HW 7 - Lena Kellner, due 20th Oct

① Matrix $V = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix}$

The higher the # for N gets, the better it approximates the function - except of the areas of the left & right limit of the interval.

(Pictures & code see below)

② Unfortunately, I could not make the code work since I could not implement neither of the barycentric lagrange formulas ...

I've attached my attempts of the code below.

③ By using Chebyshev points you place more points where the function tends to vary the most. Because of runge, it's near the endpoints of the interval.

So, it leads to a more accurate interpolation and I assume that it does not fail anymore because of that.

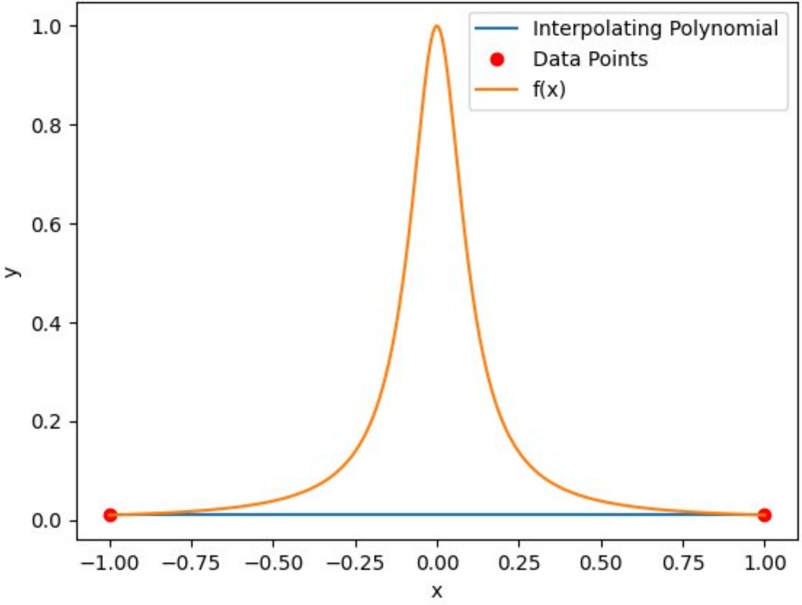
Code 1

```python
def interpolate(x, y):
    # n = len(x)
    V = np.vander(x, increasing=True)
    c = np.linalg.solve(V, y)
    return c

def eval_pol(x, c):
    n = len(c)
    return sum(c[i] * x**i for i in range(n))

def f(x):
    return 1 / (1 + (10 * x)**2)

N = 7  # #of points for the approximation

x_data = np.array([-1 + (i-1) * (2/(N-1)) for i in range(1, N+1)])
y_data = f(x_data)
coefs = interpolate(x_data, y_data)

# Evaluate the polynomial on a finer grid
x_fine = np.linspace(-1, 1, 1001)
y_polynomial = sum(c * x_fine**i for i, c in enumerate(coefs))

