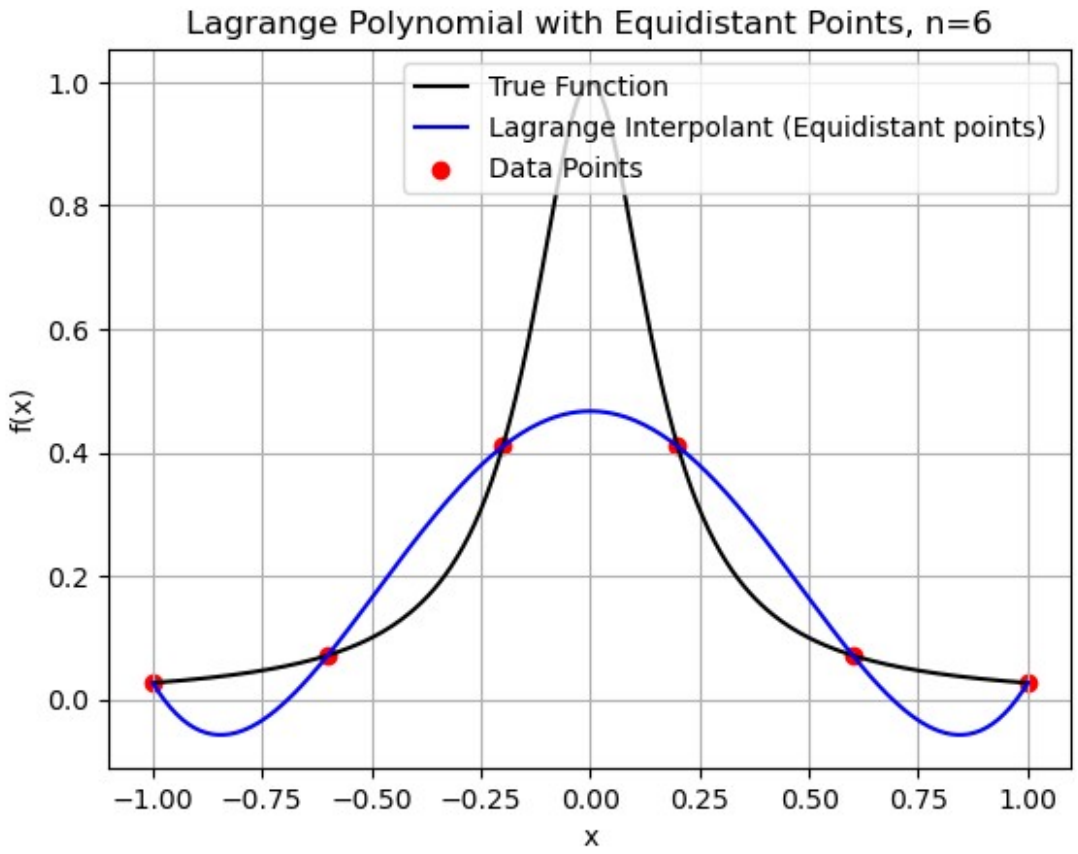
```
# Import dependencies
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import lagrange
```

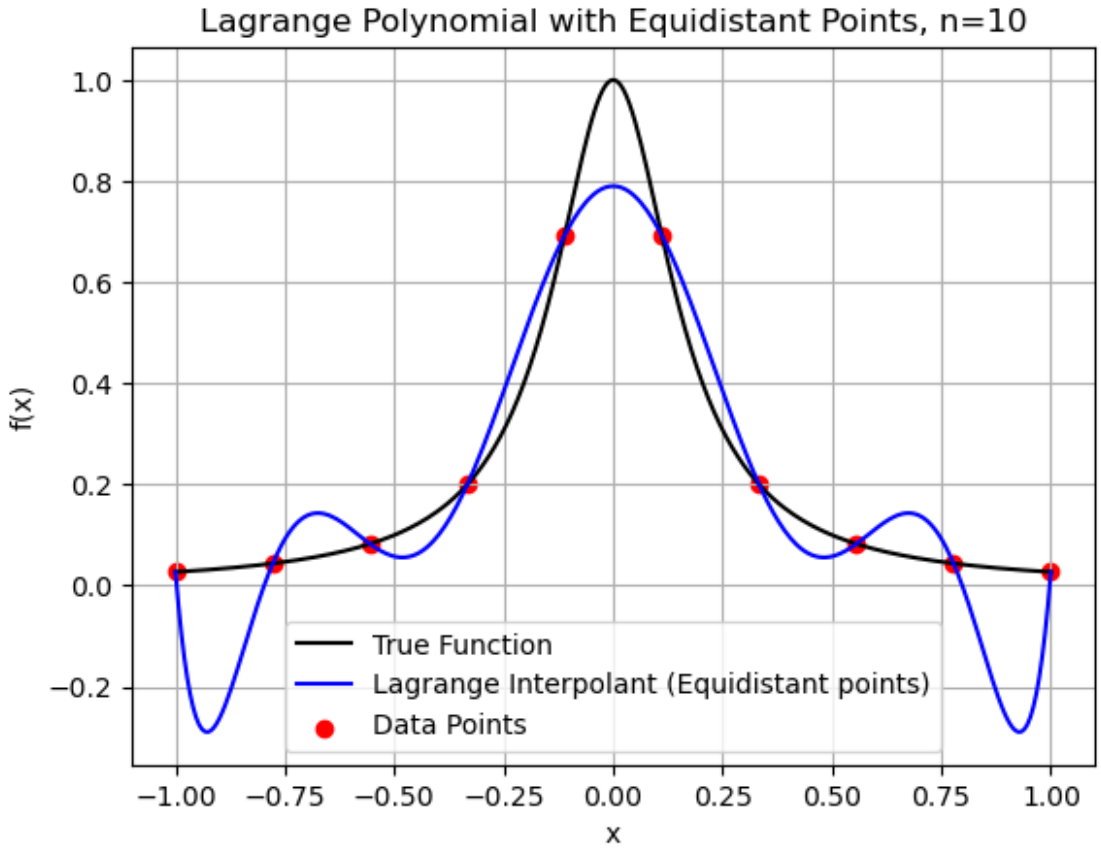
Problem 2.1.a

```
# Define the function f(x)
def f(x):
    return 1 / (1 + 36 * x**2)

# Generate equidistant points
n = 6  # n can be adjusted
x_equidistant = np.linspace(-1, 1, n)
y_equidistant = f(x_equidistant)

# Perform Lagrange interpolation
polynomial_equidistant = lagrange(x_equidistant, y_equidistant)

# Generate points for plotting
x_plot = np.linspace(-1, 1, 300)
y_plot = polynomial_equidistant(x_plot)

# Plot the true function and the Lagrange interpolant
plt.plot(x_plot, f(x_plot), label="True Function", color='black')
plt.plot(x_plot, y_plot, label="Lagrange Interpolant (Equidistant
points)", color='blue')
plt.scatter(x_equidistant, y_equidistant, color='red', label="Data
Points")
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.title('Lagrange Polynomial with Equidistant Points, n=6')
