

Assignment 2

Due Date: September 22, 2025

1 - Lagrange Interpolation and Error Analysis

Let $f \in C^{n+1}[a, b]$, and let P_n be its Lagrange interpolating polynomial at the distinct nodes $x_0, x_1, \dots, x_n \in [a, b]$. The interpolation error is:

$$R_n(x) = f(x) - P_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^n (x - x_i), \quad \xi \in (a, b)$$

Suppose $f(x) = e^x$ on the interval $[0, 1]$, for equally spaced nodes $x_i = \frac{i}{n}$.

Question 1.1

Derive an explicit bound for the maximum error $\max_{x \in [0,1]} |R_n(x)|$.

Proof. To bound $|R_n(x)|$, we start with the error formula:

$$|R_n(x)| = \left| \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^n (x - x_i) \right|, \quad \xi \in (0, 1)$$

Note $\frac{1}{(n-1)!}$ is a constant. We can bound $R_n(x)$ by bounding $f^{(n+1)}(\xi)$ and $|\prod_{i=0}^n (x - x_i)|$ separately.

Now notice that $f^{(n+1)}(x) = f(x) = e^x$ for all n . Since e^x is strictly increasing on $[0, 1]$, we have that

$$f^{(n+1)}(\xi) = \max_{x \in [0,1]} |f^{(n+1)}(x)| = f(1) = e \quad (1)$$

Next, we need to bound $|\prod_{i=0}^n (x - x_i)|$. Let $\omega_n(x) = \prod_{i=0}^n (x - x_i)$. Note that $\omega_n(x)$ is a polynomial of degree $n+1$. Since $\omega_n(x)$ is continuous on the compact interval $[0, 1]$, it attains its maximum at some point in $[0, 1]$. We may write that:

$$|\omega_n(x)| \leq W_n \quad (2)$$

Where $W_n = \max_{x \in [0,1]} |\omega_n(x)| = |\omega_n(x^*)|$ for some $x^* \in [0, 1]$.

Now, we can use (1) and (2) to bound $|R_n(x)|$ from above:

$$|R_n(x)| \leq \frac{e}{(n+1)!} W_n$$

Since $R_n(x)$ is continuous on the compact interval $[0, 1]$, $\sup_{x \in [0,1]} R_n(x) = \max_{x \in [0,1]} R_n(x)$.

Since the supremum is the least upper bound, we have that:

$$\max_{x \in [0,1]} |R_n(x)| \leq \frac{e}{(n+1)!} W_n$$

Which gives us an explicit bound for the remainder. □

Question 1.2

Show the asymptotic decay of this error as $n \rightarrow \infty$.

Proof. From the previous part, we have that:

$$\max_{x \in [0,1]} |R_n(x)| \leq \frac{e}{(n+1)!} W_n$$

Let us first make a more crude estimate of W_n . Note that for any $x \in [0, 1]$ and for each i , we have:

$$|x - x_i| \leq 1$$

Since there are $n + 1$ terms in the product, we can write:

$$|\omega_n(x)| = \left| \prod_{i=0}^n (x - x_i) \right| \leq 1^{n+1} = 1$$

Therefore, we get the crude bound:

$$W_n = \max_{x \in [0,1]} |\omega_n(x)| \leq 1$$

Plugging this into our error bound, we get:

$$\max_{x \in [0,1]} |R_n(x)| \leq \frac{e}{(n+1)!}$$

Taking a limit as $n \rightarrow \infty$, we have:

$$\lim_{n \rightarrow \infty} \max_{x \in [0,1]} |R_n(x)| \leq \lim_{n \rightarrow \infty} \frac{e}{(n+1)!} = 0$$

Therefore, we have shown that the error decays to 0 as $n \rightarrow \infty$. □

2 - Interpolation Programming Exercise

Consider the function:

$$f(x) = \frac{1}{1 + 20x}, \quad x \in [-1, 1]$$

Question 2.1

Construct the interpolation polynomial $P_n(x)$ at equidistant nodes for $n = 5$.

Solution. The following methods were used to compute this polynomial:

Listing 1: 2.1 Python

```

1 import numpy as np
2
3 def f(x):
4     return 1 / (1 + 20 * x**2)
5
6 def lagrange_coefficients(nodes, function=f):
7     x = nodes
8     num_nodes = len(nodes)
9     # Add zeroth divided differences
10    dd_table = np.array([[function(xi) for xi in nodes]])
11
12    # Calculate divided difference table
13    for i in range(1, num_nodes):
14        ith_dd = np.zeros(num_nodes)
15
16        for j in range(num_nodes - i):
17            # Calculate ith divided differences
18            ith_dd[j] = (dd_table[i - 1, j + 1] - dd_table[i - 1, j]) / (
19                x[j + i] - x[j]
20            )
21
22        # Append the ith divided difference row to the table
23        dd_table = np.vstack([dd_table, ith_dd])
24
25    # Extract coefficients (first column of the table)
26    a = np.array([dd_table[i, 0] for i in range(dd_table.shape[0])])
27    return a
28
29 def generate_lagrange(nodes, degree):
30     # Get coefficients
31     a = lagrange_coefficients(nodes)
32     # Start with function and constant term
33     equation = f"P_{{{degree}}}(x) = {a[0]}"
34     w = []
35     # Build (x - xi) terms

```

```

36     for xi in nodes:
37         if abs(xi) <= 1e-14:
38             w.append('(x)')
39         elif xi < 0:
40             w.append(f"(x + {abs(xi)})")
41         else:
42             w.append(f"(x - {xi})")
43     b = ""
44
45     # Build polynomial string
46     for i in range(1, len(a)):
47         for j in range(i):
48             # Multiply (x - xi) terms
49             b += w[j]
50             # Multiply (x - xi) product with current coefficient and add term
51             if a[i] >= 0:
52                 equation += f" + {a[i]}{b}"
53             else:
54                 equation += f" - {abs(a[i])}{b}"
55         b = ""
56
57     return equation
58
59 # Main Method
60 if __name__ == "__main__":
61     # Generate and save equation to a text file
62     nodes = np.linspace(-1, 1, 6)
63     equation = generate_lagrange(nodes, 5)
64     with open("./plots_2/q2_1/p5.txt", "w") as file:
65         file.write(equation)

```

The interpolation polynomial $P_5(x)$ at equidistant nodes for $n = 5$ is given by:

$$\begin{aligned}
 P_5(x) = & 0.047619047619047616 \\
 & + 0.18583042973286878(x + 1.0) \\
 & + 1.1227255129694158(x + 1.0)(x + 0.6) \\
 & - 2.0647825525874324(x + 1.0)(x + 0.6)(x + 0.19999999999999996) \\
 & + 1.2904890953671473(x + 1.0)(x + 0.6)(x + 0.19999999999999996) \\
 & (x - 0.200000000000000018) \\
 & - 3.552713678800501e - 15(x + 1.0)(x + 0.6)(x + 0.19999999999999996) \\
 & (x - 0.200000000000000018)(x - 0.60000000000000001)
 \end{aligned}$$

We note that the leading coefficient is extremely close to 0, indicating that the polynomial is effectively of degree 4. The occurrence of the non-zero coefficient is likely due to floating point precision errors.

Question 2.2

Plot $f(x)$ and $P_n(x)$, and report the behavior of the interpolation error as n increases from 5 to 10 and 20.

Solution. In addition to the functions defined in the previous part, the following functions were used to help plot $f(x)$ and $P_n(x)$:

Listing 2: 2.2 Python

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def calculate_lagrange(nodes, a, x):
5      # Start with constant term
6      y = a[0]
7      # Build (x - xi) terms
8      w = [(x - xi) for xi in nodes]
9      # Temporary variable to hold (x - xi) product
10     b = 1
11
12     for i in range(1, len(a)):
13         for j in range(i):
14             # Multiply (x - xi) terms
15             b *= w[j]
16         # Multiply (x - xi) product with current coefficient
17         y += a[i] * b
18         b = 1
19
20     return y
21
22 def calculate_lagrange_output(nodes, x_coords=[], function=f):
23     # Get coefficients
24     a = lagrange_coefficients(nodes, function)
25     # If no x_coords provided, use nodes as x_coords
26     if len(x_coords) == 0:
27         x_coords = nodes
28     # Calculate y coordinates for each x coordinate
29     y_coordinates = np.array([calculate_lagrange(nodes, a, x) for x in x_coords
30 ])
31
32     return y_coordinates
33
34 # Main Method
35 if __name__ == "__main__":
36     # <some matplotlib styling>
37     # Get list of default colors for style
38     prop_cycle = plt.rcParams["axes.prop_cycle"]

