Newton-Raphson's method (Fig 2.6) and Laguerre's method (Fig 2.5) however, based on the results of the comparison, it appears that the Newton-Raphson method in the original code has fewer iterations compared to the Laguerre's method. This suggests that the Newton-Raphson method may be faster in terms of convergence for the given function and initial guess. It is important for us to note that the number of iterations alone may not be the sole determinant of the overall efficiency of a root-finding method, as other factors such as computational complexity and accuracy also need to be considered.

3

Procedure B: Solving Runge's Example

3.1 Introduction to Runge's Phenomenon

Numerical interpolation is a technique used in mathematics to work out or estimate values of a function at unobserved points in a given set of data. In simpler terms, we are finding a function which passes through a known point and using that to approximate a function between those points. It is an excellent tool for estimating values, but it is important to be wary that it has its limitations such as divergence and oscillation in certain cases such as Runge's Example.

Runge's Example, or better known as Runge's Phenomenon, is a mathematical phenomenon which demonstrates the occurrence of divergence and oscillation in interpolation. Introduced in 1901 by the German mathematician Carl David Tolmé Runge, Runge's Example accentuates the limitations of using equidistant nodes, such as equally spaced points as it can lead to undesirable oscillating and diverging behaviour. This method is great for solving differential equations where interpolation occurs. We will look more into Runge's Example in the following sections.

3.2 Cubic Spline Interpolation

Cubic spline interpolation is primarily used in numerical analysis for approximating a smooth function by fitting a piece-wise cubic polynomial that goes through given data points with the desired output, the curve, to appear smooth and be continuous. A perfect description of this was described by **McKinley, Sky and Levine, Megan** [5] when they said that the fundamental idea behind cubic spline interpolation is based on the engineer's tool used to draw smooth curves through a number of points. This spline consists of weights attached to a flat surface at the points to be connected. A flexible strip is then bent across each of these weights, resulting in a pleasingly smooth curve. The result we get, known as the spline, has continuous first and second derivatives which ultimately makes it a smooth curve. This makes it quite useful in different applications of science, engineering etc. There are many different conditions in cubic spline interpolation to determine coefficients of cubic polynomials. The two common conditions that we will look at are the "Natural" and "Clamped" conditions.

3.2.1 The Natural Condition

Natural cubic splines are a type of curve used for interpolation or approximation of data points. They are usually defined as piece-wise functions that are set on intervals. A condition for these is that the second derivative is 0 at the endpoints of each interval. With all this being said, it should produce a smooth, continuous curve at the given data points with no abnormalities such as a little spike in the curves. An example of a natural cubic spline looks like this:

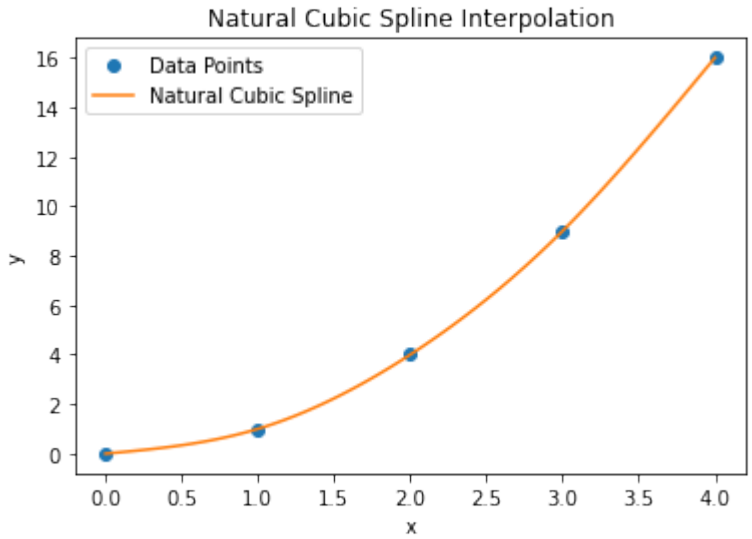

Figure 3.1: This shows the smoothness and continuity of a natural spline

The natural spline above was constructed in python. As we can see, the curve is smooth and continuous at the data points, with no abrupt changes in curvature.

Mathematically, if we have a set of data points $(x_0,\ y_0)$, $(x_1,\ y_1)$, ... , $(x_n,\ y_n)$, natural cubic splines can be used to find a set of cubic polynomials $S_i(x)$ for each interval $[x_i,\ x_{i+1}]$ such that:

- $S_i(x)$ is a cubic polynomial in the form of $S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$, where a_i, b_i, c_i, and d_i are coefficients to be figured out.
- $S_i(x)$ is well defined on the interval $[x_i,\ x_{i+1}]$ for $i = 0,\ 1,\ ...,\ n - 1$.
- $S_i(x_i) = y_i$, which means that the curve passes through each data point.

- $S_i'(x_i) = S_{i-1}'(x_i)$ for $i = 1, 2, ..., n-1$. This ensures smoothness as the first derivative is continuous at data points.
- $S_i''(x_i) = 0$ for $i = 0, n$. This means that the second derivative is zero at the endpoints of each interval. Which ensures continuity and "naturalness" behavior even beyond the data points.

The conditions stated above result in a set of $n-1$ cubic polynomials that can be used to interpolate the data points. As stated before, look at figure 3.1 for a great example of this.

Construction of the Natural Cubic Spline

As aforementioned, the spline in figure 3.1 was created in python. Firstly, we must import the cubic spline package to allow us to work with interpolation of data. The code uses the *CubicSpline* class from SciPy to compute the natural cubic spline for a set of data points. Then, we must set data points. To do this we must create an array with example data points. For this example in figure 3.1, the code used to create a simple array is $x = np.array([0, 1, 2, 3, 4])$ and $y = np.array([0, 1, 4, 9, 16])$. We use x and y to define each data set for given data. The bc_type parameter is set to 'natural' to indicate that natural boundary conditions (i.e. zero second derivatives at the endpoints) should be used for the cubic splines. To evaluate the spline at a set of data points, the code used is $x\_eval = np.linspace(0, 4, 100)$. This generates 100 equally spaced points between 0 and 4. The $y\_eval = cs(x\_eval)$ evaluates the cubic spline at these data points and stores the results in y_eval. The code $cs = CubicSpline(x, y, bc\_type =' natural')$ is used to compute the cubic spline where the *CubicSpline* function takes x and y as inputs along with $bc\_type = natural$ as an argument. Hence cs is then defined and can be used in other parts of the code. We must then generate the coefficients which is done by by $cs.c$. This returns the coefficients of the polynomial for every interval of the spline. The code then iterates, using a For Loop, the coefficients and prints equations of cubic polynomial for each interval and then finally displaying the coefficients with the corresponding x-values. The data is then plotted onto a graph using the matplotlib package. The final result plots the natural cubic spline as a smooth curve.