plt.grid(True)
plt.show()
```



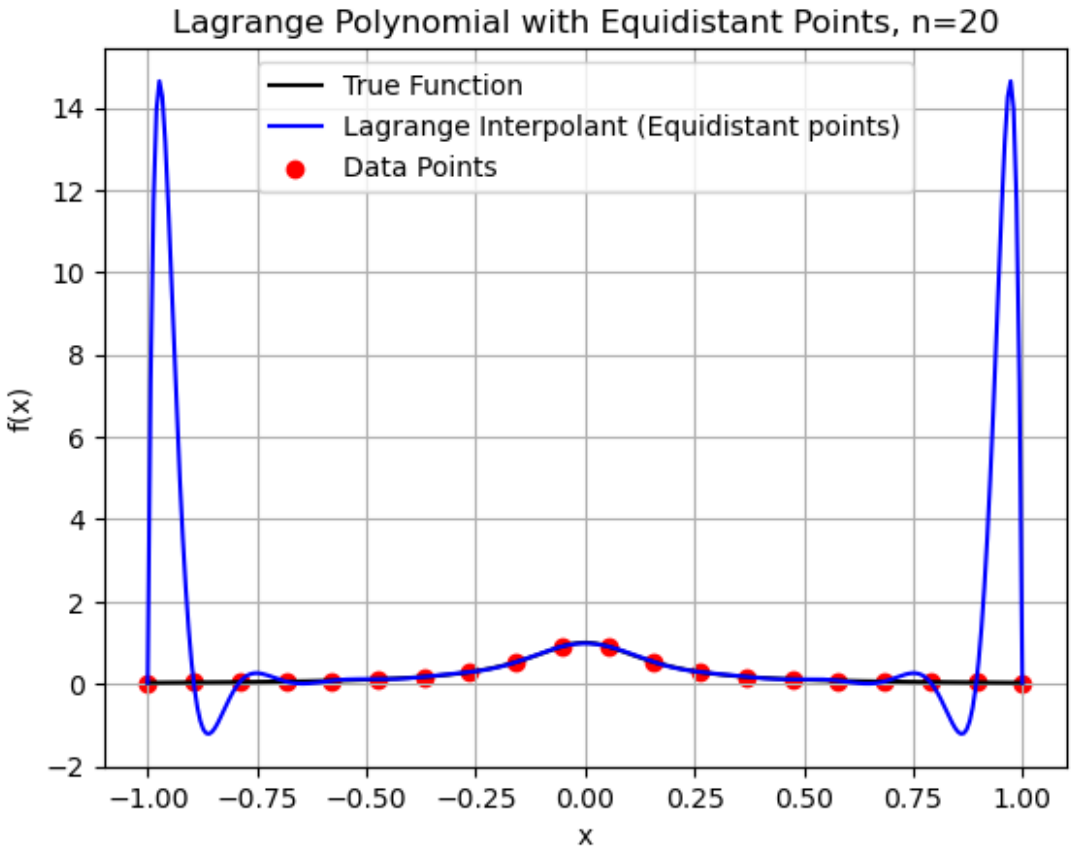

```python
# Define the function f(x)
def f(x):
    return 1 / (1 + 36 * x**2)

# Generate equidistant points
n = 10  # n can be adjusted
x_equidistant = np.linspace(-1, 1, n)
y_equidistant = f(x_equidistant)

# Perform Lagrange interpolation
polynomial_equidistant = lagrange(x_equidistant, y_equidistant)

# Generate points for plotting
x_plot = np.linspace(-1, 1, 300)
y_plot = polynomial_equidistant(x_plot)

# Plot the true function and the Lagrange interpolant
plt.plot(x_plot, f(x_plot), label="True Function", color='black')
plt.plot(x_plot, y_plot, label="Lagrange Interpolant (Equidistant
points)", color='blue')
plt.scatter(x_equidistant, y_equidistant, color='red', label="Data
Points")
plt.xlabel('x')
```

```
plt.ylabel('f(x)')
plt.legend()
plt.title('Lagrange Polynomial with Equidistant Points, n=10')
plt.grid(True)
plt.show()
```




```
# Define the function f(x)
def f(x):
    return 1 / (1 + 36 * x**2)