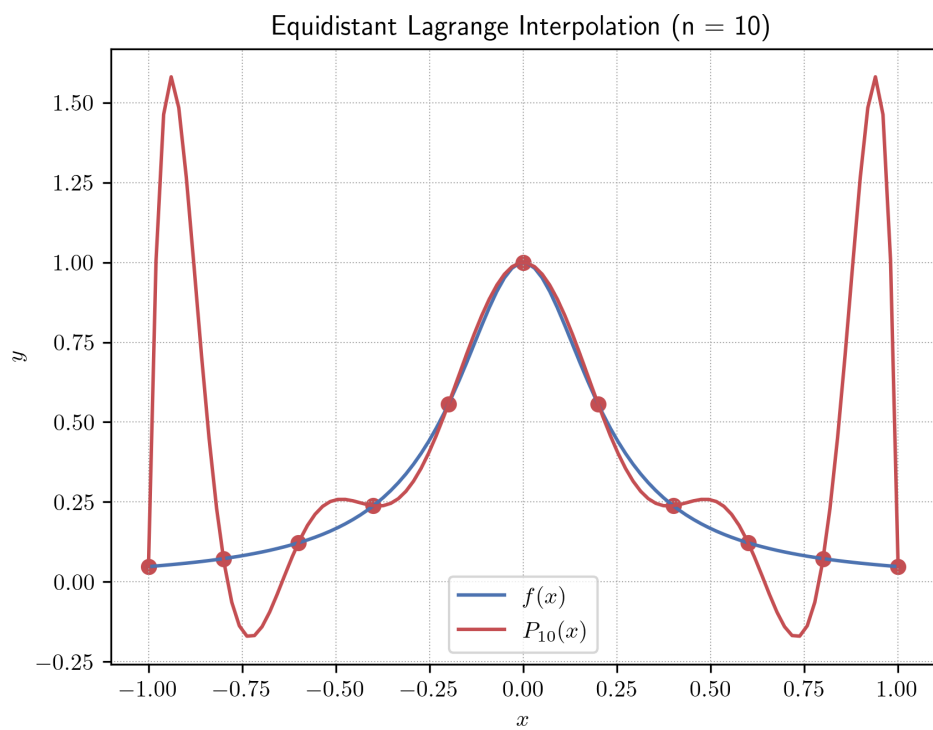
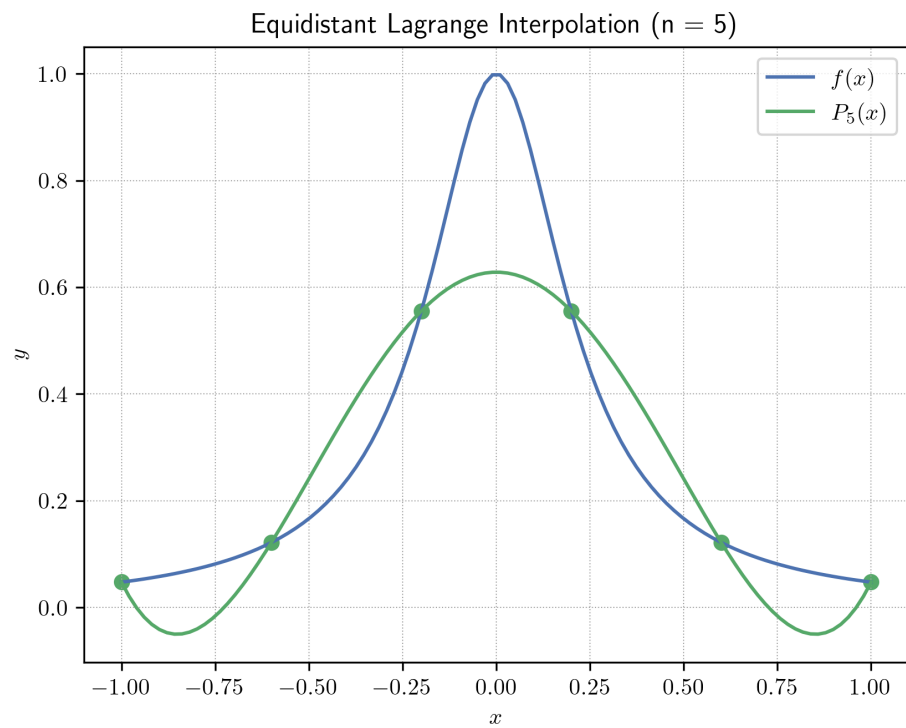
```

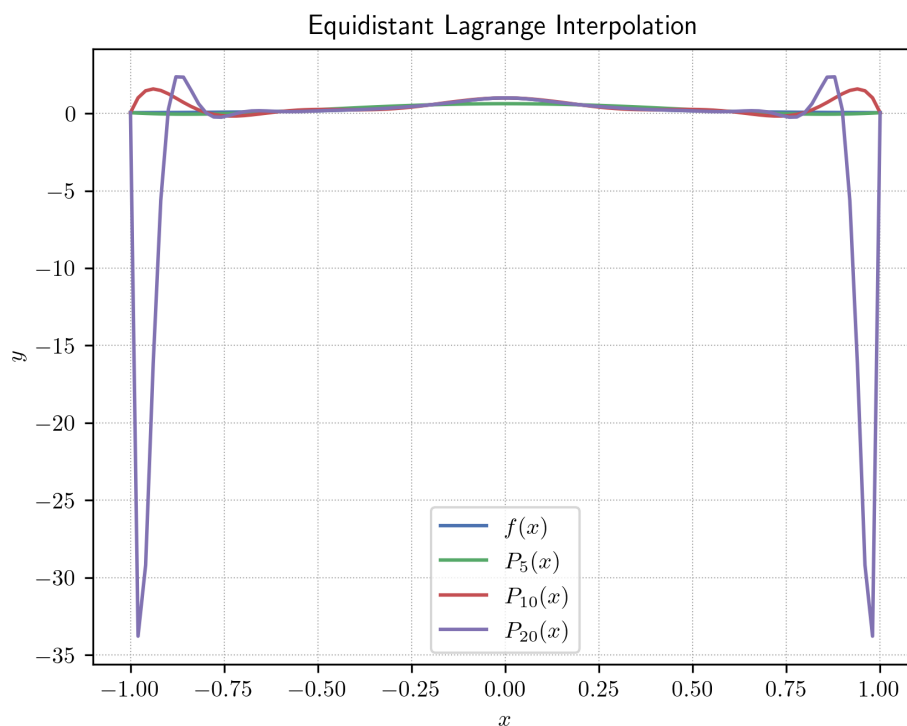
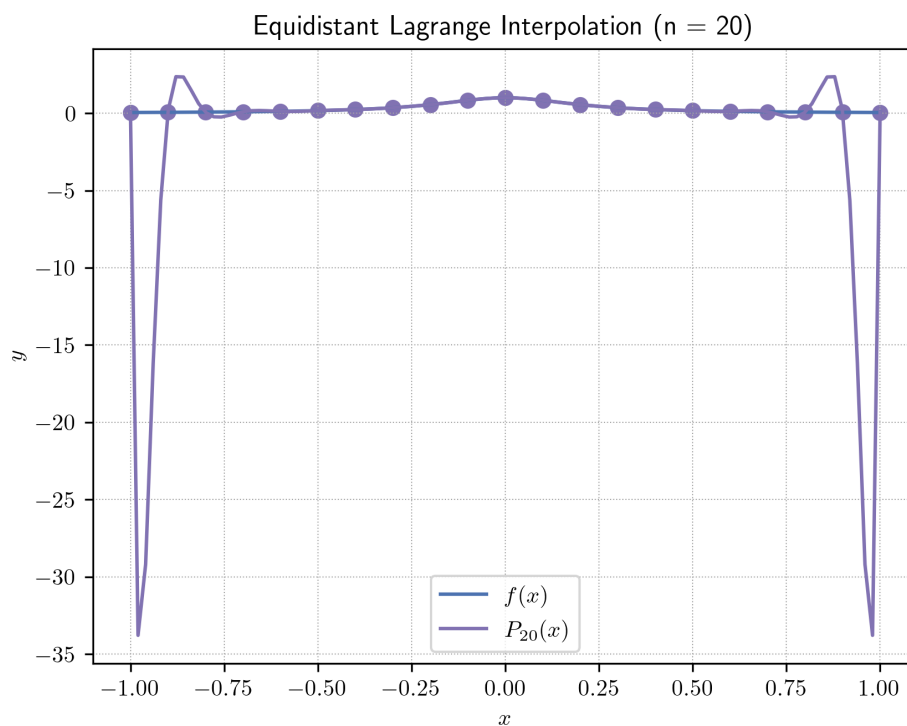
```

