CMSC 117 Problem Set 4
Florenz Jaizzer P. Calderon

ps4item1.py

The `char_lagrange` function computes the Lagrange characteristic polynomials for a set of interpolation nodes. It takes two inputs: `z`, the points where the polynomials are evaluated, and `x`, the interpolation nodes. The function initializes a matrix to store the evaluated polynomials and then iterates over each node `x_k` and each evaluation point `z_i`, computing the corresponding Lagrange polynomial using the formula:

$$\ell_k(z_i) = \prod_{t \neq k} \frac{z_i - x_t}{x_k - x_t}.$$

Finally, it returns the matrix of evaluated characteristic polynomials, which can be used for Lagrange interpolation.

```python
def char_lagrange(z, x):
    # Determine the length of the inputs
    n = len(x)
    m = len(z)

    # Initialize the matrix to store the evaluated Lagrange
      characteristic polynomials
    evaluated_characteristic_polynomial = [[0 for _ in range(m)]
       for _ in range(n)]

    # Evaluate the Lagrange characteristic polynomial at x_k with
      z_i
    for k in range(n):
        for i in range(m):
            result = 1
            for t in range(n):
                if t != k:
                    result = result * (z[i] - x[t]) / (x[k] - x[t
                       ])
            evaluated_characteristic_polynomial[k][i] = result
    return evaluated_characteristic_polynomial
```

The `lagrange` function computes the Lagrange interpolation at given points `z` using known values `x` (interpolation nodes) and `y` (function values at `x`). It first evaluates the Lagrange characteristic polynomials at points `z` using the `char_lagrange` function. The function then iterates over each point `z_i` and computes the Lagrange interpolation formula by summing the products of the values `y_k` and the corresponding characteristic polynomials. The final result is a list of evaluated Lagrange interpolated values at the points `z`.

```python
def lagrange(z, x, y):
    # Obtain the evaluated Lagrange chracteristic polynomial at
        points z = [z_0,...,z_m]
    evaluated_characteristic_polynomial = char_lagrange(z, x)

    # Determine the length of y = [y_0,..., y_n]
    y_length = len(y)

    # Determine the length of z = [z_0,..., z_m]
    m = len(z)

    # Initialize the vector that will store the evaluated Lagrange
         interpolation formula
    evaluated_lagrange_interpolation_formula = [0 for _ in range(m
        )]

    # Evaluate the Lagrange interpolation formula at z
    for i in range(m):
        result = 0
        for k in range(y_length):
            result = result + (y[k] *
                evaluated_characteristic_polynomial[k][i])
        evaluated_lagrange_interpolation_formula[i] = result

    return evaluated_lagrange_interpolation_formula
```

The **given_function** is a mathematical function defined as $f(x) = \frac{1}{x^4-3x^2+4}$. It takes an input value x and returns the value of the function evaluated at x, where the denominator is a quartic polynomial $x^4 - 3x^2 + 4$.

```python
# Given function to interpolate
def given_function(x):
    return 1 / (x**4 - 3*x**2 + 4)
```

The code below generates a plot for Lagrange interpolation using 5 equidistant nodes in the interval $[0, 3]$. The original function is plotted in purple, with the interpolation polynomial $P_4(x)$ shown in dashed pink. The nodes are marked in green, and the plot includes axis labels, a title, and a legend.

```python
"""
Create the first figure for P4 with the original function
"""
# Nodes = 5
x_lagrange_0 = np.linspace(0, 3, num=5)
y_lagrange_0 = given_function(x_lagrange_0)
```

```
z_0 = np.linspace(0, 3, num=100)

plt.figure(figsize=(8, 6))
plt.plot(np.linspace(0, 3, 1000), given_function(np.linspace(0, 3,
    1000)), label='f(x)', color='purple')
plt.plot(z_0, lagrange(z_0, x_lagrange_0, y_lagrange_0), label=r"
   P$_{4}$x", linestyle='--', color='#e54988')
plt.scatter(x_lagrange_0, y_lagrange_0, color='green', alpha=1,
   label='Nodes')
plt.legend(loc="upper right")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Lagrange Interpolation P$_{4}$x with 5 equidistant
   nodes on [0, 3]")
plt.show()
```