# Plot
plt.plot(x_fine, y_polynomial, label='Interpolating Polynomial')
plt.plot(x_data, y_data, 'ro', label='Data Points')
plt.plot(x_fine, f(x_fine), label='f(x) fine')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

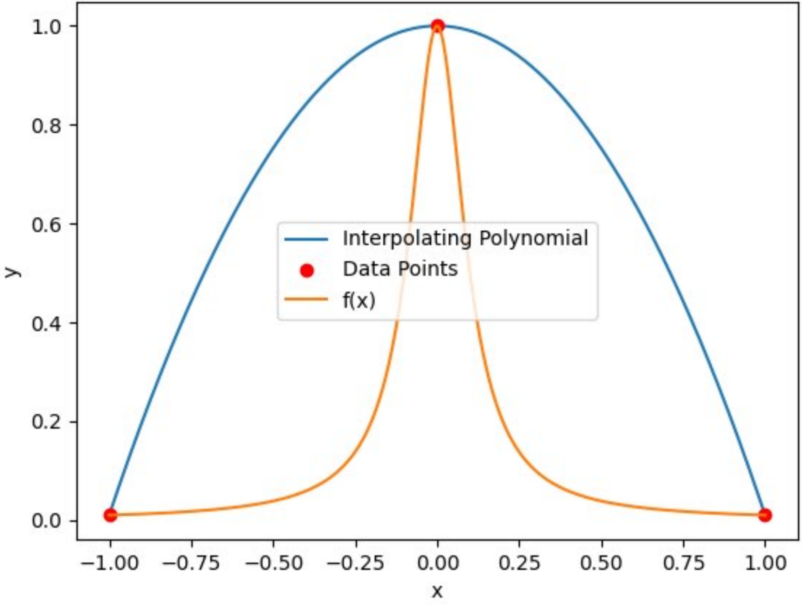
```
Coefficients: [ 1.00000000e+00 -4.05115870e-15 -1.11640376e+01  2.20328652e-14
  2.81632210e+01 -1.79817065e-14 -1.79892824e+01]
Polinomial [0.00990099 0.04467664 0.0785679  ... 0.0785679  0.04467664 0.00990099]
```

N=2



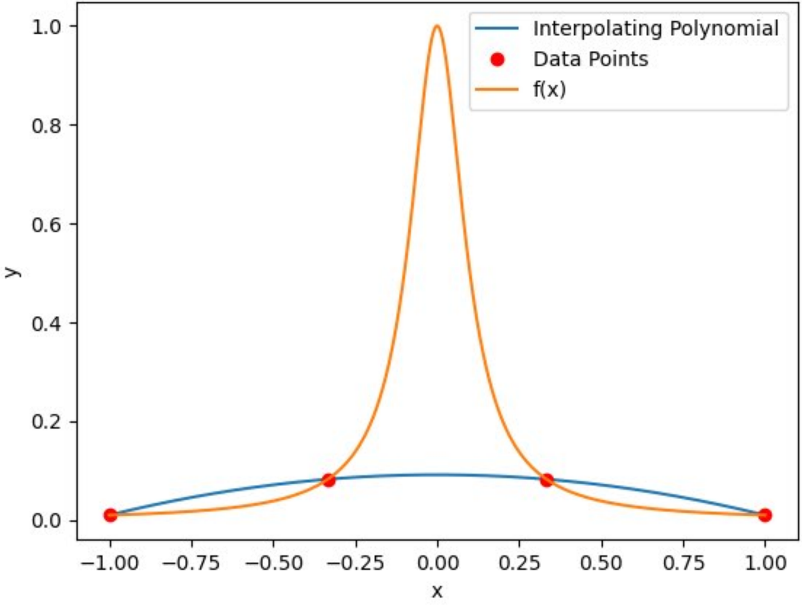
Interpolating Polynomial
Data Points
f(x)
1.0
0.8
0.6
0.4
0.2
0.0
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

N=3

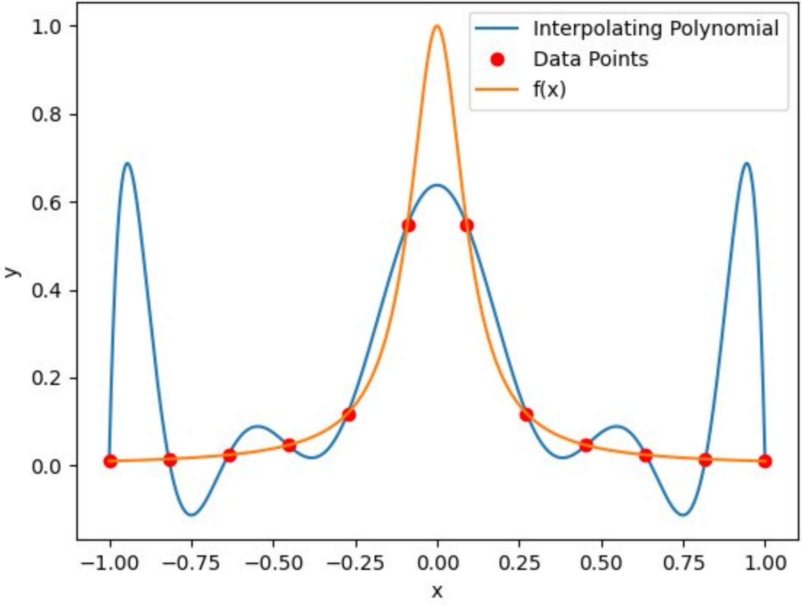


1.0
0.8
0.6
0.4
0.2
0.0
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x
Interpolating Polynomial
Data Points
f(x)

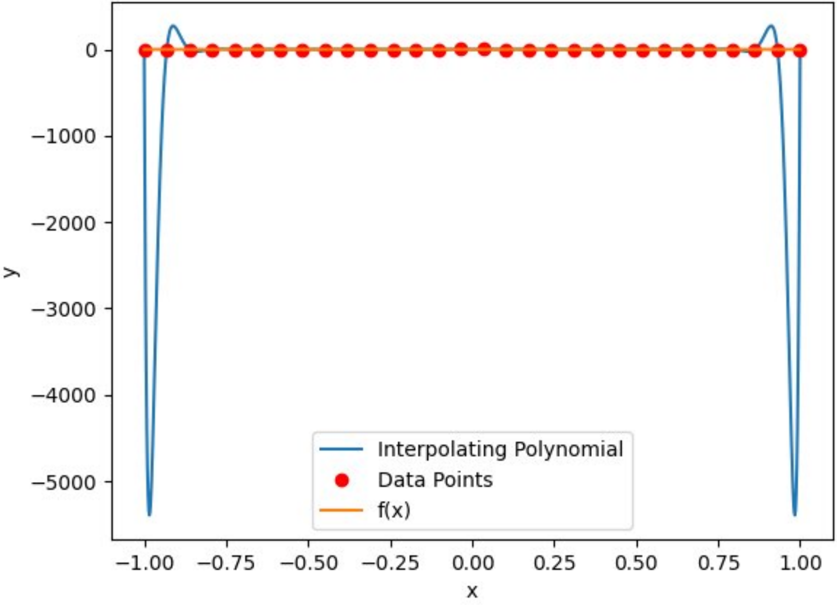
N = 4


Interpolating Polynomial
Data Points
f(x)
1.0
0.8
0.6
0.4
0.2
0.0
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

N=7


Interpolating Polynomial
Data Points
f(x)
y
x
1.0
0.8
0.6
0.4
0.2
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

N=30



0
−1000
−2000
−3000
−4000
−5000
y
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x
Interpolating Polynomial
Data Points
f(x)

Attempts for Code $\boxed{2}$