3.2.2 The Clamped Condition

For the clamped condition in cubic spline interpolation, it is known as a type of boundary condition where the values of first derivatives at endpoints of data points are specified. Using this condition, it allows us to manage the behaviour of the splines at boundaries more than the natural cubic spline. One key difference which differentiates both these conditions is that the second derivative of the natural splines are set at 0 for the endpoints whereas for the clamped condition it is the first derivative. Here is an example of a cubic spline with the clamped condition:

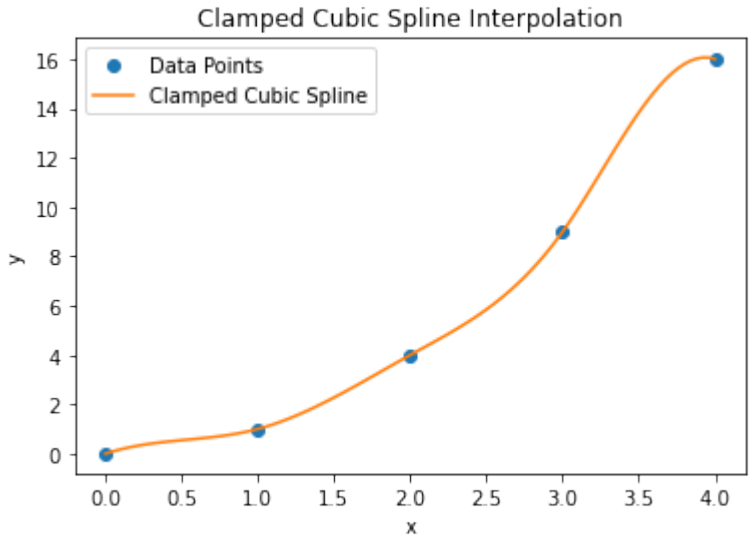

Figure 3.2: This shows a cubic spline with the clamped condition

Let us say we have a set of data points: (x_0, y_0), (x_1, y_1), ..., (x_n, y_n), and we want to use cubic splines to interpolate this data, the clamped condition specifies the values of the first derivatives $S'(x_0)$ and $S'(x_n)$ at the endpoints x_0 and x_n, respectively. The condition is usually written as: $S'(x_0) = m_0$ and $S'(x_n) = m_n$ where m_0 and m_n are specified values for the first derivatives. In addition, clamped cubic splines also satisfy the conditions $S'(x_0) = m_0$ and $S'(x_n) = m_n$, which gives us more control over the shape of the spline near boundaries of each data point.

Construction of the Clamped Cubic Spline

To obtain the spline from figure 3.2, we must firstly import the same packages as we did for the natural spline. For simplicity and to avoid any obscurities the same array from the natural spline is used. The code then specifies the

values of the first derivative at the endpoints using $m0 = 2$ and $mn = -2$. The $m0$ represents the derivative at the start point $x[0]$ and mn represents the derivative at the end point $x[-1]$. Then compute the cubic spline in the same fashion as the natural spline: $cs = CubicSpline(x, y, bc\_type = ((1, m0), (1, mn)))$ except there is no optional argument for a clamped spline as there is for natural so we instead write (1, $m0$), (1, mn)) - this just ensures that the first derivatives at the endpoints should be set to $m0$ and mn respectively, satisfying the conditions of a clamped boundary. We then evaluate the clamped spline in the same way as we did with the natural spline with $x\_eval = np.linspace(0, 4, 100)$ and $y\_eval = cs(x\_eval)$. The rest of the code is pretty much the same as the natural spline for printing coefficients and plotting the graphs. The final result should be a curve with a smooth and continuous line at the breakpoints, which are the points where the polynomial segments are joined together.

3.2.3 Natural Splines vs Clamped Splines

Although they do have their differences, let us take a look at which one we will need for solving Runge's phenomenon.

The main difference between natural splines and clamped splines is the type of boundary conditions they impose at the endpoints. As Runge's example consists of high order polynomials producing oscillating behaviour, it would be a good idea to use the natural cubic spline interpolation because it reduces the potential for larger errors far from the interpolated points. We could however use the clamped condition but for this example, the natural condition would be much more suitable.

3.3 Python Implementation of the Natural Cubic Spline Interpolating Runge's Example

We can use python effectively to construct code for interpolating Runge's Phenomenon. Here is the completed code for this:

```python
import numpy as np
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt

# Define a function to create a cubic spline interpolant
# Inputs:
# f: Function to be interpolated
# n: Number of equispaced points for interpolation
# Outputs:
# cs: Cubic spline interpolant

def defSpline(f, n):

    x = np.linspace(-1, 1, n) # Generate equispaced points in the range of -1 to 1
    y = f(x) # Evaluate the function at the equispaced points

    cs = CubicSpline(x, y, bc_type='natural') # Create a cubic spline interpolant with natural boundary conditions
    return cs

def runge(x):
    return 1 / (1 + 25 * x**2) # Defining the Runge function

# Generate cubic spline interpolants with 7 and 11 equispaced points
S7 = defSpline(runge, 7)
S11 = defSpline(runge, 11)

x_plot = np.linspace(-1, 1, 500) # Generate x values for plotting


# Evaluate the cubic spline interpolants and the Runge function at x_plot
y_plot_S7 = S7(x_plot)
y_plot_S11 = S11(x_plot)
y_plot_runge = runge(x_plot)

# Create a plot of the Runge function and the cubic spline interpolants
plt.figure(figsize=(15,10))
plt.plot(x_plot, y_plot_runge, label='Runge Function')
plt.plot(x_plot, y_plot_S7, label='S7 Cubic Spline')
plt.plot(x_plot, y_plot_S11, label='S11 Cubic Spline')
plt.scatter(S7.x, runge(S7.x), color='red', marker='o', label='7 Equispaced Points')
plt.scatter(S11.x, runge(S11.x), color='blue', marker='s', label='11 Equispaced Points')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Natural Cubic Spline Interpolation of Runge Function')
plt.show()
```

Figure 3.3: The completed code for a cubic spline interpolating Runge

3.3.1 How the Code was Developed and Each Step Broken Down

1. firstly, we want to import three python libraries - NumPy for numerical operations, SciPy for cubic spline interpolation, and Matplotlib for plotting graphs.

2. We want to define a function, we will call this $defSpline()$. This function will take two inputs, f which will be interpolated, and n which represents the number of equispaced points. The function will

then generate equispaced points for x in the interval -1 to 1 using the $np.linspace()$ command. It then evaluates f at these equispaced points to get our y. Then the function will have to create a cubic spline interpolant, cs, using the $CubicSpline()$ from SciPy library. Additionally, with x and y as inputs and set the boundaries to natural by adding the argument $bc\_type = natural$.

3. We then want to define Runge's function we will call this $Runge()$. This function will take input x and return the value of the Runge at that point. Runge's function is defined as:

$$f(x) = 1/(1 + 25x^2) \tag{3.1}$$

4. Generate cubic spline interpolated points. This is done by calling the $defSpline()$ function twice with different inputs to generate two cubic spline interpolations. We call this S7 for 7 equispaced points and S11 for 11 equispaced points. This passes the $runge()$ function as an argument to $defspline()$ to identify the function to be interpolated.

5. The code generates 500 equispaced points in the interval -1 to 1 using the linspace command and then stores these points in the x_plot as these will be used later for plotting the cubic spline.

6. The code assesses the interpolants S7 and S11 using the input values in x_plot to evaluate them at those points. These values are then stored in the y_plot_s7 and y_plot_s11.

7. We then create a plot using the Matplotlib library with an appropriate size. This then plots the Runge function, cubic spline s7 and s11 points, and the equispaced points.

3.3.2 The Result For Interpolating Runge's Function

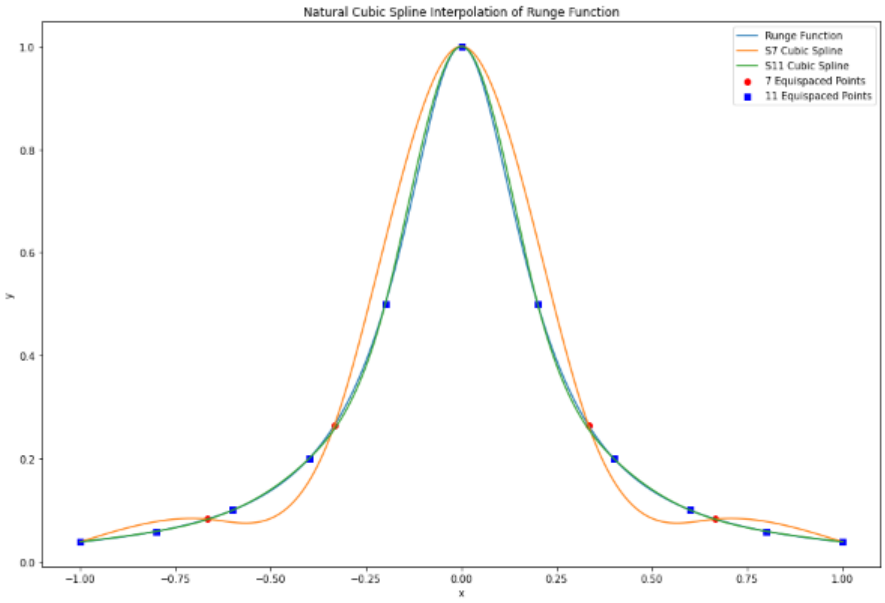

Figure 3.4: This shows the natural cubic spline interpolating Runge's function

In the plot, you can see the Runge function curve, which is a smooth curve with oscillations near the edges of the interval [-1, 1]. The cubic spline interpolants (S7 and S11) are shown to be smooth curves that pass through the equispaced points. (Red circles for 7 points and blue squares for 11 points).