# Generate equidistant points
n = 20  # n can be adjusted
x_equidistant = np.linspace(-1, 1, n)
y_equidistant = f(x_equidistant)

# Perform Lagrange interpolation
polynomial_equidistant = lagrange(x_equidistant, y_equidistant)

# Generate points for plotting
x_plot = np.linspace(-1, 1, 300)
y_plot = polynomial_equidistant(x_plot)

# Plot the true function and the Lagrange interpolant
```

```
plt.plot(x_plot, f(x_plot), label="True Function", color='black')
plt.plot(x_plot, y_plot, label="Lagrange Interpolant (Equidistant
points)", color='blue')
plt.scatter(x_equidistant, y_equidistant, color='red', label="Data
Points")
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.title('Lagrange Polynomial with Equidistant Points, n=20')
plt.grid(True)
plt.show()
```



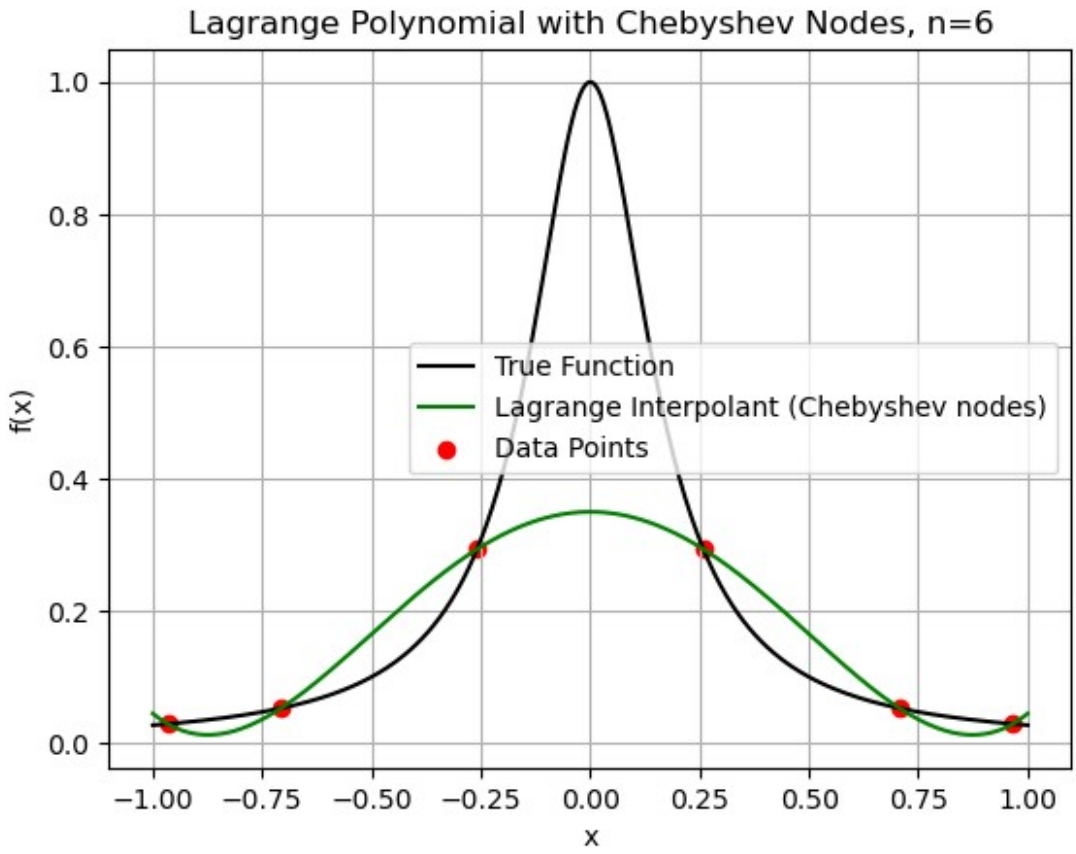
When using equidistant nodes, the polynomial interpolant can oscillate significantly near the boundaries (this is known as the Runge phenomenon), especially as n increases.