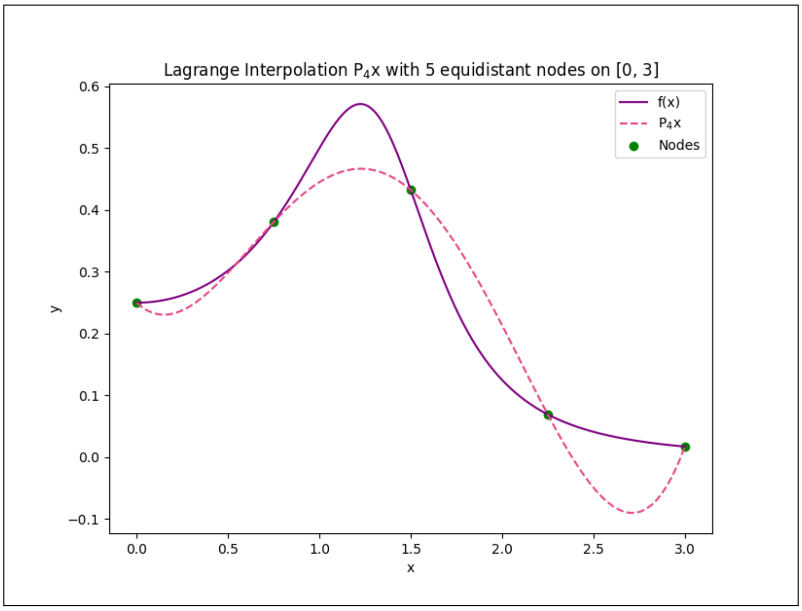

Figure 1: Lagrange Interpolation P_4x with 5 equidistant nodes on [0, 3]

The code below generates a plot for Lagrange interpolation using 9 equidistant nodes in the interval $[0, 3]$. The original function is plotted in purple, with the interpolation polynomial $P_8(x)$ shown in dashed yellow. The nodes are marked in green, and the plot includes axis labels, a title, and a legend.

```
"""
Create the second figure for P8 with the original function
"""
# Nodes = 9
```

```
x_lagrange_1 = np.linspace(0, 3, num=9)
y_lagrange_1 = given_function(x_lagrange_1)
z_1 = np.linspace(0, 3, num=100)

plt.figure(figsize=(8, 6))
plt.plot(np.linspace(0, 3, 1000), given_function(np.linspace(0, 3,
     1000)), label='f(x)', color='purple')
plt.plot(z_1, lagrange(z_1, x_lagrange_1, y_lagrange_1), label=r"
    P$_{8}$x", linestyle='--', color='#fec97b')
plt.scatter(x_lagrange_1, y_lagrange_1, color='green', alpha=1,
    label='Nodes')
plt.legend(loc="upper right")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Lagrange Interpolation P$_{8}$x with 9 equidistant
    nodes on [0, 3]")
plt.show()
```

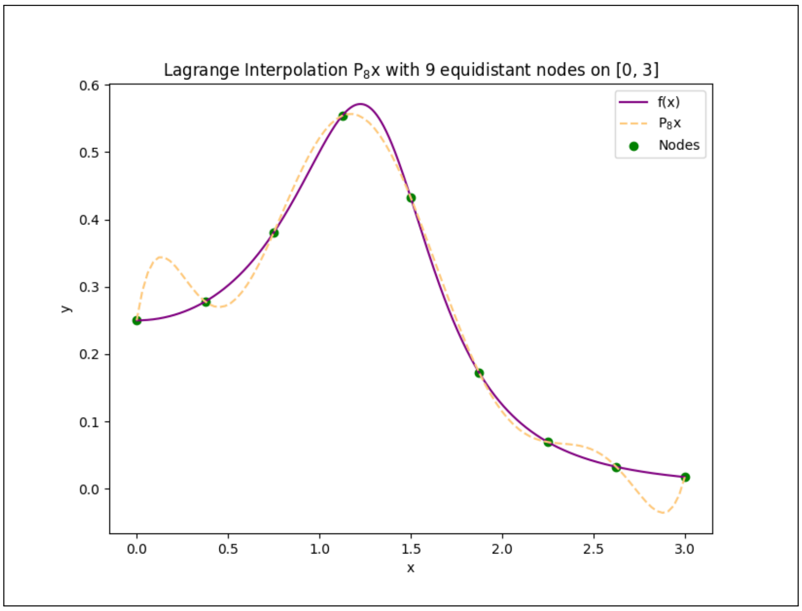

Figure 2: Lagrange Interpolation P_8x with 9 equidistant nodes on [0, 3]