```
def barycentric_lagrange_interpolation(x, f, x_fine):
    n = len(x)
    m = len(x_fine)
    p = np.zeros(m)
    #j = random.randint(0, m)
    w = np.zeros(m)
    #for i in range(m+1):
    #    phi = np.prod(x_fine-x[i])
    for j in range (m+1):
        for i in range (m+1):
            if i != j:
                w[j] = 1 / np.prod(x[j] - x[i])

   # for j in range (n):
    #    if x[j] != x_fine:
     #       pn += (wj * f[j] / (x_fine - x[j]))
      #      pd += (wj / (x_fine - x[j]))

    #p = pn/pd

    for j in range(n):
        wj = 1 / np.prod(x[j] - np.delete(x,j))
        p += (wj * f[j] / (x_fine - x[j]))
    return p

def f(x):
    return 1 / (1 + (10 * x)**2)

N = 7  # #of points for the approximation
x_data = np.array([-1 + (i-1) * (2/(N-1)) for i in range(1, N+1)])
y_data = f(x_data)
coefs = interpolate(x_data, y_data)

# Evaluate the polynomial on a finer grid
x_fine = np.linspace(-1, 1, 1001)
y_monomial = sum(c * x_fine**i for i, c in enumerate(coefs))
y_barycentric = barycentric_lagrange_interpolation(x_data, y_data, x_fine)
```