- For small n (e.g. n=6): The interpolation is less accurate compared to the true function.
- For moderate n (e.g. n=10): As n increases, the interpolant starts showing more pronounced oscillations, especially at the edges of the interval. The approximation is still relatively accurate in the middle of the interval, but near the boundaries ($x = -1$ and $x = 1$), the polynomial starts to deviate from the true function, introducing larger errors.

- For large n (e.g. n=20): The interpolant exhibits significant oscillations, particularly near the boundaries, which is a manifestation of Runge's phenomenon. While the interpolant passes through all the data points exactly, it no longer provides a good approximation of the true function at the edges. The peaks and valleys near $x = -1$ and $x = 1$ are highly exaggerated, indicating poor approximation in those regions, even though the fit remains accurate in the middle part of the interval.
- Runge's phenomenon is the observation that, for certain functions (especially those with rapid changes), using high-degree polynomials for interpolation with equidistant nodes can cause large oscillations, particularly at the interval's edges. This phenomenon occurs because the polynomial tries to fit all the data points exactly, and as the number of points increases, it becomes more difficult for the polynomial to balance the fit in the middle of the interval with the fit at the boundaries. As a result, the polynomial tends to oscillate more at the edges.

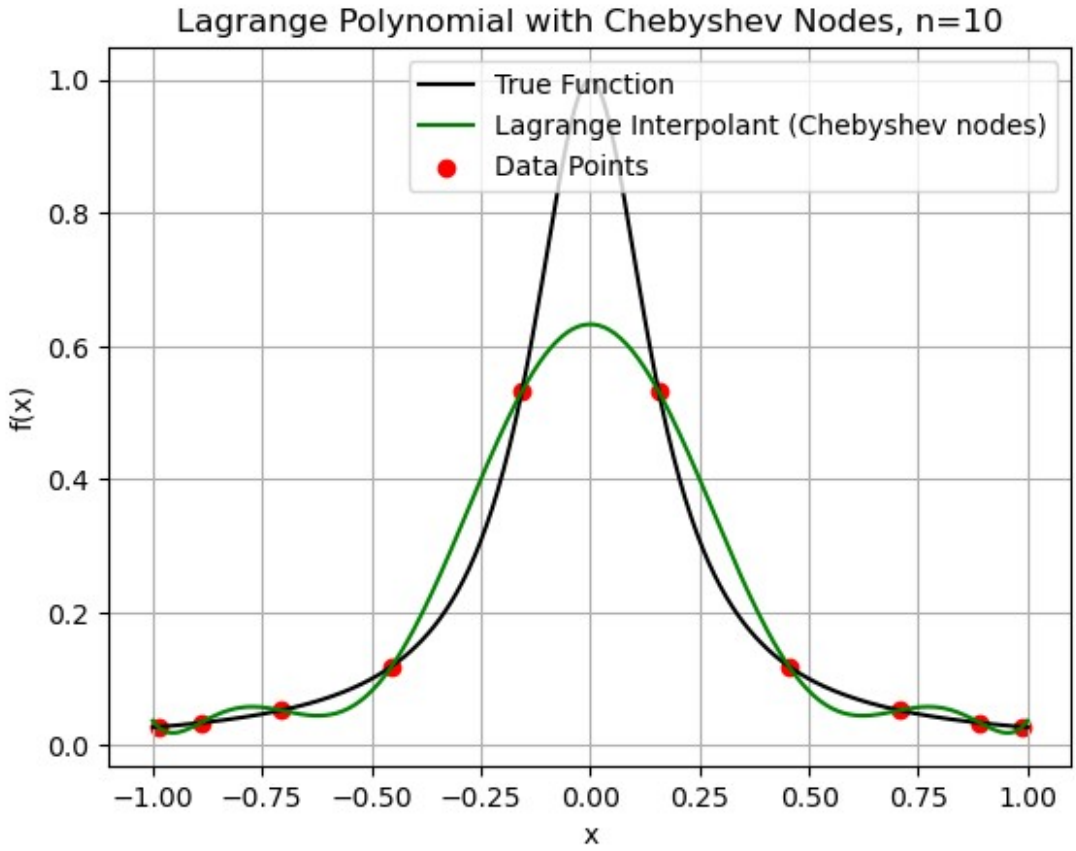
Problem 2.1.b

```