The code below generates a plot for Lagrange interpolation using 14 equidistant nodes in the interval $[0, 3]$. The original function is shown in purple, with the interpolation polynomial $P_{13}(x)$ represented in dashed teal. The nodes are marked in green, and the plot includes axis labels, a title, and a legend.

```
"""
Create the third figure for P13 with the original function
```

```
"""
# Nodes = 14
x_lagrange_2 = np.linspace(0, 3, num=14)
y_lagrange_2 = given_function(x_lagrange_2)
z_2 = np.linspace(0, 3, num=100)

plt.figure(figsize=(8, 6))
plt.plot(np.linspace(0, 3, 1000), given_function(np.linspace(0, 3,
    1000)), label='f(x)', color='purple')
plt.plot(z_2, lagrange(z_2, x_lagrange_2, y_lagrange_2), label=r"
    P$_{13}$x", linestyle='--', color='#67bfaf')
plt.scatter(x_lagrange_2, y_lagrange_2, color='green', alpha=1,
    label='Nodes')
plt.legend(loc="upper right")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Lagrange Interpolation P$_{13}$x with 14 equidistant
    nodes on [0, 3]")
plt.show()
```

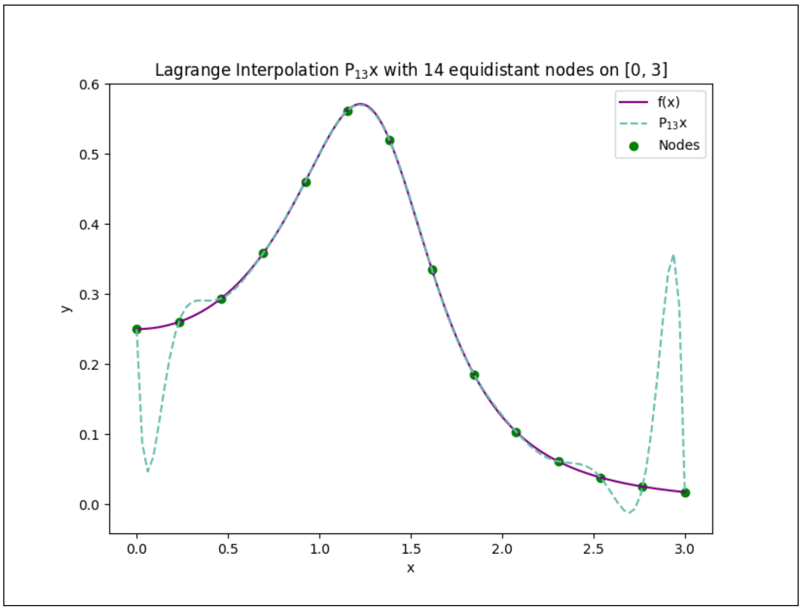

Figure 3: Lagrange Interpolation P_{13}x with 14 equidistant nodes on [0, 3]

The code below displays the original function in purple and the three Lagrange interpolation polynomials $P_4(x)$, $P_8(x)$, and $P_{13}(x)$ using 5, 9, and 14 equidistant nodes, respectively. The nodes are marked in green, with a dummy plot used to label the nodes. The figure includes axis labels, a title, and a legend to distinguish between the different plots.

```
"""
Create the combined plot with all three Lagrange functions and the original function
"""
plt.figure(figsize=(8, 6))
plt.plot(np.linspace(0, 3, 1000), given_function(np.linspace(0, 3, 1000)), label='f(x)', color='purple')

# Plot all Lagrange functions
plt.plot(z_0, lagrange(z_0, x_lagrange_0, y_lagrange_0), label=r"P$_{4}$x", linestyle='--', color='#e54988')
plt.plot(z_1, lagrange(z_1, x_lagrange_1, y_lagrange_1), label=r"P$_{8}$x", linestyle='--', color='#fec97b')
plt.plot(z_2, lagrange(z_2, x_lagrange_2, y_lagrange_2), label=r"P$_{13}$x", linestyle='--', color='#67bfaf')

# Scatter plot for nodes (in green)
plt.scatter(x_lagrange_0, y_lagrange_0, color='green', alpha=1)
plt.scatter(x_lagrange_1, y_lagrange_1, color='green', alpha=1)
plt.scatter(x_lagrange_2, y_lagrange_2, color='green', alpha=1)

# Add a dummy plot for the 'nodes' label
plt.scatter([], [], color='green', label='Nodes')

# Show combined plot
plt.legend(loc="upper right")
plt.xlabel("x")
plt.ylabel("y")
plt.title("All the Lagrange Interpolation with Original Function and Nodes")
plt.show()
```