In summary, the plotted graph visually demonstrates how the cubic spline interpolants (S7 and S11) are used to approximate the Runge function, employing natural boundary conditions. Furthermore, the plot demonstrates the impact of the number of equispaced points used for interpolation on the accuracy of the interpolation, particularly near the edges of the interval [-1, 1].

3.4 Lagrange Interpolation

We are asked to compute the Lagrange interpolating polynomials $P_6(x)$ and $P_{10}(x)$ with degrees of 6 and 10 respectively. When computed in python the result we obtained was this graph:

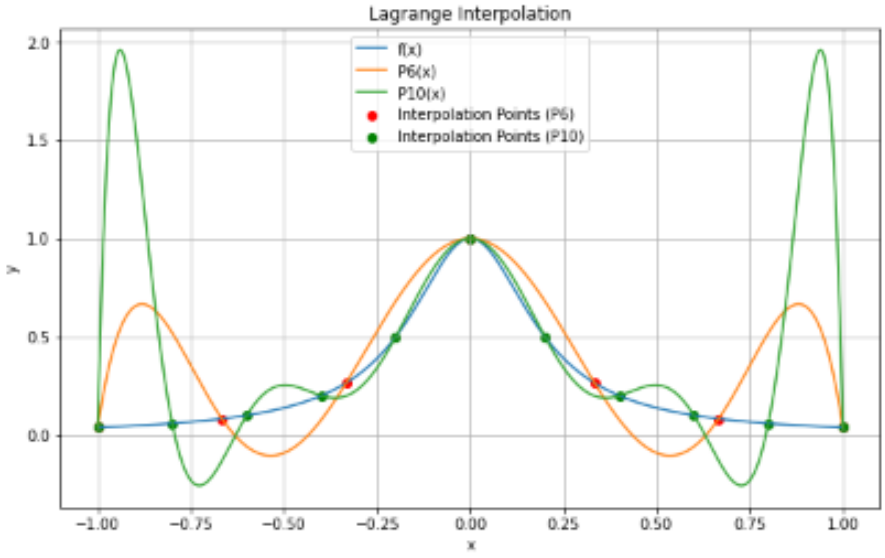

Figure 3.5: Lagrange's interpolation of degrees 6 and 10

The blue curve represents the original function $f(x) = 1/(1 + 25 * x^2)$, which is a standard example of a Runge's function. It is a smooth function with a peak near the edges of the interval [-1, 1].

The orange curve represents the Lagrange interpolating polynomial P6(x) with degree 6, which is obtained by interpolating the function f(x) at 7 equispaced points in the interval [-1, 1]. The interpolation points are marked on the graph with red dots. The orange curve tries to approximate the shape of the original function f(x) by passing through these interpolation points.

The green curve represents the Lagrange interpolating polynomial P10(x) with degree 10, which is obtained by interpolating the function f(x) at 11 equispaced points in the interval [-1, 1]. The interpolation points are marked on the graph with green dots. The green curve tries to approximate the shape of the original function f(x) more closely by passing through these additional interpolation points.

Also, it is important to note that as the degree of the Lagrange polynomial increases, the more the curve tends to oscillate more near the edges of the interval which, as aforementioned, correlates with Runge's phenomenon! This is shown in the graph where orange curve exhibits more oscillations compared to the green curve. This is a common issue with interpolation and can be mitigated by using other methods.

3.5 Cubic Spline vs The Lagrange

We are now asked to plot the function $f(x)$ and $P_6(x)$ and $S_7(x)$ on a single graph, and on a separate graph, $f(x)$, $P_{10}(x)$ and $S_{11}(x)$.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline
from numpy.polynomial.polynomial import Polynomial

def f(x):                                          # Define the original function f(x)
    return 1 / (1 + 25 * x**2)

def defSpline(f, n):                               # Define the cubic spline function defSpline(f, n)
    x = np.linspace(-1, 1, n)
    y = f(x)
    cs = CubicSpline(x, y, bc_type='natural')
    return cs

def plot_graph(x, y1, y2, y3, title):              # Plot the graphs
    plt.figure(figsize=(15,10))
    plt.plot(x, y1, label='f(x)')                  # Plot the functions
    plt.plot(x, y2, label='Cubic Spline')
    plt.plot(x, y3, label='Lagrange Interpolation')
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(title)
    plt.grid(True)
    plt.show()

x = np.linspace(-1, 1, 1000)                       # Generate x-values

# Plot f(x), S7(x), and P6(x)
S7 = defSpline(f, 7)                               # Calculate cubic spline and Lagrange interpolation For 7 equispaced points
P6 = Polynomial.fit(np.linspace(-1, 1, 7 + 1), f(np.linspace(-1, 1, 7 + 1)), 6)
plot_graph(x, f(x), S7(x), P6(x), 'Cubic Spline vs Lagrange Interpolation (7 Equispaced Points)')