# Generate Chebyshev nodes
n = 6  # n can be adjusted
i = np.arange(n)
x_chebyshev = np.cos((2*i+1) * np.pi / (2*(n)))
y_chebyshev = f(x_chebyshev)

# Perform Lagrange interpolation with Chebyshev nodes
polynomial_chebyshev = lagrange(x_chebyshev, y_chebyshev)

# Generate points for plotting
y_plot_chebyshev = polynomial_chebyshev(x_plot)

# Plot the true function and the Lagrange interpolant (Chebyshev
nodes)
plt.plot(x_plot, f(x_plot), label="True Function", color='black')
plt.plot(x_plot, y_plot_chebyshev, label="Lagrange Interpolant
(Chebyshev nodes)", color='green')
plt.scatter(x_chebyshev, y_chebyshev, color='red', label="Data
Points")
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.title('Lagrange Polynomial with Chebyshev Nodes, n=6')
plt.grid(True)
plt.show()
```



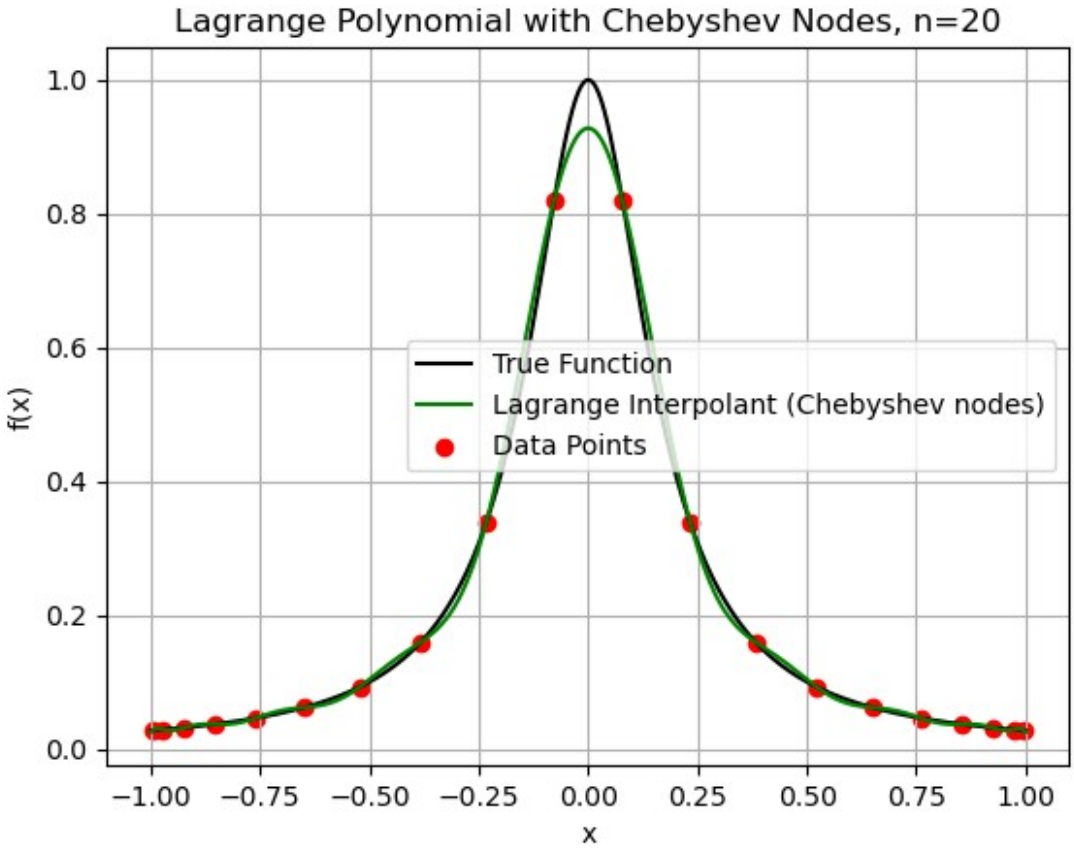

```
# Generate Chebyshev nodes
n = 10  # n can be adjusted
i = np.arange(n)
x_chebyshev = np.cos((2*i+1) * np.pi / (2*(n)))
y_chebyshev = f(x_chebyshev)

# Perform Lagrange interpolation with Chebyshev nodes
polynomial_chebyshev = lagrange(x_chebyshev, y_chebyshev)

# Generate points for plotting
y_plot_chebyshev = polynomial_chebyshev(x_plot)

# Plot the true function and the Lagrange interpolant (Chebyshev
nodes)
plt.plot(x_plot, f(x_plot), label="True Function", color='black')
plt.plot(x_plot, y_plot_chebyshev, label="Lagrange Interpolant
(Chebyshev nodes)", color='green')
plt.scatter(x_chebyshev, y_chebyshev, color='red', label="Data
Points")
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.title('Lagrange Polynomial with Chebyshev Nodes, n=10')
```