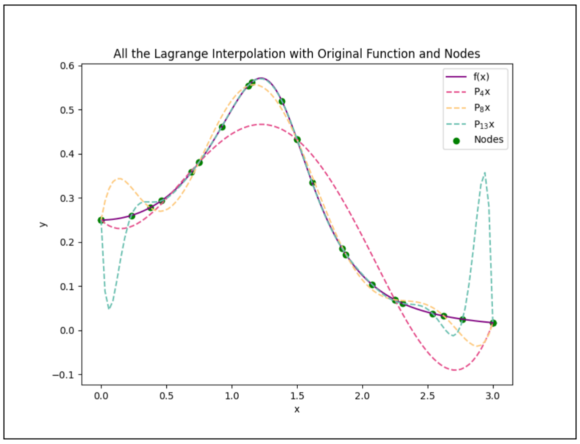

Figure 4: All the Lagrange Interpolation with Original Function and Nodes

ps4item2.py

The code below imports Python libraries were used to create the visualizations, including the color maps and 3D scatter plots:

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import gridspec
```

Moving on the main features, the function `char_lagrange(z, x)` computes the Lagrange characteristic polynomials at the points in `z` for the interpolation nodes in `x`. It iterates over each node and evaluates the polynomial using the Lagrange basis formula:

$$L_k(z_i) = \prod_{\substack{m=0 \\ m \neq k}}^{n} \frac{z_i - x_m}{x_k - x_m}$$

where x_k are the interpolation nodes and z_i are the evaluation points. The result is stored in a matrix, with each element corresponding to the evaluated Lagrange polynomial for a given node and evaluation point.

```python
def char_lagrange(z, x):
    # Determine the length of the inputs
    nx = len(x)
    z_length = len(z)

    # Initialize the matrix to store the evaluated Lagrange characteristic polynomials
    evaluated_characteristic_polynomial = [[0 for _ in range(z_length)] for _ in range(nx)]

    # Evaluate the Lagrange characteristic polynomial at x_k with z_i
    for k in range(nx):
        for i in range(z_length):
            result = 1
            for m in range(nx):
                if m != k:
                    result = result * (z[i] - x[m]) / (x[k] - x[m])
            evaluated_characteristic_polynomial[k][i] = result
    return evaluated_characteristic_polynomial
```

The function `lagrange2D(zx, zy, x, y, w)` computes the 2D Lagrange interpolation at points in `zx` and `zy`, given the interpolation nodes `x`, `y`, and the weight matrix `w`. The Lagrange interpolation formula is expressed as:

$$P(x,y) = \sum_{i=0}^{n_x}\sum_{j=0}^{n_y} w_{i,j}\ell_{n_x,i}(x)\ell_{n_y,j}(y)$$

where $\ell_{n_x,i}(x)$ and $\ell_{n_y,j}(y)$ are the Lagrange characteristic polynomials in the x-axis and y-axis, respectively, calculated as:

$$\ell_{n_x,i}(x) = \prod_{\substack{m=0 \\ m\neq i}}^{n_x} \frac{x - x_m}{x_i - x_m}, \quad \ell_{n_y,j}(y) = \prod_{\substack{n=0 \\ n\neq j}}^{n_y} \frac{y - y_n}{y_j - y_n}$$

The result is stored in a matrix where each element corresponds to the evaluated 2D Lagrange interpolation at a given point (z_x, z_y).