38     default_colors = prop_cycle.by_key()["color"]
39
40     title = "Equidistant Lagrange Interpolation"
41     x_coords = np.linspace(-1, 1, 100)
42     y_coords = f(x_coords)
43     n = [5, 10, 20]
44
45     for i in range(len(n) + 1):
46         fig, ax = plt.subplots()
47         ax.plot(x_coords, y_coords, label=r"$f(x)$")
48
49         # Plot f(x), P(x), and nodes
50         if i <= len(n) - 1:
51             x_nodes = np.linspace(-1, 1, n[i] + 1)
52             y_nodes = calculate_lagrange_output(x_nodes)
53             y_poly = calculate_lagrange_output(x_nodes, x_coords)
54             ax.plot(
55                 x_coords,
56                 y_poly,
57                 label=rf"$P_{\{{{n[i]}}\}}(x)$",
58                 color=default_colors[i + 1],
59             )
60             ax.scatter(x_nodes, y_nodes, color=default_colors[i + 1])
61             ax.set_title(title + f" (n = {n[i]})")
62             path = f"./plots_2/q2_2/p{n[i]}.png"
63
64         # Plot f(x) with all P(x)
65         else:
66             for j in range(len(n)):
67                 x_nodes = np.linspace(-1, 1, n[j] + 1)
68                 y_nodes = calculate_lagrange_output(x_nodes)
69                 y_poly = calculate_lagrange_output(x_nodes, x_coords)
70                 ax.plot(
71                     x_coords,
72                     y_poly,
73                     label=rf"$P_{\{{{n[j]}}\}}(x)$",
74                     color=default_colors[j + 1],
75                 )
76             ax.set_title(title)
77             path = "./plots_2/q2_2/all.png"
78
79         # Configure axis and save figure
80         ax.set_xlabel(r"$x$")
81         ax.set_ylabel(r"$y$")
82         ax.legend()
83         fig.savefig(path, dpi=300)
84         plt.show()
85         ax.cla()

```

The following plots show $f(x)$ and $P_n(x)$ for $n = 5, 10, 20$:





As n increases from 5 to 10 and 20, we see that the interpolation error decreases towards the center of the interval $[-1, 1]$. However, the error increases significantly near the edges of the interval, demonstrating Runge's phenomenon.

Question 2.3

Repeat the interpolation using Chebyshev nodes, and compare the results with the equidistant-node case.

Solution. In addition to the functions defined in the previous parts, the following function was used to generate the Chebyshev nodes:

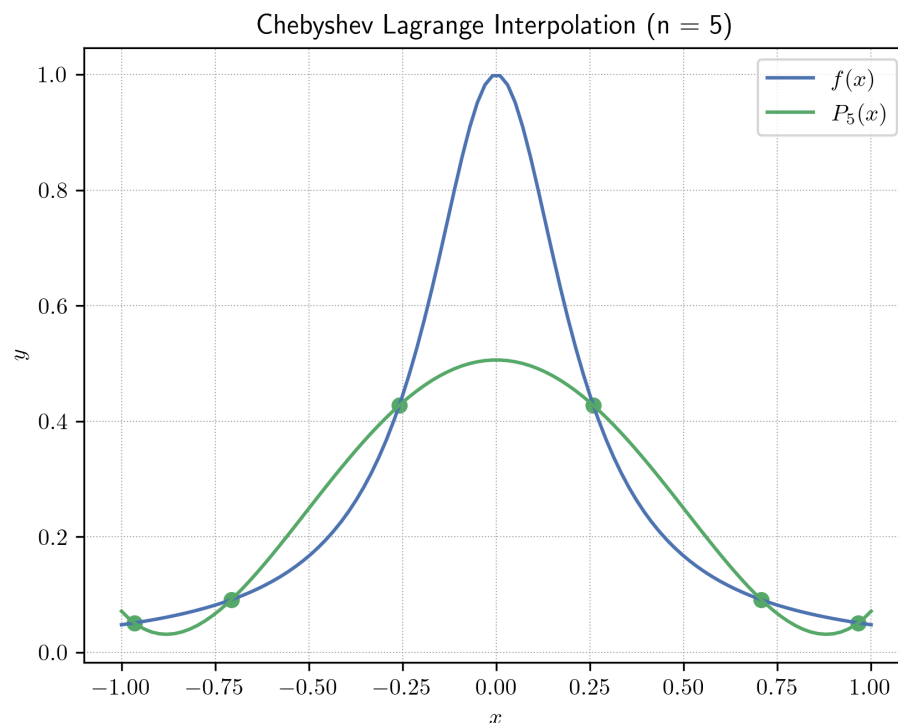
Listing 3: 2.3 Python

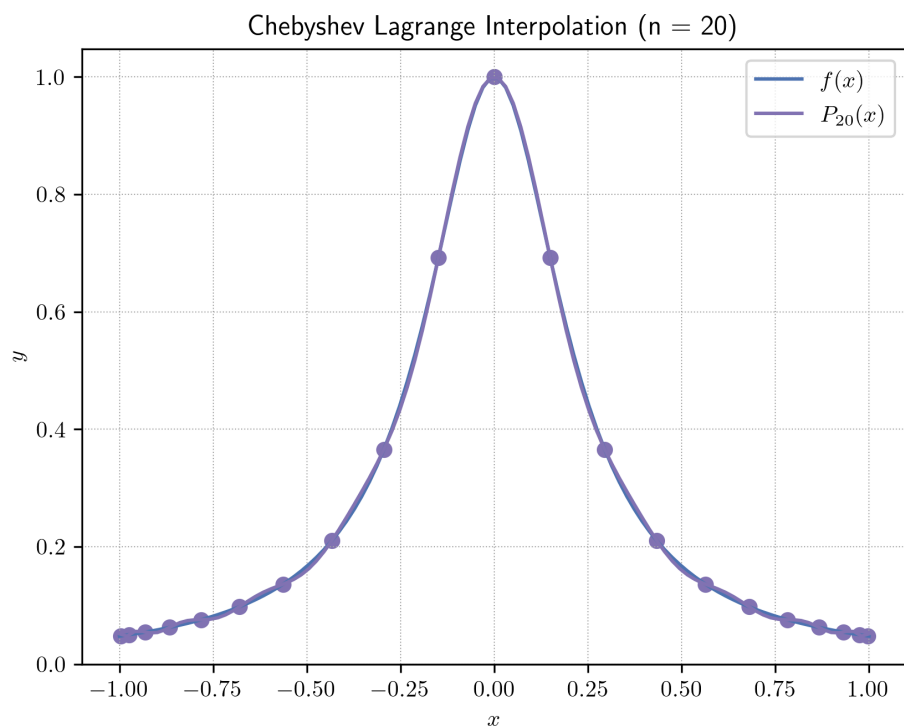
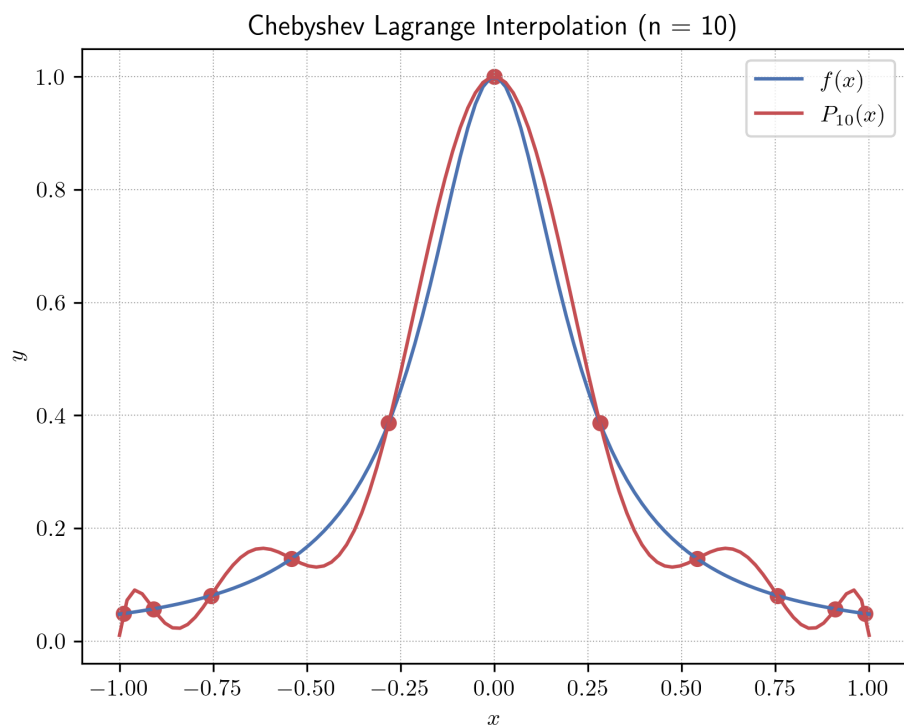
```