```
plt.grid(True)
plt.show()
```




```
# Generate Chebyshev nodes
n = 20  # n can be adjusted
i = np.arange(n)
x_chebyshev = np.cos((2*i + 1) * np.pi / (2*(n)))
y_chebyshev = f(x_chebyshev)

# Perform Lagrange interpolation with Chebyshev nodes
polynomial_chebyshev = lagrange(x_chebyshev, y_chebyshev)

# Generate points for plotting
y_plot_chebyshev = polynomial_chebyshev(x_plot)

# Plot the true function and the Lagrange interpolant (Chebyshev
nodes)
plt.plot(x_plot, f(x_plot), label="True Function", color='black')
plt.plot(x_plot, y_plot_chebyshev, label="Lagrange Interpolant
(Chebyshev nodes)", color='green')
plt.scatter(x_chebyshev, y_chebyshev, color='red', label="Data
Points")
plt.xlabel('x')
```

```
plt.ylabel('f(x)')
plt.legend()
plt.title('Lagrange Polynomial with Chebyshev Nodes, n=20')
plt.grid(True)
plt.show()
```



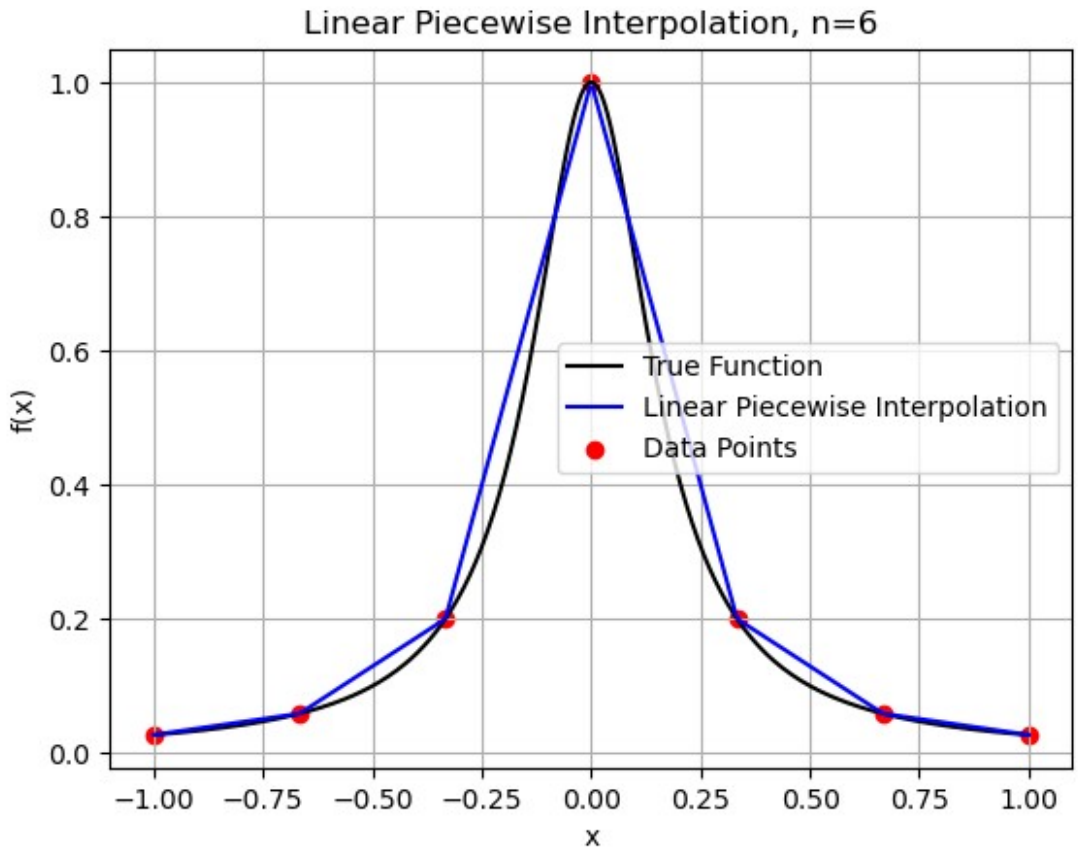
Using Chebyshev nodes significantly improves the interpolation quality, particularly as the number of nodes n increases, which helps avoid the oscillations seen in the case of equidistant nodes (Runge's phenomenon).

- For small n (e.g. n=6): The interpolant is still somewhat inaccurate, especially near the edges of the interval. The interpolant does not perfectly match the true function but performs reasonably well.
- For moderate n (e.g. n=10): The interpolation improves, with the polynomial becoming more accurate across the interval. There are still minor deviations from the true function near the boundaries, but this is much less severe than the oscillations seen with equidistant nodes.