```
def lagrange2D(zx, zy, x, y, w):
    # Determine the length of x
    nx = len(x)

    # Determine the length of y
    ny = len(y)

    # Determine the length of zx
    mx = len(zx)

    # Determine the lenght of zy
    my = len(zy)

    # Evaluate the Lagrange characteristic polynomials on the x-
       axis
    evaluated_characteristic_polynomial_x = char_lagrange(zx, x)

    # Evaluate the Lagrange characteristic polynomials on the y-
       axis
    evaluated_characteristic_polynomial_y = char_lagrange(zy, y)

    # Initialize the vector that will store the evaluated 2D
       Lagrange interpolation formula
    evaluated_lagrange_interpolation_formula_2D = [[0 for _ in
       range(my)] for _ in range(mx)]

    # Evaluate the 2D Lagrange interpolation formula
    for zx_index in range(mx):
        for zy_index in range(my):
            outer_result = 0
            for i in range(nx):
                inner_result = 0
                for j in range(ny):
                    inner_result += w[i][j] *
                       evaluated_characteristic_polynomial_x[i][
```

```
                            zx_index] *
                            evaluated_characteristic_polynomial_y[j][
                            zy_index]
                outer_result += inner_result
            evaluated_lagrange_interpolation_formula_2D[zx_index][
                zy_index] = outer_result

    return evaluated_lagrange_interpolation_formula_2D
```

The code initializes the interpolation nodes and the corresponding 2D grid of points (x, y, w) for Lagrange interpolation:

- The x-interpolation nodes are created using `np.linspace(-1, 2, 10)`, which generates 10 equally spaced points along the x-axis between -1 and 2.
- Similarly, the y-interpolation nodes are created using `np.linspace(-2, 2, 10)`, generating 10 equally spaced points along the y-axis between -2 and 2.
- The matrix $W_{\text{interpolation\_nodes}}$ (representing the 2D grid of points (x, y, w)) is predefined as a 10x10 array, with each element corresponding to a specific value in the grid.
- The `np.meshgrid()` function is then used to generate two 2D arrays, $X_{\text{interpolation\_nodes}}$ and $Y_{\text{interpolation\_nodes}}$, which represent the grid of x- and y-coordinates in the 2D plane. These arrays are then combined with $W_{\text{interpolation\_nodes}}$ to form the full set of interpolation points (x, y, w).

```
"""
Initialize the interpolation nodes to be used for Lagrange
    Interpolation
"""

# Interpolation Nodes
x_interpolation_nodes = np.linspace(-1, 2, 10)
y_interpolation_nodes = np.linspace(-2, 2, 10)

# W data from Google Drive
W_interpolation_nodes = [
        [2.260329406981054240e-06, 7.673192648360093273e
            -06,2.085790014230277754e-05,4.539992976248485417e
            -05,7.912794612036177339e-05,1.104319447771195890e
            -04,1.234098040866795612e-04,1.104319447771195890e
            -04,7.912794612036177339e-05,4.539992976248485417e-05],
        [2.669954063117813402e-05, 9.063754966554979050e
            -05,2.463784042319182343e-04,5.362746091796782681e
            -04,9.346778420779018430e-04,1.304448009856573176e
            -03,1.457745525197095757e-03,1.304448009856574260e
            -03,9.346778420779018430e-04,5.362746091796782681e-04],
```