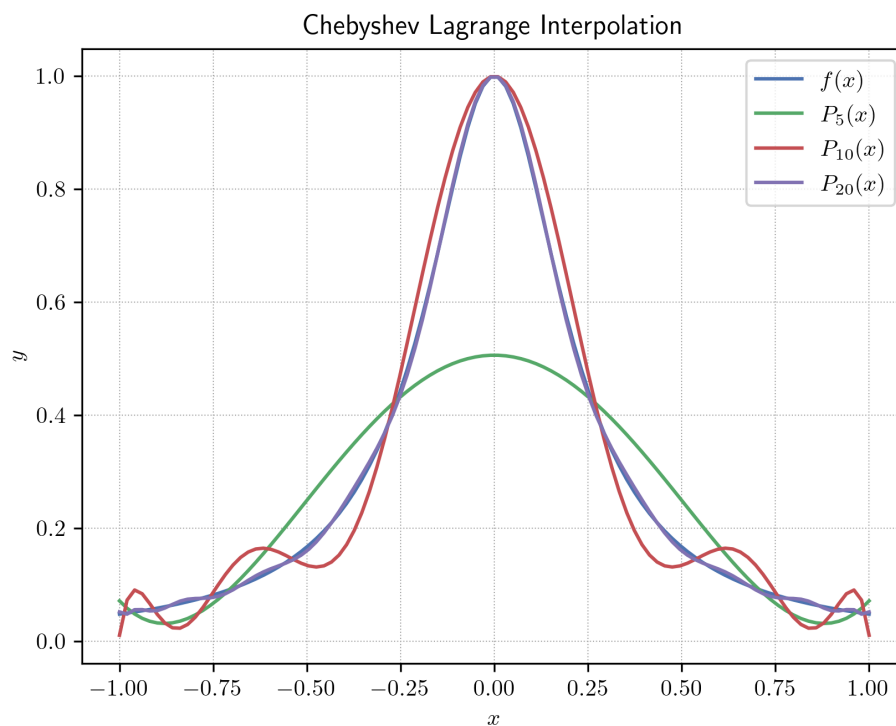
1 import math
2 import numpy as np
3
4 def chebyshev_nodes(n):
5     nodes = np.array(
6         [(math.cos((2 * k - 1) * math.pi / (2 * n))) for k in range(1, n + 1)]
7     )
8
9     return nodes
10
11 # The main method is almost identical to the previous part.
12 # The only difference is that chebyshev_nodes(n[i] + 1)
13 # is used instead of np.linspace(-1, 1, n[i] + 1) to generate the nodes.

```

The following plots show $f(x)$ and $P_n(x)$ for Chebyshev nodes with $n = 5, 10, 20$:







Comparing these plots to those with equidistant nodes, we see that using Chebyshev nodes significantly reduces the interpolation error across the entire interval $[-1, 1]$. The oscillations near the edges of the interval are much less pronounced, demonstrating that Chebyshev nodes help mitigate Runge's phenomenon and provide a more accurate approximation of $f(x)$.

3 - Chebyshev Polynomials and Their Roots

The Chebyshev polynomials of the first kind is defined by:

$$T_n(x) = \cos(n \arccos x), \quad x \in [-1, 1]$$

Question 3.1

Prove that the numbers

$$x_k = \cos\left(\frac{(2k-1)\pi}{2n}\right), \quad k = 1, 2, \dots, n$$

are the roots of $T_n(x)$.

Proof. We want to find the values of x such that $T_n(x) = 0$. By the definition of Chebyshev polynomials, we have:

$$T_n(x) = \cos(n \arccos x)$$

Setting $T_n(x) = 0$, we get:

$$T_n(x) = \cos(n \arccos x) = 0$$

The cosine function is zero at odd multiples of $\frac{\pi}{2}$. So $T_n(x)$ can only equal zero when $n \arccos x$ is an odd multiple of $\frac{\pi}{2}$. Therefore, we can write:

$$n \arccos x = \frac{(2k-1)\pi}{2}, \quad k \in \mathbb{Z}$$

Note that for $n = 0$, $T_0(x) = \cos(0 \cdot \arccos(x)) = 1$ which has no roots. So consider $n \geq 1$. Dividing both sides by n , we have:

$$\arccos x = \frac{(2k-1)\pi}{2n}, \quad k \in \mathbb{Z}$$

Taking the cosine of both sides, we get:

$$x = \cos\left(\frac{(2k-1)\pi}{2n}\right), \quad k \in \mathbb{Z}$$

Therefore, the set of x defined above are the roots of $T_n(x)$ for $n \geq 1$. Note that if we restrict k to the integers $1, 2, \dots, n$, we exactly get that:

$$x_k = \cos\left(\frac{(2k-1)\pi}{2n}\right), \quad k = 1, 2, \dots, n$$

are the roots of $T_n(x)$. □

Question 3.2

Show that these roots are distinct and lie in the interval $(-1, 1)$.

Proof. To show that these roots are distinct, we need to show that for $k_1 \neq k_2$, we have $x_{k_1} \neq x_{k_2}$. Assume $k_1, k_2 \in \{1, 2, \dots, n\}$ such that $k_1 \neq k_2$ and $k_1 < k_2$. Then, assuming for fixed $n \geq 1$:

$$x_{k_1} = \cos\left(\frac{2k_1 - 1}{2n}\pi\right), \quad x_{k_2} = \cos\left(\frac{2k_2 - 1}{2n}\pi\right)$$

Note that $\frac{2k_1 - 1}{2n}\pi$ and $\frac{2k_2 - 1}{2n}\pi$ are distinct angles in the interval $(0, \pi)$. Since the cosine function is strictly decreasing on the interval $[0, \pi]$, and since $\frac{2k_1 - 1}{2n}\pi < \frac{2k_2 - 1}{2n}\pi$ are distinct, we have:

$$x_{k_1} = \cos\left(\frac{2k_1 - 1}{2n}\pi\right) > \cos\left(\frac{2k_2 - 1}{2n}\pi\right) = x_{k_2}$$

By the law of trichotomy, $x_{k_1} \neq x_{k_2}$. Therefore, the roots x_k are distinct for $k = 1, 2, \dots, n$.

Next, we need to show that these roots lie in the interval $(-1, 1)$. Note that from above, we have shown that x_k is a strictly decreasing sequence. Therefore, we only need to show that the largest root x_1 is less than 1 and the smallest root x_n is greater than -1 .

First, let us show that $x_1 < 1$:

$$x_1 = \cos\left(\frac{2(1) - 1}{2n}\pi\right) = \cos\left(\frac{\pi}{2n}\right)$$

Note that $\frac{\pi}{2n} \in (0, \frac{\pi}{2}]$. In this interval, the range of the cosine function is $[0, 1)$. Therefore, we have:

$$x_1 = \cos\left(\frac{\pi}{2n}\right) < 1 \tag{1}$$

Next, let us show that $x_n > -1$:

$$x_n = \cos\left(\frac{2n - 1}{2n}\pi\right) = \cos\left(\pi - \frac{\pi}{2n}\right)$$

Note that $\pi - \frac{\pi}{2n} \in [\frac{\pi}{2}, \pi)$. In this interval, the range of the cosine function is $(-1, 0]$. Therefore, we have:

$$x_n = \cos\left(\pi - \frac{\pi}{2n}\right) > -1 \tag{2}$$

Combining (1) and (2), we have shown that:

$$-1 < x_n < x_k < x_1 < 1$$

Therefore, x_k are distinct roots that lie in the interval $(-1, 1)$ for $k = 1, 2, \dots, n$. □

Question 3.3

Write your own code to generate $T_n(x)$ using the recursion formula.

Solution. The following code was used to generate the Chebyshev polynomials::

Listing 4: 3.3 Python