- For large n (e.g. n=20): The interpolant closely follows the true function throughout the interval. The interpolation near the boundaries is significantly more accurate compared to equidistant nodes, with almost no severe oscillations.

- Chebyshev nodes minimize the maximum error in polynomial interpolation according to Chebyshev's minimax theorem, which states that the interpolation error is minimized when the nodes are distributed in a Chebyshev fashion.
- Chebyshev nodes are not uniformly distributed like equidistant nodes. Instead, they are more densely spaced near the boundaries of the interval. This clustering of points near the edges reduces the error that typically arises at the boundaries in polynomial interpolation with equidistant nodes.

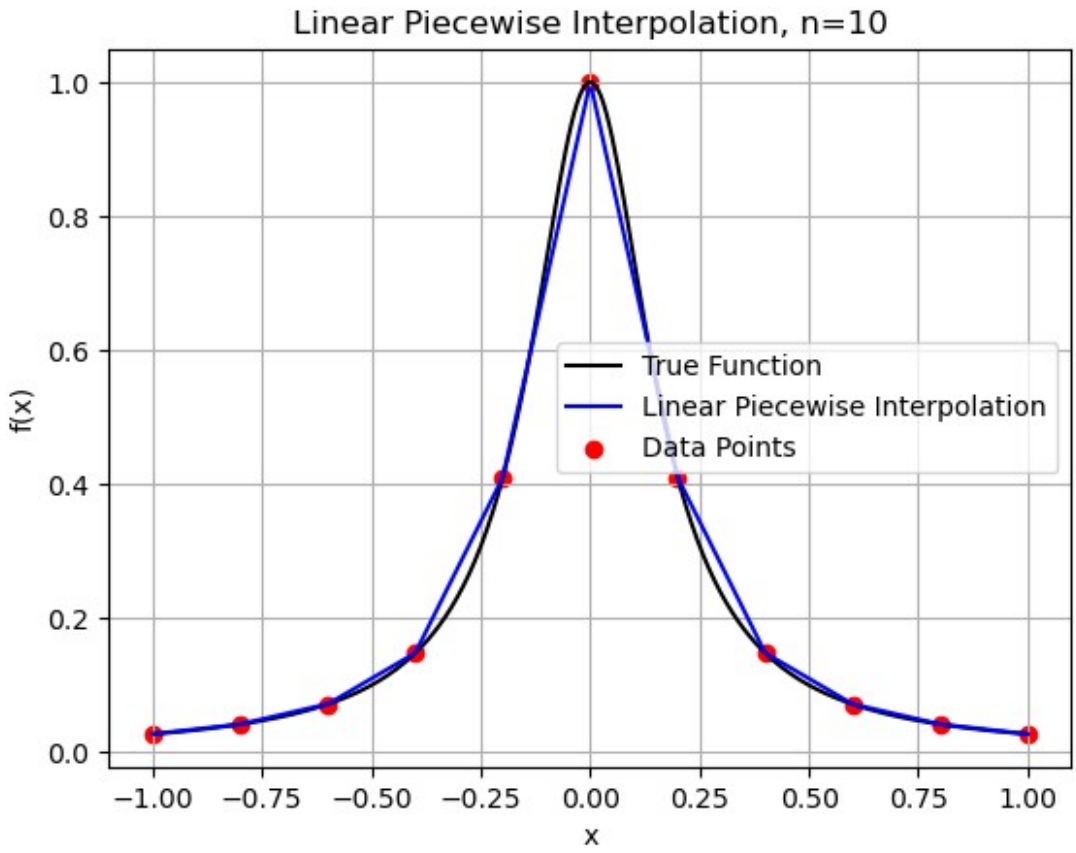
Problem 2.2

```
# Define the function f(x)
def f(x):
    return 1 / (1 + 36 * x**2)

# Number of points (n can be adjusted)
n = 6   # n can be adjusted

# Generate equidistant points
x_equidistant = np.linspace(-1, 1, n+1)
y_equidistant = f(x_equidistant)

# Generate x values for the plot
x_plot = np.linspace(-1, 1, 300)
y_plot = f(x_plot)

# Perform linear piecewise interpolation
y_linear = np.interp(x_plot, x_equidistant, y_equidistant)

# Plot the true function and the linear interpolation
plt.plot(x_plot, y_plot, label="True Function", color='black')
plt.plot(x_plot, y_linear, label="Linear Piecewise Interpolation",
color='blue')
plt.scatter(x_equidistant, y_equidistant, color='red', label="Data
Points")
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.title('Linear Piecewise Interpolation, n=6')
plt.grid(True)
plt.show()
```



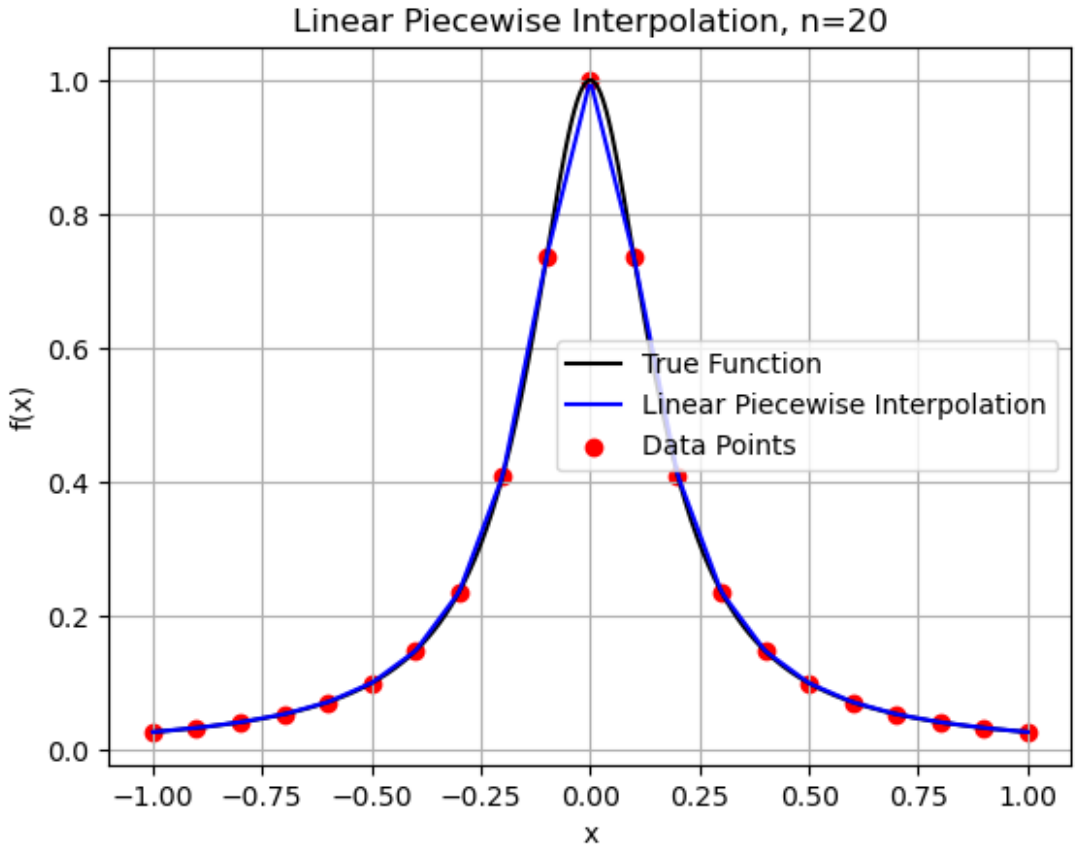

```
# Define the function f(x)
def f(x):
    return 1 / (1 + 36 * x**2)

# Number of points (n can be adjusted)
n = 10   # n can be adjusted

# Generate equidistant points
x_equidistant = np.linspace(-1, 1, n+1)