# Plot f(x), S11(x), and P10(x)
S11 = defSpline(f, 11)                             # Calculate cubic spline and Lagrange interpolation For 11 equispaced points
P10 = Polynomial.fit(np.linspace(-1, 1, 10 + 1), f(np.linspace(-1, 1, 10 + 1)), 10)
plot_graph(x, f(x), S11(x), P10(x), 'Cubic Spline vs Lagrange Interpolation (11 Equispaced Points)')
```

Figure 3.6: Code for Lagrange and Cubic spline comparisons

3.5.1 How the Code was Assembled

We first define the original function f(x), The function f(x) is defined, which represents the original function we want to interpolate. In this case, it is defined as$1/(1 + 25 * x * *2)$. Then we define the cubic spline function $defSpline(f, n)$ to calculate the cubic spline of a given function f with n equispaced points. It takes the function f and the number of equispaced points n as input. Then we generate the X-values; the numpy function $np.linspace()$ is used to generate an array of x-values that range from -1 to 1 with 1000 points (1000 points allows for a smooth and visually appealing plot of the functions). These x-values will be used for plotting the functions.

The code uses the $defSpline()$ function to calculate cubic spline interpolants for two sets of equispaced points, one with 7 points and the other with 11 points. The resulting cubic spline functions are stored in S7 and S11 variables. Also, the code uses the $Polynomial.fit()$ function to perform Lagrange interpolation and obtain polynomial functions of degree 6 and 10 using the equispaced points. The resulting Lagrange polynomial functions are stored in P6 and P10 variables. We then plot the graphs appropriately.

3.5.2 Our Result From the Cubic Spline vs Lagrange Case

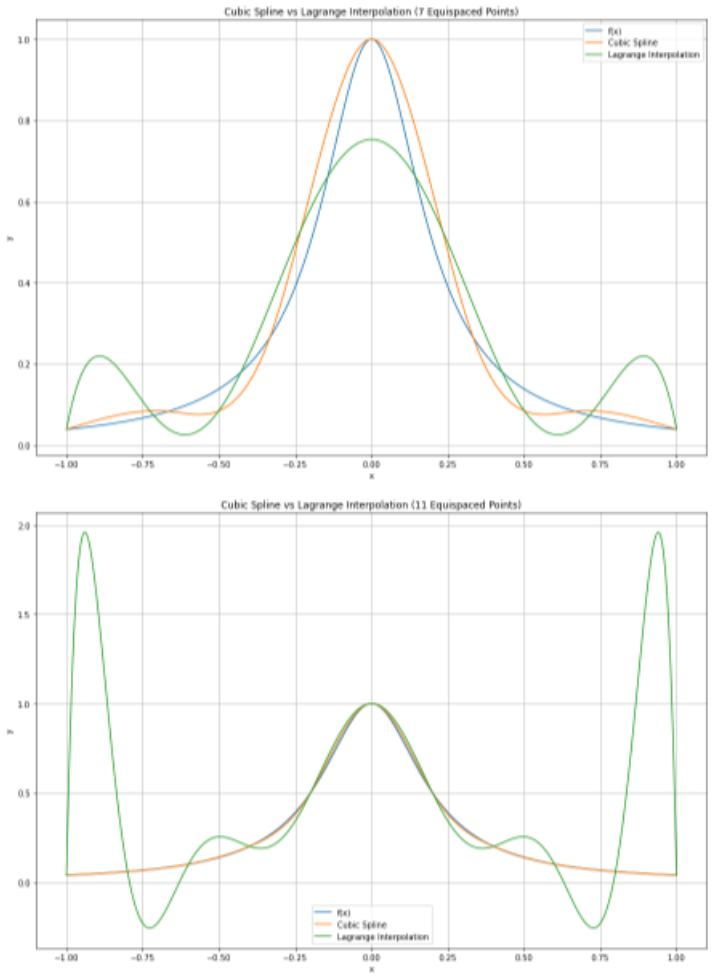

Figure 3.7: Two graphs. The first graph compares the functions f(x), S7(x) (cubic spline with 7 equispaced points), and P6(x) (Lagrange interpolation with 7 equispaced points). The second graph compares the functions f(x), S11(x) (cubic spline with 11 equispaced points), and P10(x) (Lagrange interpolation with 11 equispaced points).

Upon observing the graphs, it is evident that there is a stark contrast between the two interpolation methods. We can see how cubic spline follows the original function of the Runge function really well. However, we can see the uncontrollable oscillation of the Lagrange in the second graph towards the endpoints especially with higher degree polynomials. This ties back to Runge's Phenomenon where it can be seen in the graph with Lagrange interpolation using 6 and 10 equispaced points.

The cubic spline with natural boundary conditions offers a smoother fit to the data, resulting in a visually pleasing and accurate interpolation. The natural boundary conditions ensure that the cubic spline function smoothly connects the data points, leading to a smoother and more continuous interpolation. This is evident in the graph where S7(x) and S11(x) closely follow the shape of the original function f(x) without exhibiting abrupt changes or oscillations. To conclude, the cubic spline interpolation method is generally considered to be a better choice for interpolating functions with complex or rapidly changing behavior.

4

Procedure C: Chebyshev polynomial approximation

4.1 What are Chebyshev polynomials?

Chebyshev polynomials are a family of orthogonal polynomials which form an orthogonal set on the interval [-1, 1]. The weight function is written primarily as:

$$w(x) = (1 - x^2)^{(-1/2)} \tag{4.1}$$

Also, for a weight function on an interval let's say, for example, [a, b], the inner products f(x) and g(x) can be defined like so:

$$(f, g)_w = \int_a^b f(x)g(x)w(x)\ dx \tag{4.2}$$

The weight function determines the behavior of the polynomials orthogonal with respect to this inner product. According to **Iserles, Arieh** [4], for Chebyshev polynomials, we have the explicit formula:

$$T_n(x) = cos(n\ arccos\ x),\ n \geq 0 \tag{4.3}$$

Thus, we get:

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_2(x) = 2x^2 - 1, \quad T_3(x) = 4x^3 - 3x, \quad ... \tag{4.4}$$

He also states in his research that the T_n is a polynomial of degree n and is known as the nth Chebyshev polynomial. This is a great example of the first definition of Chebyshev polynomials. These polynomials satisfy the recursion formula $T_n(x) = 2xT_{n-1}(x)$ - $T_{n-2}(x)$ with initial values $T_0(x) = 1$ and $T_1(x) = x$.

Chebyshev polynomials of the second kind has a slightly different modified function. Although they satisfy the same weight function as the first kind (see 4.1) and are on the same interval, the second kind is denoted by $U_n(x)$.

The first seven Chebyshev polynomials of the second kind are:

$$U_0(x) = 1$$

$$U_1(x) = 2x$$

$$U_2(x) = 4x^2 - 1$$

$$U_3(x) = 8x^3 - 4x$$

$$U_4(x) = 16x^4 - 12x^2 + 1$$

$$U_5(x) = 32x^5 - 32x^3 + 6x$$

$$U_6(x) = 64x^6 - 80x^4 + 24x^2 - 1$$

$$U_7(x) = 128x^7 - 192x^5 + 80x^3 - 8x$$

4.1.1 Application of Chebyshev polynomials

Chebyshev polynomials have a wide range of various different applications in fields of mathematics. Some of these applications include polynomial approximation, numerical integration and spectral methods. We will take a brief look to how these polynomials are applied.

Polynomial Approximation

On a given integral, Chebyshev polynomials are often used in polynomial approximation specifically when the function has singularities or other demanding features. The Chebyshev approximation allows a way to efficiently and accurately approximate a function using small polynomial terms. Which allows us to get a desired result in numerical methods for funcion approximation in numerical analysis, signal processing and loads of other avenues.

Spectral Methods

Spectral methods are numerical methods for solving different differential equations, and Chebyshev polynomials are great for these. Spectral based on Chebyshev provide very high accuracy with little room of error and fast convergence rates. They are pinncale for solving ordinary differential equations, partial differential equations, and integrals.

Numerical Integration

In numerical integration, we can use these polynomials to to create quadrature (process of determining area) rules which, again like these other methods, provide accurate results for different integration on a given interval. These work specifically for integrals with singularities.

Control Theory

Chebyshev polynomials are vital for control theory for system identification, controller design and model reduction. The polynomials can be used to approximate system responses and analyse system perforomance and stability.

Quantum Mechanics

In quantum mechanics, we can use Chebyshev to solve wave function problems, different energy levels and scattering processes. They are accurate in describing the quantum behavior of physical systems. They are also able to solve Schrödinger's Equation numerically where this equation is used to describe the behaviour of quantum systems.

Schrödinger's Equation is given by:

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},t) = \hat{H}\Psi(\mathbf{r},t) \tag{4.5}$$

4.2 Subroutine for Chebyshev polynomials

We are asked to create a procedure in Python which given N as input returns the Chebyshev polynomials up to the degree N. We can produce a way to do this in Python. Here is a way we can do this:

1. Firstly, we want to initialise an empty list - lets call it T - to store the Chebyshev polynomials.
2. We then want to initialise the first two polynomials: $T_0(x) = 1$ and $T_1(x) = x$, and append them to the list T.
3. Then create an iteration to loop over each degree i from 2 to N.
4. Then calculate the coefficients of the Chebyshev polynomial of degree i, denoted as $T_i(x)$
5. After it has finished looping, the final result should be that T contains the Chebyshev polynomials $T_0(x)$ to $T_N(x)$.

For this, we are assuming N ≥ 1 because the first two polynomials are fixed ($T_0(x) = 1$ and $T_1(x) = x$). If a user uses a input which does not satisfy this, then appropriate error handling will be applied.

4.2.1 First Attempt at Creating the Subroutine

The first attempt, there were a few error messages appearing which indicated the code was inaccurate as also there were no results outputted. Here is the first draft:

```
def chebyshev_polynomials(N):

    T = [[1], [0, 1]]  # Initialise the list to store the Chebyshev polynomials with the first two polynomials
    for i in range(2, N + 1):

        if i == 3:
            T_i = [-2 * T[i - 1][0], 0] + [2 * T[i - 1][j - 1] - T[i - 2][j - 2] for j in range(2, i)]
            T_i = [-2 * T[i - 1][0], 0] + [2 * T[i - 1][j - 1] - T[i - 2][j] for j in range(2, i + 1)]
        T.append(T_i)
    return T

# example
N = 5  # Maximum degree of Chebyshev polynomials
T = chebyshev_polynomials(N)  # Call the subroutine to calculate the polynomials

# Print the Chebyshev polynomials
for i, t in enumerate(T):
    print("T_{}(x): {}".format(i, t))
```

Figure 4.1: First attempt at tackling the subroutine for Chebyshev polynomials

Following the step by step plan in the previous subsection, this is what was produced in python, but the output we got was:

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_61240/3205364301.py in <module>
     12 # example
     13 N = 5  # Maximum degree of Chebyshev polynomials
---> 14 T = chebyshev_polynomials(N)  # Call the subroutine to calculate the polynomials
     15
     16 # Print the Chebyshev polynomials

~\AppData\Local\Temp/ipykernel_61240/3205364301.py in chebyshev_polynomials(N)
      7             T_i = [-2 * T[i - 1][0], 0] + [2 * T[i - 1][j - 1] - T[i - 2][j - 2] for j in range(2, i)]
      8             T_i = [-2 * T[i - 1][0], 0] + [2 * T[i - 1][j - 1] - T[i - 2][j] for j in range(2, i + 1)]
----> 9         T.append(T_i)
     10     return T
     11

UnboundLocalError: local variable 'T_i' referenced before assignment
```

Figure 4.2: Error message appearing after code has been ran

Identifying and Correcting the Errors

After carefully analysing the code, three major errors were found hindering our result.

In Line 7 where it reads $T_i = [-2*T[i-1][0], 0]+[2*T[i-1][j-1]-T[i-2][j-2]$ for j in range$(2, i)]$, the range j only iterates from $j = 2$ to $j = i - 1$ instead of $j = 2$ and $j = i$. This results in missing the last coefficient of $T_i(x)$ which gave us the error. The correct code should read range(2, i + 1) to correctly include the last element in the range.

We were given another "IndexError" from Line 8 where we wrote $T_i = [-2*T[i-1][0], 0]+[2*T[i-1][j-1]-T[i-2][j]$ for j in range$(2, i+1)]$. The part where it says $T[i-2][j]$ is giving us our error because the coefficient of x^j in $T_i(x)$ will be calculated using the wrong coefficient from $T[i-2]$, resulting in an incorrect Chebyshev polynomial.

The last error that has been identified was just a simple "IndentationError" where a missing indent was not placed in Line 9 where it reads $T.append(T_i)$.

4.2.2 A Working Subroutine!

Here is a final version of the code with working functionality:

```
def chebyshev_polynomials(N):

    T = [[1], [0, 1]]  # Initialse the list to store the Chebyshev polynomials with the first two polynomials
    for i in range(2, N + 1):
        T_i = [-2 * T[i - 1][0], 0] + [2 * T[i - 1][j - 1] - T[i - 2][j - 2]
    for j in range(2, i + 1)]
        T.append(T_i)  # Add the coefficients of Chebyshev polynomial of degree i to the list T
    return T

# Example N = 7
N = 7  # Maximum degree of Chebyshev polynomials
T = chebyshev_polynomials(N)  # Calls the subroutine to calculate the polynomials

# Print the Chebyshev polynomials
for i, t in enumerate(T):
    print("T_{}(x): {}".format(i, t))
```

Figure 4.3: Working subroutine for Chebyshev polynomials

The code here slightly differs from the one in Figure 4.1 as we have made it more simple and easier to understand by just creating two separate iterative loops on separate lines. The idea is the exact same as the first attempt except here, all the errors have been amended and the code provides results. Here are the results when we let N be 7 for example:

```
T_0(x): [1]
T_1(x): [0, 1]
T_2(x): [-1, 0, 2]
T_3(x): [0, -2, 0, 4]
T_4(x): [1, 0, -4, 0, 8]
T_5(x): [0, 3, 0, -8, 0, 16]
T_6(x): [-1, 0, 7, 0, -16, 0, 32]
T_7(x): [0, -4, 0, 15, 0, -32, 0, 64]
```

Figure 4.4: Results obtained when running the code for the Chebyshev polynomial subroutine

The output here shows the coefficients of the Chebyshev polynomials up to degree 7 (i.e., $T_0(x)$ to $T_7(x)$). The coefficients are listed in the form $[c_0, c_1, c_2, ..., c_n]$, where c_i represents the coefficient of x^i in the polynomial $T_i(x)$.

4.3 Using Chebyshev Procedure to Approximate Runge's Function

We can write a procedure called $defCheb(f, N)$ which return Chebyshev polynomials of degree f and approximates N when given a function on the interval (-1,1). Using our subroutine from the previous section, Figure 4.3, to create a full procedure. We are asked to also plot the graphs of Runge's function $f(x) = 1/(1 + 25x^2)$ with degree 6 and 10. The code for this can be found in the appendix.

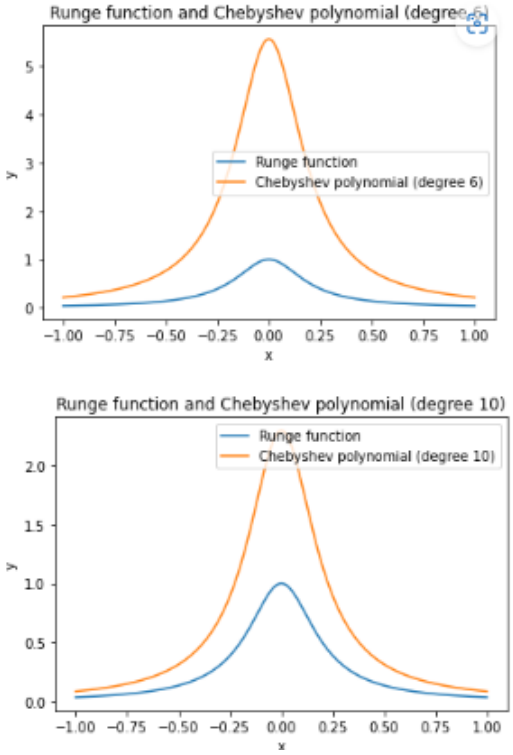

Figure 4.5: Graphs of Runge and Chebyshev approximations

The results of the Chebyshev polynomial approximation for the Runge function using degrees N1 = 6 and N2 = 10 are plotted and displayed in two separate graphs.

The first graph shows the Runge function and the Chebyshev polynomial approximation of degree 6. It can be observed that the Chebyshev polynomial approximation does not really follow the shape of the Runge function in the interval (-1, 1).

The second graph on the other hand, shows the Runge function and the Chebyshev polynomial approximation of degree 10. Compared to the approx-

imation of degree 6, the approximation of degree 10 shows reduced oscillations near the edges of the interval, indicating that higher degree Chebyshev polynomials provide a better approximation to the Runge function in this case. When lookin back at the Lagrange case for Runge's function with 10 degrees (second graph from Figure 3.7) we can clearly see major oscillations towards the starting and endpoints of the graph. This indicates that the Chebyshev polynomial for higher degrees is a better method for approximating Runge. However it is importnt to note that higher degree polynomials may provide better approximation in some cases, but they can also suffer from increased oscillations and numerical instability. Therefore, careful consideration of the degree is needed to achieve an optimal approximation result.