```

1  import sympy as smp
2
3  def chebyshev_poly(n):
4      # Base Cases
5      if n == 0:
6          return 1
7      elif n == 1:
8          return smp.symbols("x")
9      # Recursive Case
10     else:
11         return 2 * smp.symbols('x') * chebyshev_poly(n - 1) - chebyshev_poly(n
- 2)
12
13 def generate_chebyshev(n):
14     x = smp.symbols('x')
15     poly = smp.expand(chebyshev_poly(n))
16     function = f"T_{{{n}}}({x})"
17     equation = function + " = " + smp.latex(poly)
18     return equation, poly
19
20 # Main Method
21 if __name__ == "__main__":
22     # Used for plotting in next part
23     polys = []
24
25     # Generate and save equations to a text file
26     with open("./plots_2/q3_3/chebyshev_polynomials.txt", "w") as file:
27         for n in range(6):
28             equation, poly = generate_chebyshev(n)
29             polys.append(poly)
30             file.write(equation + "\n")

```

The Chebyshev polynomials $T_n(x)$ for $n = 0, 1, \dots, 5$ are given by:

$$T_0(x) = 1$$

$$T_1(x) = x$$

$$T_2(x) = 2x^2 - 1$$

$$T_3(x) = 4x^3 - 3x$$

$$T_4(x) = 8x^4 - 8x^2 + 1$$

$$T_5(x) = 16x^5 - 20x^3 + 5x$$

Question 3.4

Plot the first six polynomials $T_0(x), T_1(x), \dots, T_5(x)$ on the interval $[-1, 1]$.

Solution. The following code was used to plot the Chebyshev polynomials:

Listing 5: 3.4 Python

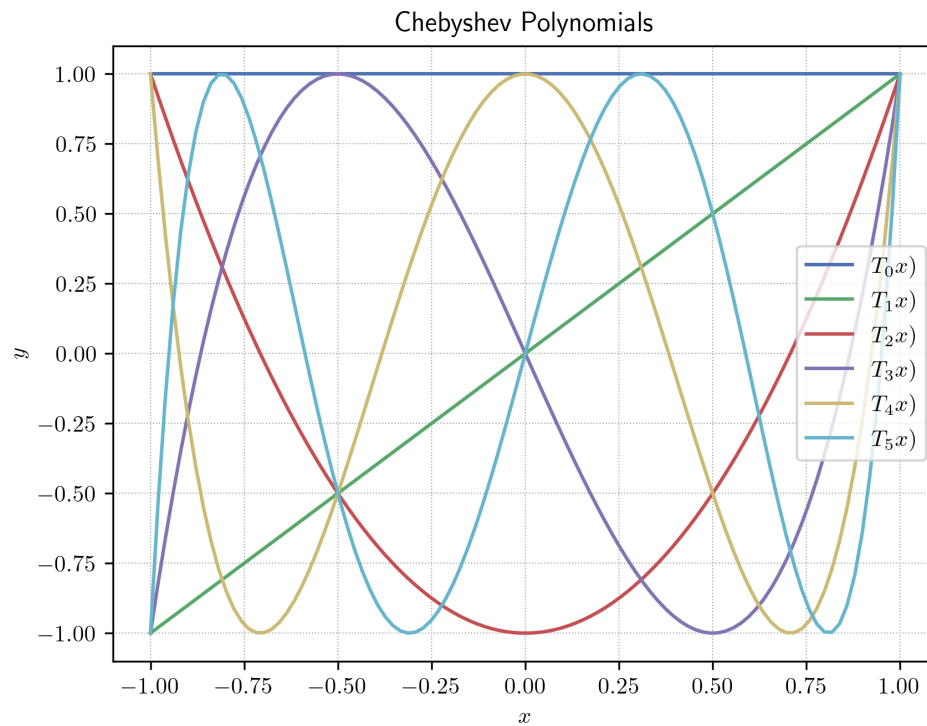
```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import sympy as smp
4
5 if __name__ == "__main__":
6     # <some matplotlib styling>
7     # polys list was generated in previous part
8     x_coords = np.linspace(-1, 1, 100)
9     x = smp.symbols("x")
10
11     fig, ax = plt.subplots()
12     # Plot each polynomial
13     for i in range(6):
14         # Evaluate polynomial at each x coordinate
15         if i != 0:
16             y_coords = np.array(
17                 [polys[i].evalf(subs={x: x_coords[j]}) for j in range(len(
18 x_coords))]
19             )
20             # Case for T_0(x) = 1
21         else:
22             y_coords = np.array([1 for _ in range(len(x_coords))])
23         ax.plot(x_coords, y_coords, label=r"$T_{\{{{i}\}}x}$")
24
25     # Configure axis and save figure
26     ax.set_title("Chebyshev Polynomials")
27     ax.set_xlabel(r"$x$")

```