y_equidistant = f(x_equidistant)

# Generate x values for the plot
x_plot = np.linspace(-1, 1, 300)
y_plot = f(x_plot)

# Perform linear piecewise interpolation
y_linear = np.interp(x_plot, x_equidistant, y_equidistant)

# Plot the true function and the linear interpolation
plt.plot(x_plot, y_plot, label="True Function", color='black')
plt.plot(x_plot, y_linear, label="Linear Piecewise Interpolation",
color='blue')
plt.scatter(x_equidistant, y_equidistant, color='red', label="Data
```

```
Points")
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.title('Linear Piecewise Interpolation, n=10')
plt.grid(True)
plt.show()
```




```
# Define the function f(x)
def f(x):
    return 1 / (1 + 36 * x**2)

# Number of points (n can be adjusted)
n = 20   # n can be adjusted

# Generate equidistant points
x_equidistant = np.linspace(-1, 1, n+1)
y_equidistant = f(x_equidistant)

# Generate x values for the plot
x_plot = np.linspace(-1, 1, 300)
y_plot = f(x_plot)
```

```
# Perform linear piecewise interpolation
y_linear = np.interp(x_plot, x_equidistant, y_equidistant)

# Plot the true function and the linear interpolation
plt.plot(x_plot, y_plot, label="True Function", color='black')
plt.plot(x_plot, y_linear, label="Linear Piecewise Interpolation",
color='blue')
plt.scatter(x_equidistant, y_equidistant, color='red', label="Data
Points")
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.title('Linear Piecewise Interpolation, n=20')
plt.grid(True)
plt.show()
```



- As n increases, the linear piecewise interpolant approximates the true function better.
- Increasing n leads to shorter segments, which better approximate the local shape of the function.

- More data points provide more information, leading to a better approximation of the true function.
- Stability is maintained, as linear interpolation avoids the oscillatory problems seen with high-degree polynomials.