5

Procedure D: Double Integrals

5.1 Introduction to Double Integrals in Numerical Analysis

In numerical analysis, we can use double integrals to find an approximation of a definite integral of a two-dimensional function over a given region in the xy-plane. They have a wide range of various of applications including engineering, economics. There are a lot of different rules/methods for approximating double integrals. Here are some of the common methods:

- Rectangular Rule
- Trapezoidal Rule
- Simpson's Rule

5.1.1 The Rectangular Rule

The rectangular rule is probably one of the most basic rules for approximating double integrals. It is very simple; all it does is it divides the region of integration into different equal sized rectangles and evaluates the corner of each rectangle. The area of each rectangle is multiplied by the function value to obtain an approximation of the integral.

However, this method albeit being the simplest, has a lot of cause for errors. It can be less accurate for functions with rapidly changing values or for curved boundary regions.

5.1.2 The Trapezoidal Rule

This rule is simply just an extension of the rectangular rule where instead of having rectangles, they are replaced by a trapezoid. We evaluate the function at the endpoints of every trapezoid. Then the area of each trapezoid is calcualted as an average of the function value at each endpoint. Then we calculate the sum of these areas, it provides an approximation of the double integral. Similarly with the rectangular rule, the trapezoid may also be limited for functions with rapid changes or regions with curved boundaries

5.1.3 Simpson's Rule

Simpson's rule is the far more superior method out of the three for approximating double integrals. It uses quadratic polynomials (parabolic arcs) to

approximate the function over each interval of the region. The function is evaluated at the endpoints and midpoints of each interval. The area under the parabolic arcs passing through these points is then calculated and then the sum of these areas gives us an approximation of the double integral. In comparison to the other two rules, Simpson's rule provides a much higher accuracy with little room of error especially when dealing with functions with curved regions.

Here is the basic Simpson's rule formula:

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f(a) + 4f\left(a + \frac{h}{2}\right) + f(b)\right) \tag{5.1}$$

According to **Weisstein, E. W** [8] Simpson's rule can be derived by integrating a third-order Lagrange interpolating polynomial fit to the function at three equally spaced points. Which is relevant as we can use this in interpolation.

5.2 Solving Double Integrals

Let us consider double integral:

$$\int\int_R f(x, y)\, dx\, dy,$$

where R is a rectangular region given by $R = \{(x, y) \,|\, a \leq x \leq b,\, c \leq y \leq d\}$.

We are to develop a formula for solving this integral starting with Simpson's rule. We are asked to specifically use the rule for "standard" integrals which is shown in Equation 5.1

5.2.1 Developing the Formula for Solving the Double Integral

To use Simpson's rule for solving a double integral, we can break down the rectangular region R into smaller sub-rectangles, and then apply Simpson's rule separately to each sub-rectangle.

Assuming we have a set of equally spaced points in the x-direction denoted by x_0, x_1, ..., x_N, where x_0 represents the lower limit a and x_N represents the upper limit b. Similarly, we have another set of equally spaced points in the y-direction denoted by y_0, y_1, ..., y_M, where y_0 represents the lower limit c and y_M represents the upper limit d. We can then define the step sizes in the x and y directions separately as $h_x = (b - a)/N_x$ and

$h_y = (d-c)/N_y$, where N_x and N_y are the number of intervals in the x and y directions, respectively.

Now, we can define the function $f(x,y)$ at each grid point (x_i, y_j) as $f_{ij} = f(x_i, y_j)$, where $0 \leq i \leq N_x$ and $0 \leq j \leq N_y$. And where $x_i = a + i * h_x$ and $y_j = c + j * h_y$ for $i = 0, 1, ..., N_x$ and $j = 0, 1, ..., N_y$. The double integral can then be approximated as:

$$\int\int_R f(x,y)dxdy \approx \frac{h_x \cdot h_y}{9}\Bigg(f(a,c) + f(a,d) + f(b,c) + f(b,d) + 4\sum_{i=0}^{N_x}\sum_{j=0}^{N_y} f(x_i, y_j) + 2\sum_{i=0}^{N_x}\sum_{k=0}^{N_y} f(x_i, y_k) + 2\sum_{j=0}^{N_x}\sum_{i=0}^{N_y} f(x_j, y_i)\Bigg)$$

In this context, Σ represents a summation over the relevant indices, while f(a, c), f(a, d), f(b, c), and f(b, d) correspond to the function values at the corners of the rectangular region R. With extensive research from **Cheney, E Ward and Kincaid, David R** [3], **Chapra, Steven C and Canale, Raymond P** [2] we were able to discover a formula for solving the double integral using Simpson's rule for standard integrals.

5.3 Python Procedure for Simpson's Rule

We are asked to develop a procedure with defining function $defMultSim(a, b, c, d, f)$ which takes as input the extremes of R and a function in two variables f. Then must return an approximate double integral:

$$\int\int_R f(x,y)\,dx\,dy \tag{5.2}$$

We can break this down in simple steps so we will fully understand how it can be implemented in Python.

5.3.1 The Procedure Broken Down in to Steps

Here is a rough idea on how we can write this up in Python:

1. Firstly, we want to define the function $f(x,y)$ that we want integrated over a specific rectangular region.

2. We then want to define the region, call it R, by specifying the lower and upper limits of x and y which are written as a, b, c, d respectively.

3. We create step sizes in each directions, x and y. This will be denoted by h_x and h_y respectively.

4. Initialise the integral variable to zero, which will be used to accumulate the result. We do this because it serves as an accumulator that will store the cumulative result of the double integration using Simpson's rule.

5. Use loops to iterate over the intervals in both x and y directions.

6. We then compute the coefficient.

7. Then we multiply the coefficient with the value of the function $f(x_i, y_j)$ evaluated at the current coordinates x_i and y_j, and add the result to the integral variable.

8. Multiply integral by $(h_x * h_y)/9$ to get the approximate double integral.

9. Finally, return the approximated result and print.

5.3.2 A Working Procedure!

In python, we were able to develop a procedure which solves the double integral:

$$\int \int_R e^{y-x}\, dx\, dy \tag{5.3}$$

Here is the full coding on how it was done with comments where necessary:

```
import math

def f(x, y):
    return math.exp(y - x)

def defMultSim(a, b, c, d, f):
    N_x = 10  # number of intervals in x direction
    N_y = 10  # number of intervals in y direction

    h_x = (b - a) / N_x  # step size in x direction
    h_y = (d - c) / N_y  # step size in y direction

    integral = 0

    for i in range(N_x + 1):
        for j in range(N_y + 1):
            x_i = a + i * h_x
            y_j = c + j * h_y
            coefficient = 1

            # Update coefficient based on position in grid
            if i == 0 or i == N_x:
                coefficient *= 0.5
            if j == 0 or j == N_y:
                coefficient *= 0.5
            if i % 2 == 1 and j % 2 == 1:
                coefficient *= 4
            elif i % 2 == 0 and j % 2 == 0:
                coefficient *= 16
            else:
                coefficient *= 8

            integral += coefficient * f(x_i, y_j)

    integral *= (h_x * h_y) / 9

    return integral

# Input values of the bottom and top of each integrand.
a = 0
b = 0.1
c = 0
d = 0.1


# Print the result
print("Approximate double integral using Simpson's rule:", defMultSim(a,b,c,d,f))
```

Figure 5.1: Working code for the example in the assignment

Breakdown of the Code

Firstly, the code defines two functions: $f(x, y)$ and $defMultSim(a, b, c, d, f)$. The $f(x, y)$ function takes two arguments, x and y, and calculates the value of the function $f(x, y) = exp(y - x)$, where $exp()$ is the exponential function taken from the math module in Python.

The $defMultSim(a, b, c, d, f)$ function takes five different arguments. Arguments a and b represent the lower and upper bounds in the x-direction

respectively, and c and d are the lower and upper bounds in the y-direction. The f argument represents the function that is to be integrated over the given interval.