```
27 ax.set_ylabel(r"$y$")
28 ax.legend()
29 fig.savefig("./plots_2/q3_4/chebyshev_polys.png", dpi=300)
30 plt.show()
```

The following plot was generated of Chebyshev polynomials $T_n(x)$ for $n = 0, 1, \dots, 5$:



4 - Lagrange Interpolation for Nonsmooth Function

Let $f(x) = |x|$ on the interval $[-1, 1]$.

Question 4.1

Construct the interpolation polynomial $P_n(x)$ for equidistant nodes when n is even.

Solution. The interpolation polynomial $P_n(x)$ can be constructed using the Lagrange interpolation formula:

$$P_n(x) = \sum_{i=0}^n f(x_i) \prod_{\substack{0 \leq j \leq n \\ j \neq i}} \frac{x - x_j}{x_i - x_j}$$

where x_i are the equidistant nodes in the interval $[-1, 1]$. For even n , the nodes are given by:

$$x_i = -1 + \frac{2i}{n}, \quad i = 0, 1, \dots, n$$

Substituting $f(x) = |x|$ into the Lagrange formula gives that:

$$P_n(x) = \sum_{i=0}^n |x_i| \prod_{\substack{0 \leq j \leq n \\ j \neq i}} \frac{x - x_j}{x_i - x_j}$$

Question 4.2

Show that $P_n(x)$ is an even polynomial.

Proof. To show that $P_n(x)$ is an even polynomial, we need to show that $P_n(-x) = P_n(x)$ for all x in the interval $[-1, 1]$.

From the previous part, we have that:

$$P_n(x) = \sum_{i=0}^n |x_i| L_i(x)$$

where

$$L_i(x) = \prod_{\substack{0 \leq j \leq n \\ j \neq i}} \frac{x - x_j}{x_i - x_j}$$

is the Lagrange basis polynomial. If we plug in $-x$ into $P_n(x)$, we have:

$$P_n(-x) = \sum_{i=0}^n |x_i| L_i(-x)$$

Let us first focus on $L_i(-x)$. Since n is even, we have that the nodes x_i are symmetric about 0. So we have that for each node x_i , there exists a corresponding node $x_{n-i} = -x_i$. Starting with $L_i(-x)$, we have:

$$L_i(-x) = \prod_{\substack{0 \leq j \leq n \\ j \neq i}} \frac{-x - x_j}{x_i - x_j}$$

Let focus on each factor in the product and let $x_i = -x_{n-i}$, $x_j = -x_{n-j}$. Then, we have:

$$\frac{-x - x_j}{x_i - x_j} = \frac{-x + x_{n-j}}{-x_{n-i} + x_{n-j}} = \frac{-(x - x_{n-j})}{-(x_{n-i} - x_{n-j})} = \frac{x - x_{n-j}}{x_{n-i} - x_{n-j}}$$

Therefore, we can rewrite $L_i(-x)$ as:

$$L_i(-x) = \prod_{\substack{0 \leq j \leq n \\ j \neq i}} \frac{x - x_{n-j}}{x_{n-i} - x_{n-j}}$$

Note that as j goes from 0 to n , $j \neq i$, $n - j$ goes from n to 0 and $n - j \neq n - i$, and $n - i$ goes from n to 0. This is just the original product in backwards order. Letting $m = n - j$, we can rewrite the product as:

$$L_i(-x) = \prod_{\substack{0 \leq m \leq n \\ m \neq n-i}} \frac{x - x_m}{x_{n-i} - x_m} = L_{n-i}(x) \quad (1)$$

Now reconsidering $P_n(-x)$, we have:

$$P_n(-x) = \sum_{i=0}^n |x_i| L_i(-x)$$

Using (1), we can rewrite this as:

$$P_n(-x) = \sum_{i=0}^n |x_i| L_{n-i}(x)$$

By the symmetry of the nodes, we have that $|x_i| = |-x_{n-i}| = |x_{n-i}|$. So:

$$P_n(-x) = \sum_{i=0}^n |x_{n-i}| L_{n-i}(x)$$

Letting $m = n - i$, we can rewrite the sum as:

$$P_n(-x) = \sum_{m=0}^n |x_m| L_m(x) = P_n(x)$$

Therefore have shown that $P_n(-x) = P_n(x)$ for all x in the interval $[-1, 1]$. So $P_n(x)$ is an even polynomial. \square

Question 4.3

Investigate analytically (for small n) how well $P_n(x)$ approximates $f(x)$.

Solution. The remainder term for Lagrange interpolation is given by:

$$R_n(x) = f(x) - P_n(x) = |x| - P_n(x), \quad x \in [-1, 1]$$

Since $f(x)$ and $P_n(x)$ are both even functions, $R_n(x)$ is also an even function. Therefore, we only need to analyze $R_n(x)$ on the interval $[0, 1]$. Let us analyze $\max_{x \in [0, 1]} |R_n(x)|$ for small even $n = 2k$. Note that we may rewrite $R_n(x)$ as:

$$R_n(x) = x - P_n(x), \quad x \in [0, 1]$$

Since $P_n(x)$ is even, all odd constants ($a_{2k-1}, a_{2k-3}, \dots, a_1$) must vanish. Also, given that $n = 2k$ is even, we have that 0 is always a node. Therefore, we have that $P_n(0) = a_{2k}x^{2k} + a_{2k-2}x^{2k-2} + \dots + a_2x^2 + a_0 = f(0) = 0$. So $a_0 = 0$.

Case: $n = 2$

Our nodes are $\{-1, 0, 1\}$.

Note $P_2(x)$ must be of the form:

$$P_2(x) = a_2x^2$$

So we have:

$$P_2(1) = a_2 = 1$$

Therefore, we have:

$$P_2(x) = x^2$$

So the remainder term is:

$$R_2(x) = x - x^2, \quad x \in [0, 1]$$

To find the maximum error, we can take the derivative of $R_2(x)$ and set it to 0:

$$R_2'(x) = 1 - 2x = 0$$

$$\implies x = \frac{1}{2}$$

Evaluating $|R_2(x)|$ at the critical point (note: maximum cannot occur at endpoint by construction):

$$|R_2\left(\frac{1}{2}\right)| = \frac{1}{2} - \frac{1}{4} = \frac{1}{4}$$

So the maximum error occurs at $x = \frac{1}{2}$ and is:

$$\max_{x \in [0, 1]} |R_2(x)| = \frac{1}{4} = 0.25$$

Case: $n = 4$

Our nodes are $\{-1, -\frac{1}{2}, 0, \frac{1}{2}, 1\}$.

Note $P_4(x)$ must be of the form:

$$P_4(x) = a_4x^4 + a_2x^2$$

So we have the following system of equations:

$$\begin{aligned} P_4(1) &= a_4 + a_2 = 1 \\ P_4\left(\frac{1}{2}\right) &= \frac{a_4}{16} + \frac{a_2}{4} = \frac{1}{2} \end{aligned}$$

Rewriting the second equation, we have:

$$a_4 + 4a_2 = 8$$

Solving this system of equations, we get:

$$a_4 = -\frac{4}{3}, \quad a_2 = \frac{7}{3}$$

So therefore:

$$P_4(x) = -\frac{4}{3}x^4 + \frac{7}{3}x^2$$

As a result, our remainder term is:

$$R_4(x) = x - \left(-\frac{4}{3}x^4 + \frac{7}{3}x^2\right) = x + \frac{4}{3}x^4 - \frac{7}{3}x^2, \quad x \in [0, 1]$$

Again, to find the maximum error, we can take the derivative of $R_4(x)$ and set it to 0:

$$R'_4(x) = 1 + \frac{16}{3}x^3 - \frac{14}{3}x = 0$$

We may use the cubic formula to find the roots of this equation. Using WolframAlpha, we find that the roots in the interval $[0, 1]$ are approximately $x \approx 0.22779, 0.80048$. By the Runge phenomenon, we expect the maximum error to occur in the last sub-interval $[\frac{1}{2}, 1]$. Evaluating $|R_4(x)|$ at the critical points of the interval, we have:

$$|R_4(0.22779)| \approx 0.1103071958$$

$$|R_4(0.80048)| \approx 0.1472006375$$

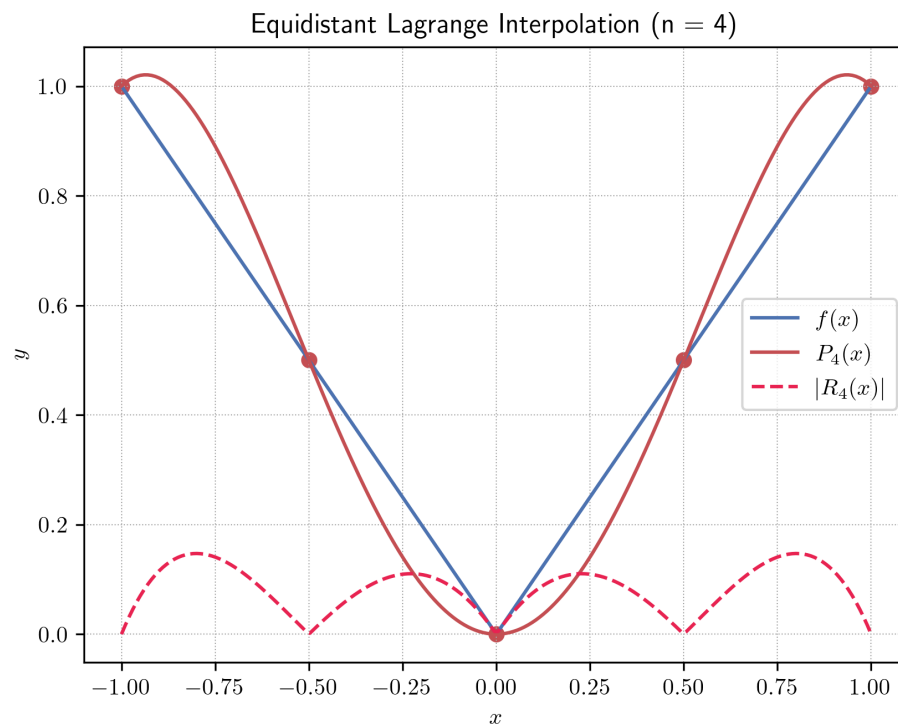
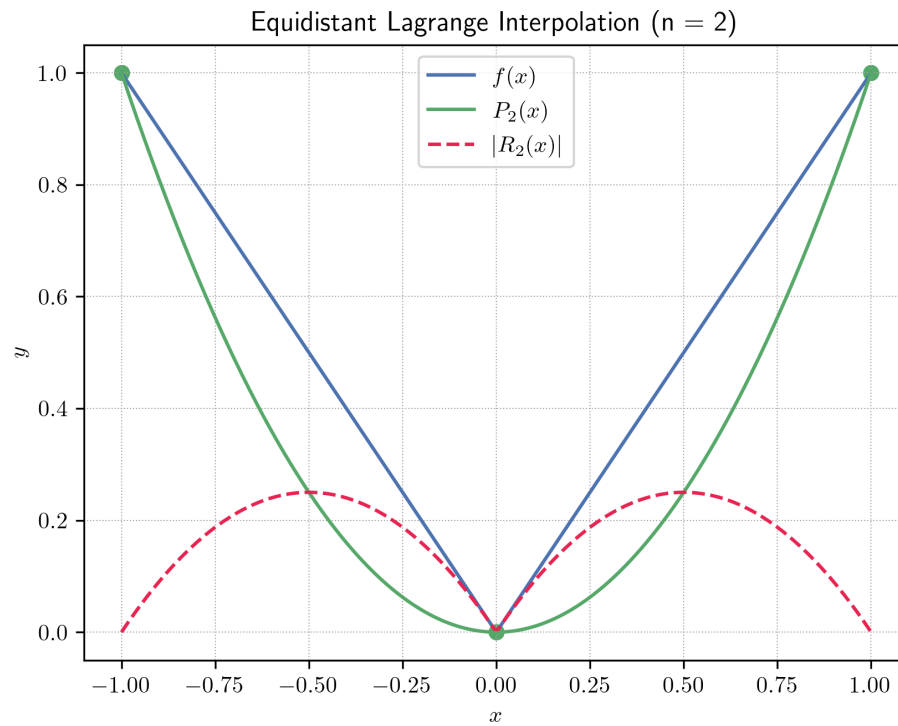
So the maximum error occurs at $x \approx 0.80048$ and is approximately:

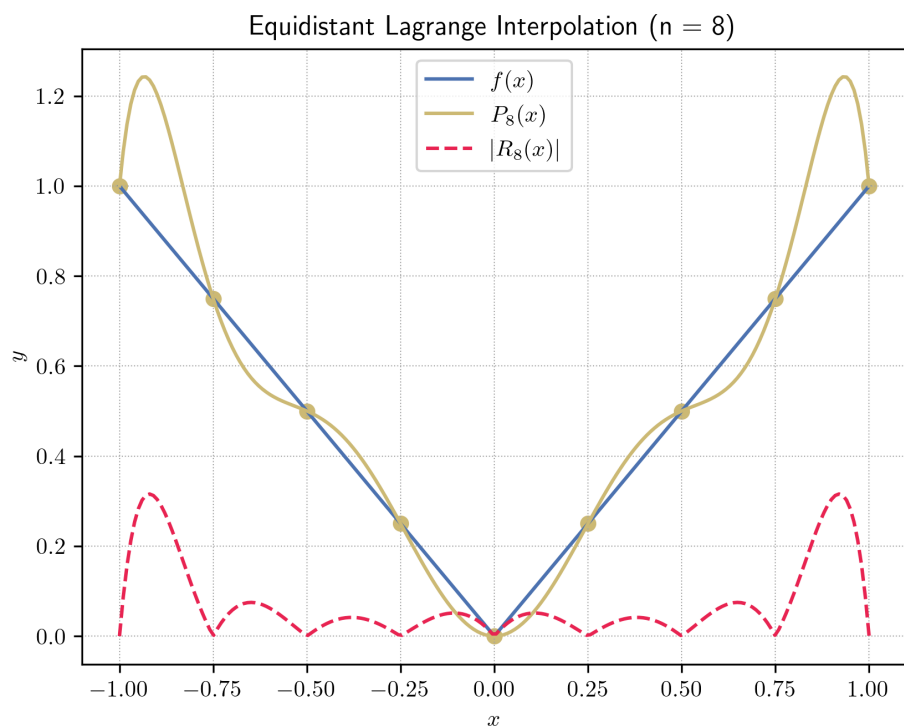
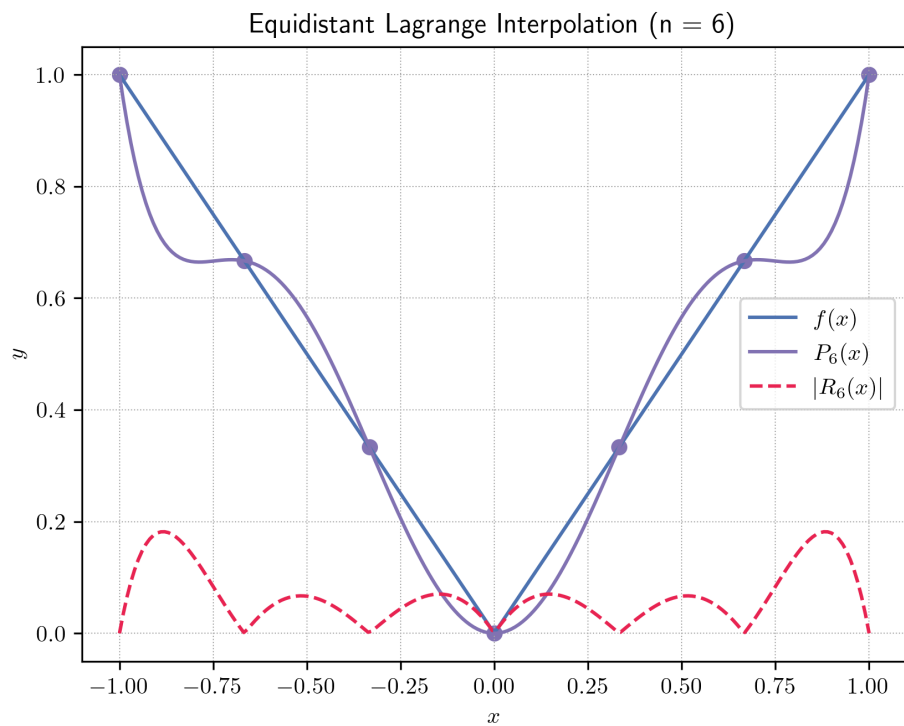
$$\max_{x \in [0, 1]} |R_4(x)| \approx 0.1472006375$$

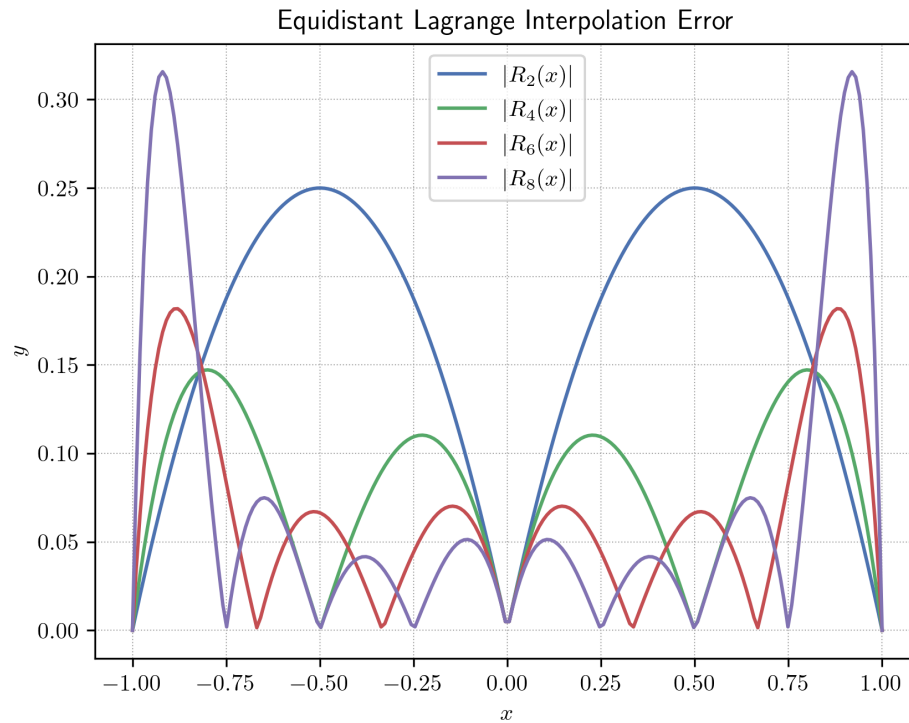
For higher even n , a similar approach can be used to find $P_n(x)$ and analyze the error. The roots of $R'_n(x)$ can be found using numerical methods such as Newton's method or the bisection method since they don't have closed-form solutions.

To avoid the heavy computation, we can observe $|R_n(x)|$ graphically for different n . A modified version of the code from question 2.2 was used to plot $f(x)$, $P_n(x)$, and $|R_n(x)|$ for even $n = 2, 4, 6, 8$ (see Appendix).

The following plots show $f(x)$ and $P_n(x)$ for even $n = 2, 4, 6, 8$:







We note that initially, $\max_{x \in [0,1]} |R_n(x)|$ decreases from $n = 2$ (0.25) to $n = 4$ (0.1472006375). However, starting from $n = 6$, the maximum error (roughly 0.18) starts to increase again occurring near the endpoints. This trend continues, eventually surpassing the maximal error for $n = 2$ (0.25) at $n = 8$ (roughly 0.315). Since our nodes are equally spaced, this is an observation of the Runge phenomenon. Centrally, the error decreases as n increases, but near the endpoints, the error increases significantly.

Question 4.4

Discuss why convergence may be slower for nonsmooth functions.

Solution. Nonsmooth functions, such as $f(x) = |x|$, have points where they are not differentiable. In the case of $f(x) = |x|$, f is not differentiable at $x = 0$. This lack of smoothness can lead to larger interpolation errors, especially near the nonsmooth points. Since our n is even, we always have a node at $x = 0$. However, if n were odd, we would not have a node at 0, and the interpolation error would be nonzero at this point. Since Lagrange interpolation uses smooth basis polynomials to approximate f , it may not be able to accurately capture the behavior of the nonsmooth function near these points, leading to slower convergence.

This effect is more apparent when interpolating functions that are near vertical or have jump discontinuities. When using Fourier series to approximate such functions (like a step function), we can observe the Gibbs phenomenon, where oscillations occur near the discontinuities. These oscillations do not diminish as more terms are added to the series, leading to a persistent error near the discontinuity. Similarly, for polynomial interpolation of nonsmooth functions, we may observe oscillatory behavior near the nonsmooth points, leading to slower convergence.

Appendix

This appendix contains the complete code used for Assignment 2.

Listing 6: assignment_2.py

```

1  import math
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import sympy as smp
5
6
7  def f(x):
8      return 1 / (1 + 20 * x**2)
9
10
11 def lagrange_coefficients(nodes, function=f):
12     x = nodes
13     num_nodes = len(nodes)
14     # Add zeroth divided differences
15     dd_table = np.array([[function(xi) for xi in nodes]])
16
17     # Calculate divided difference table
18     for i in range(1, num_nodes):
19         ith_dd = np.zeros(num_nodes)
20
21         for j in range(num_nodes - i):
22             # Calculate ith divided differences
23             ith_dd[j] = (dd_table[i - 1, j + 1] - dd_table[i - 1, j]) / (
24                 x[j + i] - x[j]
25             )
26
27         # Append the ith divided difference row to the table
28         dd_table = np.vstack([dd_table, ith_dd])
29
30     # Extract coefficients (first column of the table)
31     a = np.array([dd_table[i, 0] for i in range(dd_table.shape[0])])
32     return a
33
34
35 def calculate_lagrange(nodes, a, x):
36     # Start with constant term
37     y = a[0]
38     # Build (x - xi) terms
39     w = [(x - xi) for xi in nodes]
40     # Temporary variable to hold (x - xi) product
41     b = 1
42

```

```

43     for i in range(1, len(a)):
44         for j in range(i):
45             # Multiply (x - xi) terms
46             b *= w[j]
47             # Multiply (x - xi) product with current coefficient
48             y += a[i] * b
49             b = 1
50
51     return y
52
53
54 def generate_lagrange(nodes, degree):
55     # Get coefficients
56     a = lagrange_coefficients(nodes)
57     # Start with function and constant term
58     equation = f"P_{{{degree}}}(x) = {a[0]}"
59     w = []
60     # Build (x - xi) terms
61     for xi in nodes:
62         if abs(xi) <= 1e-14:
63             w.append('(x)')
64         elif xi < 0:
65             w.append(f"(x + {abs(xi)})")
66         else:
67             w.append(f"(x - {xi})")
68     b = ""
69
70     # Build polynomial string
71     for i in range(1, len(a)):
72         for j in range(i):
73             # Multiply (x - xi) terms
74             b += w[j]
75             # Multiply (x - xi) product with current coefficient and add term
76             if a[i] >= 0:
77                 equation += f" + {a[i]}{b}"
78             else:
79                 equation += f" - {abs(a[i])}{b}"
80             b = ""
81
82     return equation
83
84
85 def calculate_lagrange_output(nodes, x_coords=[], function=f):
86     # Get coefficients
87     a = lagrange_coefficients(nodes, function)
88     # If no x_coords provided, use nodes as x_coords
89     if len(x_coords) == 0:

```