In our function, we used nested loops to perform approximation of the integral using Simpson's rule for standard integrals. The outer loop iterates over i from 0 to N_x and the inner loops over j from 0 to N_y. In this case, N_x and N_y are the number of intervals in the x and y direction respectively.

For each combination of i and j, the code calculates the values of x_i and y_j which represent the current positions in the x and y directions, respectively, using the given formulas:

$$x_i = a + i * h_x \tag{5.4}$$

$$y_j = c + j * h_y \tag{5.5}$$

Where h_x and h_y are the step sizes in the x and y direction.

The code also calculates a coefficient for each combination of i and j based on their positions in the grid. The coefficients are used to weight the function values at x_i and y_j to obtain the approximation of the double integral.

The weighted sum of the function values at x_i and y_j is accumulated in the integral variable. Finally, the code multiplies the accumulated sum by a scaling factor $(h_x * h_y)/9$ and returns the result as the approximate double integral. We then print out the result using the $print()$ command with the conditions of the question applied.

The Result

The result we obtained was:

```
Approximate double integral using Simpson's rule: 0.010008669723922949
```

Figure 5.2: The double integral solved gives us this result

We can check to see if this result is correct using online tools for solving double integrals. The support software we used is a website called **Symbolab** [7]. We got:

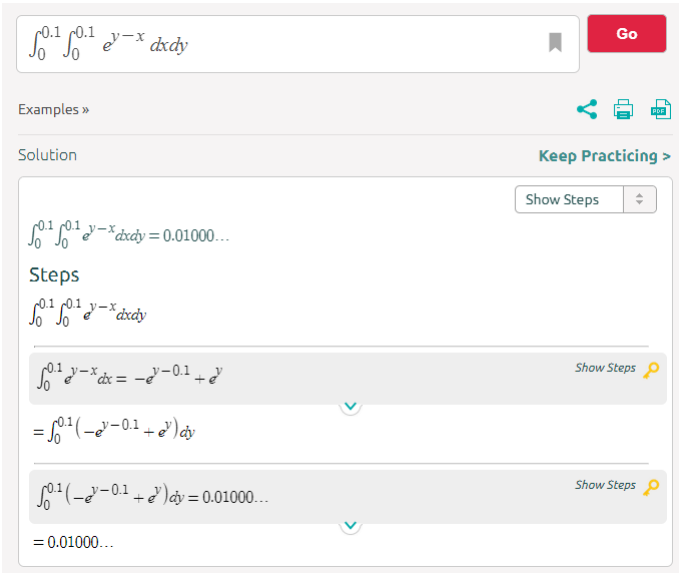

Figure 5.3: Double integral solved using a website tool

The result we obtained was:

$$\int\int_R e^{y-x}\,dx\,dy = 0.01000... \tag{5.6}$$

Which similarly is roughly the same as our result from the procedure, hence we can deduce that this procedure prints somewhat accurate results. You can argue and say our results were more accurate as python allows us to display many more decimal figures.

6

Appendix

Newton-Raphson Method

```
import math

def newton_raphson(f, f_prime, x0, tol=1e-6, max_iter=100):

    x = x0
    num_iter = 0

    while abs(f(x)) > tol and num_iter < max_iter:
        x = x - f(x) / f_prime(x)
        num_iter += 1

    return x, num_iter


# Define the function whose root needs to be found
def f(x):
    return x**3 - x - 1

# Define the derivative of the function f
def f_prime(x):
    return 3*x**2 - 1

# Initial guess for the root
x0 = 1.0

# Call the Newton-Raphson method
root, num_iter = newton_raphson(f, f_prime, x0)

# Print the result
print("Approximate root:", root)
print("Number of iterations:", num_iter)
```

Laguerre's Method

```
import math
from sympy import *
import numpy as np

x = symbols('x')

def lag(f, n, P, tol, iter):
    step = 1
    P0 = P
    f_diff = diff(f, x)
    f_second_diff = diff(f_diff, x)
    fx = lambdify(x, f)
    fx_diff = lambdify(x, f_diff)
    fx_second_diff = lambdify(x, f_second_diff)

    for step in range(iter):
        A = (fx_diff(P0)) / (fx(P0))
        B = A ** 2 - (fx_second_diff(P0) / fx(P0))
        if A > 0:
            d = n / ((A) + ((n - 1) * (n * B) - (A ** 2)) ** 0.5)
            if d <= tol:
                break
            else:
                P0 = P0 - d
                step = step + 1
                if step == 100:
                    break
        elif A < 0:
            d = n / ((A) - ((n - 1) * (n * B) - (A ** 2)) ** 0.5)
            if d <= tol:
                break
            else:
                P0 = P0 - d
                step = step + 1
                if step == 100:
                    break

    print("Approximate root is: ", P0)
    print("Number of iterations: ", step) s

lag(5 * x ** 5 + 2.3 * x ** 4 - 2 * x ** 2 + 6 * x - 3, 40, 100, 10 ** -4, 10)
```

Natural Cubic Spline

```
import numpy as np
from scipy.interpolate import CubicSpline

# Example data points
x = np.array([0, 1, 2, 3, 4])
y = np.array([0, 1, 4, 9, 16])

# Compute natural cubic spline
cs = CubicSpline(x, y, bc_type='natural')

# Evaluate the spline at a set of points
x_eval = np.linspace(0, 4, 100)
y_eval = cs(x_eval)

# Print the coefficients of the cubic polynomials
coefficients = cs.c
for i in range(len(coefficients)):
    print("S_{}(x) = {} + {}(x - {}) + {}(x - {})^2 + {}(x - {})^3".format(
        i, coefficients[i, 3], coefficients[i, 2], x[i],
        coefficients[i, 1], x[i], coefficients[i, 0], x[i]))

# Plot the data points and the cubic spline
import matplotlib.pyplot as plt
plt.plot(x, y, 'o', label='Data Points')
plt.plot(x_eval, y_eval, label='Natural Cubic Spline')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Natural Cubic Spline Interpolation')
plt.show()
```

Clamped Cubic Spline

```
import numpy as np
from scipy.interpolate import CubicSpline

# Example data points
x = np.array([0, 1, 2, 3, 4])
y = np.array([0, 1, 4, 9, 16])

# Values of first derivatives at endpoints
m0 = 2
mn = -2

# Compute clamped cubic spline
cs = CubicSpline(x, y, bc_type=((1, m0), (1, mn)))

# Evaluate the spline at a set of points
x_eval = np.linspace(0, 4, 100)
y_eval = cs(x_eval)

# Print the coefficients of the cubic polynomials
coefficients = cs.c
for i in range(len(coefficients)):
    print("S_{}(x) = {} + {}(x - {}) + {}(x - {})^2 + {}(x - {})^3".format(
        i, coefficients[i, 3], coefficients[i, 2], x[i],
        coefficients[i, 1], x[i], coefficients[i, 0], x[i]))

# Plot the data points and the clamped cubic spline
import matplotlib.pyplot as plt
plt.plot(x, y, 'o', label='Data Points')
plt.plot(x_eval, y_eval, label='Clamped Cubic Spline')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Clamped Cubic Spline Interpolation')
plt.show()
```

Interpolating Runge's Function

```
import numpy as np
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt

def defSpline(f, n):

    x = np.linspace(-1, 1, n)
    y = f(x)

    cs = CubicSpline(x, y, bc_type='natural')
    return cs

def runge(x):
    return 1 / (1 + 25 * x**2)


S7 = defSpline(runge, 7)
S11 = defSpline(runge, 11)

x_plot = np.linspace(-1, 1, 500)



y_plot_S7 = S7(x_plot)
y_plot_S11 = S11(x_plot)
y_plot_runge = runge(x_plot)

# Create a plot of the Runge function and the cubic spline interpolants
plt.figure(figsize=(15,10))
plt.plot(x_plot, y_plot_runge, label='Runge Function')
plt.plot(x_plot, y_plot_S7, label='S7 Cubic Spline')
plt.plot(x_plot, y_plot_S11, label='S11 Cubic Spline')
plt.scatter(S7.x, runge(S7.x), color='red', marker='o', label='7 Equispaced Points')
plt.scatter(S11.x, runge(S11.x), color='blue', marker='s', label='11 Equispaced Point
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Natural Cubic Spline Interpolation of Runge Function')
plt.show()
```

Lagrange's Interpolation

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.polynomial.polynomial import Polynomial


def f(x):
    return 1 / (1 + 25 * x**2)


degree6 = 6
degree10 = 10


x = np.linspace(-1, 1, 1000)
x_interp6 = np.linspace(-1, 1, degree6 + 1)
x_interp10 = np.linspace(-1, 1, degree10 + 1)


y = f(x)
y_interp6 = Polynomial.fit(x_interp6, f(x_interp6), degree6)(x)
y_interp10 = Polynomial.fit(x_interp10, f(x_interp10), degree10)(x)

plt.figure(figsize=(15,10))
plt.plot(x, y, label='f(x)')
plt.plot(x, y_interp6, label='P6(x)')
plt.plot(x, y_interp10, label='P10(x)')
plt.scatter(x_interp6, f(x_interp6), color='red', marker='o', label='Interpolation Po
plt.scatter(x_interp10, f(x_interp10), color='green', marker='o', label='Interpolatio
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Lagrange Interpolation')
plt.grid(True)
plt.show()
```

S7, P6, S11, P10 Graph

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline
from numpy.polynomial.polynomial import Polynomial

def f(x):
    return 1 / (1 + 25 * x**2)

def defSpline(f, n):
    x = np.linspace(-1, 1, n)
    y = f(x)
    cs = CubicSpline(x, y, bc_type='natural')
    return cs

def plot_graph(x, y1, y2, y3, title):
    plt.figure(figsize=(15,10))
    plt.plot(x, y1, label='f(x)')
    plt.plot(x, y2, label='Cubic Spline')
    plt.plot(x, y3, label='Lagrange Interpolation')
    plt.legend()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(title)
    plt.grid(True)
    plt.show()

x = np.linspace(-1, 1, 1000)
# Plot f(x), S7(x), and P6(x)
S7 = defSpline(f, 7)
P6 = Polynomial.fit(np.linspace(-1, 1, 7 + 1), f(np.linspace(-1, 1, 7 + 1)), 6)
plot_graph(x, f(x), S7(x), P6(x), 'Cubic Spline vs Lagrange Interpolation (7 Equispac

# Plot f(x), S11(x), and P10(x)
S11 = defSpline(f, 11)
P10 = Polynomial.fit(np.linspace(-1, 1, 10 + 1), f(np.linspace(-1, 1, 10 + 1)), 10)
plot_graph(x, f(x), S11(x), P10(x), 'Cubic Spline vs Lagrange Interpolation (11 Equis
```

Subroutine which given N returns Chebyshev polynomials up to the degree N

```
def chebyshev_polynomials(N):
    T = []  # Initialise the list to store the Chebyshev polynomials
    for i in range(N + 1):
        if i == 0:
            T.append([1])  # T_0(x) = 1
        elif i == 1:
            T.append([0, 1])  # T_1(x) = x
        else:
            T_i = [0] * (i + 1)
            T_i[0] = -T[i - 2][0]
            T_i[i] = 2 * T[i - 1][i - 1]  for i > 1
            for j in range(1, i - 1):
                T_i[j] = T[i - 1][j - 1] - T[i - 2][j]
            T.append(T_i)
    return T


N = 7
T = chebyshev_polynomials(N)


for i in range(N + 1):
    print("T_{}(x): {}".format(i, T[i]))
```

Chebysev approximating Runge's Function

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function to approximate
def f(x):
    return 1 / (1 + 25 * x**2)

# Define the Chebyshev polynomials subroutine
def chebyshev_polynomials(N):
    T = []  # Ilist to store the Chebyshev polynomials
    x = np.linspace(-1, 1, 1000)  # Generate 1000 equidistant points in the interval
    for i in range(N + 1):
        if i == 0:
            T.append(np.ones_like(x))  # T_0(x) = 1
        elif i == 1:
            T.append(np.copy(x))  # T_1(x) = x
        else:
            T_i = 2 * x * T[i - 1] - T[i - 2]  # Recurrence relation for Chebyshev po
            T.append(np.copy(T_i))
    return T

# Define the defCheb procedure
def defCheb(f, N):
    T = chebyshev_polynomials(N)
    a = -1  # Lower bound of the interval
    b = 1  # Upper bound of the interval
    x = np.linspace(a, b, 1000)
    c = (b - a) / 2
    p = np.zeros_like(x)
    for i in range(N + 1):
        p += f(x) * T[N][i]
    return x, p

# Plot the graphs of the Runge function and the approximating Chebyshev polynomials o
N1 = 6
x1, p1 = defCheb(f, N1)
plt.plot(x1, f(x1), label='Runge function')
plt.plot(x1, p1, label='Chebyshev polynomial (degree {})'.format(N1))
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Runge function and Chebyshev polynomial (degree {})'.format(N1))
```

```
plt.show()

N2 = 10
x2, p2 = defCheb(f, N2)
plt.plot(x2, f(x2), label='Runge function')
plt.plot(x2, p2, label='Chebyshev polynomial (degree {})'.format(N2))
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Runge function and Chebyshev polynomial (degree {})'.format(N2))
plt.show()
```

Simpson's Rule Procedure

```
import math

def f(x, y):
    return math.exp(y - x)

def defMultSim(a, b, c, d, f):
    N_x = 10  # number of intervals in x direction
    N_y = 10  # number of intervals in y direction

    h_x = (b - a) / N_x  # step size in x direction
    h_y = (d - c) / N_y  # step size in y direction

    integral = 0

    for i in range(N_x + 1):
        for j in range(N_y + 1):
            x_i = a + i * h_x
            y_j = c + j * h_y
            coefficient = 1

            # Update coefficient based on position in grid
            if i == 0 or i == N_x:
                coefficient *= 0.5
            if j == 0 or j == N_y:
                coefficient *= 0.5
            if i % 2 == 1 and j % 2 == 1:
                coefficient *= 4
            elif i % 2 == 0 and j % 2 == 0:
                coefficient *= 16
            else:
                coefficient *= 8

            integral += coefficient * f(x_i, y_j)

    integral *= (h_x * h_y) / 9

    return integral

# Input values of the bottom and top of each integrand.
a = 0
b = 0.1
c = 0
```

```
d = 0.1



# Print the result
print("Approximate double integral using Simpson's rule:", defMultSim(a,b,c,d,f))
```

References

[1] Burden, R. L. and Faires, J. D. (2005). Numerical analysis 8th ed. *Thomson Brooks/Cole*.

[2] Chapra, S. C. and Canale, R. P. (2015). Mathematical background. In *Numerical Methods for Engineers*, page 7. McGraw-Hill Education New York, NY, USA.

[3] Cheney, E. W. and Kincaid, D. R. (2012). *Numerical mathematics and computing*. Cengage Learning.

[4] Iserles, A. (2009). *A first course in the numerical analysis of differential equations*. Number 44. Cambridge university press.

[5] McKinley, S. and Levine, M. (1998). Cubic spline interpolation. *College of the Redwoods*, 45(1):1049–1060.

[6] Mehtre, V. V. and Sharma, S. (2019). Root finding methods: Newton raphson method. *International Journal for Research in Applied Science and Engineering Technology*, 7(11):411–414.

[7] Symbolab (2011). https://www.symbolab.com/solver/double-integrals-calculator.

[8] Weisstein, E. W. (2003). Simpson's rule. *https://mathworld. wolfram. com/*.