```

90     x_coords = nodes
91     # Calculate y coordinates for each x coordinate
92     y_coordinates = np.array([calculate_lagrange(nodes, a, x) for x in x_coords
93 ])
94
95     return y_coordinates
96
97 def chebyshev_nodes(n):
98     nodes = np.array(
99         [(math.cos((2 * k - 1) * math.pi / (2 * n))) for k in range(1, n + 1)]
100     )
101
102     return nodes
103
104
105 def chebyshev_poly(n):
106     # Base Cases
107     if n == 0:
108         return 1
109     elif n == 1:
110         return smp.symbols("x")
111     # Recursive Case
112     else:
113         return 2 * smp.symbols("x") * chebyshev_poly(n - 1) - chebyshev_poly(n
114 - 2)
115
116
117 def generate_chebyshev(n):
118     x = smp.symbols("x")
119     poly = smp.expand(chebyshev_poly(n))
120     function = f"T_{{{n}}}({x})"
121     equation = function + " = " + smp.latex(poly)
122     return equation, poly
123
124 # Main Method
125 if __name__ == "__main__":
126     # Enable Latex and Styling
127     plt.rcParams["text.usetex"] = True
128     plt.rcParams["axes.grid"] = True
129     plt.rc("grid", color="#a6a6a6", linestyle="dotted", linewidth=0.5)
130     plt.style.use("seaborn-v0_8-deep")
131     # Get list of default colors for style
132     prop_cycle = plt.rcParams["axes.prop_cycle"]
133     default_colors = prop_cycle.by_key()["color"]
134

```

```

135 # Question 2
136 # -----
137 # Question 2.1 (Find P5(x) Equation)
138 # -----
139 # Generate and save equation to a text file
140 nodes = np.linspace(-1, 1, 6)
141 equation = generate_lagrange(nodes, 5)
142 with open("./plots_2/q2_1/p5.txt", "w") as file:
143     file.write(equation)
144
145 # -----
146 # Question 2.2 (Equidistant Nodes)
147 # -----
148 title = "Equidistant Lagrange Interpolation"
149 x_coords = np.linspace(-1, 1, 100)
150 y_coords = f(x_coords)
151 n = [5, 10, 20]
152
153 for i in range(len(n) + 1):
154     fig, ax = plt.subplots()
155     ax.plot(x_coords, y_coords, label=r"$f(x)$")
156
157     # Plot f(x), P(x), and nodes
158     if i <= len(n) - 1:
159         x_nodes = np.linspace(-1, 1, n[i] + 1)
160         y_nodes = calculate_lagrange_output(x_nodes)
161         y_poly = calculate_lagrange_output(x_nodes, x_coords)
162         ax.plot(
163             x_coords,
164             y_poly,
165             label=rf"$P_{{{n[i]}}}(x)$",
166             color=default_colors[i + 1],
167         )
168         ax.scatter(x_nodes, y_nodes, color=default_colors[i + 1])
169         ax.set_title(title + f" (n = {n[i]})")
170         path = f"./plots_2/q2_2/p{n[i]}.png"
171
172     # Plot f(x) with all P(x)
173     else:
174         for j in range(len(n)):
175             x_nodes = np.linspace(-1, 1, n[j] + 1)
176             y_nodes = calculate_lagrange_output(x_nodes)
177             y_poly = calculate_lagrange_output(x_nodes, x_coords)
178             ax.plot(
179                 x_coords,
180                 y_poly,
181                 label=rf"$P_{{{n[j]}}}(x)$",

```

```

182         color=default_colors[j + 1],
183     )
184     ax.set_title(title)
185     path = "./plots_2/q2_2/all.png"
186
187     # Configure axis and save figure
188     ax.set_xlabel(r"$x$")
189     ax.set_ylabel(r"$y$")
190     ax.legend()
191     fig.savefig(path, dpi=300)
192     plt.show()
193     ax.cla()
194
195     # -----
196     # Question 2.3 (Chebyshev Nodes)
197     # -----
198     title = "Chebyshev Lagrange Interpolation"
199     x_coords = np.linspace(-1, 1, 100)
200     y_coords = f(x_coords)
201     n = [5, 10, 20]
202
203     for i in range(len(n) + 1):
204         fig, ax = plt.subplots()
205         ax.plot(x_coords, y_coords, label=r"$f(x)$")
206
207         # Plot f(x), P(x), and nodes
208         if i <= len(n) - 1:
209             x_nodes = chebyshev_nodes(n[i] + 1)
210             y_nodes = calculate_lagrange_output(x_nodes)
211             y_poly = calculate_lagrange_output(x_nodes, x_coords)
212             ax.plot(
213                 x_coords,
214                 y_poly,
215                 label=rf"$P_{{{n[i]}}}(x)$",
216                 color=default_colors[i + 1],
217             )
218             ax.scatter(x_nodes, y_nodes, color=default_colors[i + 1])
219             ax.set_title(title + f" (n = {n[i]})")
220             path = f"./plots_2/q2_3/chebyshev_p{n[i]}.png"
221
222         # Plot f(x) with all P(x)
223         else:
224             for j in range(len(n)):
225                 x_nodes = chebyshev_nodes(n[j] + 1)
226                 y_nodes = calculate_lagrange_output(x_nodes)
227                 y_poly = calculate_lagrange_output(x_nodes, x_coords)
228                 ax.plot(

```

```

229         x_coords,
230         y_poly,
231         label=rf"$P_{{{n[j]}}}(x)$",
232         color=default_colors[j + 1],
233     )
234     ax.set_title(title)
235     path = "./plots_2/q2_3/chebyshev_all.png"
236
237     # Configure axis and save figure
238     ax.set_xlabel(r"$x$")
239     ax.set_ylabel(r"$y$")
240     ax.legend()
241     fig.savefig(path, dpi=300)
242     plt.show()
243     ax.cla()
244
245     # -----
246     # Question 3
247     # -----
248     # Question 3.3 (Chebyshev Polynomials)
249     # -----
250     # Polynomials list used for plotting in 3.4
251     polys = []
252
253     # Generate and save equations to a text file
254     with open("./plots_2/q3_3/chebyshev_polynomials.txt", "w") as file:
255         for n in range(6):
256             equation, poly = generate_chebyshev(n)
257             polys.append(poly)
258             file.write(equation + "\n")
259
260     # -----
261     # Question 3.34 (Plot Chebyshev Polynomials)
262     # -----
263     # Plot Chebyshev Polynomials
264     x_coords = np.linspace(-1, 1, 100)
265     x = smp.symbols("x")
266
267     fig, ax = plt.subplots()
268     # Plot each polynomial
269     for i in range(6):
270         # Evaluate polynomial at each x coordinate
271         if i != 0:
272             y_coords = np.array(
273                 [polys[i].evalf(subs={x: x_coords[j]}) for j in range(len(
274                     x_coords))]
275             )

```

```

275     # Case for  $T_0(x) = 1$ 
276     else:
277         y_coords = np.array([1 for _ in range(len(x_coords))])
278         ax.plot(x_coords, y_coords, label=rf"$T_{i}(x)$")
279
280 # Configure axis and save figure
281 ax.set_title("Chebyshev Polynomials")
282 ax.set_xlabel(r"$x$")
283 ax.set_ylabel(r"$y$")
284 ax.legend()
285 fig.savefig("./plots_2/q3_4/chebyshev_polys.png", dpi=300)
286 plt.show()
287
288 # -----
289 # Question 4
290 # -----
291 # Question 4.3 (Lagrange for Nonsmooth Functions)
292 # -----
293 title = "Equidistant Lagrange Interpolation"
294 x_coords = np.linspace(-1, 1, 200)
295 y_coords = abs(x_coords)
296 n = [2, 4, 6, 8]
297
298 # Plot  $f(x)$ ,  $P(x)$ , nodes, and  $|R(x)|$  for each  $n$ 
299 for i in range(len(n)+1):
300     fig, ax = plt.subplots()
301
302     if i < len(n):
303         ax.plot(x_coords, y_coords, label=rf"$f(x)$")
304         x_nodes = np.linspace(-1, 1, n[i] + 1)
305         y_nodes = calculate_lagrange_output(x_nodes, x_nodes, abs)
306         y_poly = calculate_lagrange_output(x_nodes, x_coords, abs)
307         y_error = np.abs(y_coords - y_poly)
308         ax.plot(
309             x_coords,
310             y_poly,
311             label=rf"$P_{n[i]}(x)$",
312             color=default_colors[i + 1],
313         )
314         ax.scatter(x_nodes, y_nodes, color=default_colors[i + 1])
315         ax.plot(
316             x_coords,
317             y_error,
318             label=rf"$|R_{n[i]}(x)|$",
319             linestyle="dashed",
320             color='e82351'
321         )

```

```

322     ax.set_title(title + f" (n = {n[i]})")
323     path = f"./plots_2/q4_3/p{n[i]}.png"
324 else:
325     # Plot  $|R(x)|$  for all n
326     for i in range(len(n)):
327         x_nodes = np.linspace(-1, 1, n[i] + 1)
328         y_nodes = calculate_lagrange_output(x_nodes, x_nodes, abs)
329         y_poly = calculate_lagrange_output(x_nodes, x_coords, abs)
330         y_error = np.abs(y_coords - y_poly)
331         ax.plot(
332             x_coords,
333             y_error,
334             label=rf"$|R_{{{n[i]}}}(x)|$",
335             color=default_colors[i]
336         )
337     path = "./plots_2/q4_3/error.png"
338     ax.set_title(title + " Error")
339
340
341     # Configure axis and save figure
342     ax.set_xlabel(r"$x$")
343     ax.set_ylabel(r"$y$")
344     ax.legend()
345     fig.savefig(path, dpi=300)
346     plt.show()
347     ax.cla()
348 # -----

```