```
        [2.124529250204041243e-04, 7.212188707337513017e
            -04,1.960476150657309170e-03,4.267231069936552011e
            -03,7.437395431040693013e-03,1.037972147383890015e
            -02,1.159953667524437834e-02,1.037972147383890882e
            -02,7.437395431040693013e-03,4.267231069936552011e-03],
        [1.138802761345787586e-03, 3.865920139472804086e
            -03,1.050866046540278181e-02,2.287346491123889991e
            -02,3.986636782372491444e-02,5.563799827784279145e
            -02,6.217652402211629181e-02,5.563799827784281227e
            -02,3.986636782372493526e-02,2.287346491123889991e-02],
        [4.112076444169911020e-03, 1.395936125214680690e
            -02,3.794547802860596952e-02,8.259326325034499483e
            -02,1.439525417455284062e-01,2.009019558827834229e
            -01,2.245117666465471229e-01,2.009019558827834784e
            -01,1.439525417455284340e-01,8.259326325034499483e-02],
        [1.000231941277372576e-02, 3.395510563531246168e
            -02,9.229954663187718567e-02,2.009019558827835061e
            -01,3.501538267511653535e-01,4.886790313053682722e
            -01,5.461081359780511901e-01,4.886790313053683832e
            -01,3.501538267511654090e-01,2.009019558827835061e-01],
        [1.638955379021359016e-02, 5.563799827784279145e
            -02,1.512397596904957175e-01,3.291929878079055682e
            -01,5.737534207374326289e-01,8.007374029168078389e
            -01,8.948393168143696785e-01,8.007374029168081719e
            -01,5.737534207374328510e-01,3.291929878079055682e-01],
        [1.809090996006276764e-02, 6.141363151714355345e
            -02,1.669395585727310727e-01,3.633656399769043532e
            -01,6.333132437099517897e-01,8.838598318931834008e
            -01,9.877302162356106363e-01,8.838598318931837339e
            -01,6.333132437099520118e-01,3.633656399769043532e-01],
        [1.345180507918091774e-02, 4.566515461063024722e
            -02,1.241307599718489835e-01,2.701867276014116581e
            -01,4.709108788478272856e-01,6.572090736282378831e
            -01,7.344436719297313676e-01,6.572090736282381052e
            -01,4.709108788478273966e-01,2.701867276014116581e-01],
        [6.737946999085467001e-03, 2.287346491123889991e
            -02,6.217652402211629181e-02,1.353352832366127023e
            -01,2.358770829856999540e-01,3.291929878079055682e
            -01,3.678794411714423340e-01,3.291929878079056238e
            -01,2.358770829857000650e-01,1.353352832366127023e-01]
    ]
X_interpolation_nodes,Y_interpolation_nodes = np.meshgrid(
    x_interpolation_nodes,y_interpolation_nodes)
```

The code below generates the figures before performing Lagrange Interpolation, including the following plots:

- A scatter plot showing the interpolation nodes within the region $[-1,2]\times[-2,2]$, represented by $X_{\text{interpolation\_nodes}}$ and $Y_{\text{interpolation\_nodes}}$. This plot does not yet apply the interpolation.

- A set of two subplots:
  - A color map of the interpolation values, where $W_{\text{interpolation\_nodes}}$ represents the values associated with the nodes.
  - A 3D scatter plot showing the relationship between the interpolation nodes ($X_{\text{interpolation\_nodes}}$ and $Y_{\text{interpolation\_nodes}}$) and their corresponding values $W_{\text{interpolation\_nodes}}$, with a color map applied based on the values.

Note that $W_{\text{interpolation}}$ represents the interpolation values in this context and is crucial for the later stages of Lagrange interpolation.

```python
"""
Create the figures before performing Lagrange Interpolation
"""

plt.title("Interpolation Nodes in [-1,2] x [-2,2]")
plt.scatter(X_interpolation_nodes,Y_interpolation_nodes)
plt.xlabel("x"); plt.ylabel("y")
plt.show()

fig = plt.figure(figsize=plt.figaspect(.5))
fig.tight_layout()
gs = gridspec.GridSpec(1, 2, width_ratios=[3, 1])

ax = fig.add_subplot(gs[1])
ax.pcolormesh(X_interpolation_nodes, Y_interpolation_nodes,
    W_interpolation_nodes)
ax.set_xlabel('x',fontsize = 15); ax.set_ylabel('y',rotation = 0,
    fontsize = 15)
ax.set_title('Color Map of Interpolation Points')

ax = fig.add_subplot(gs[0],projection='3d')
ax.scatter(X_interpolation_nodes, Y_interpolation_nodes,
    W_interpolation_nodes, c=W_interpolation_nodes, cmap='viridis')
ax.set_xlabel('x',fontsize = 15); ax.set_ylabel('y',fontsize = 15)
    ; ax.set_zlabel('w',linespacing=3.4,fontsize = 15)
ax.set_title('3D Scatter Plot of Interpolation Points')
plt.show()
```

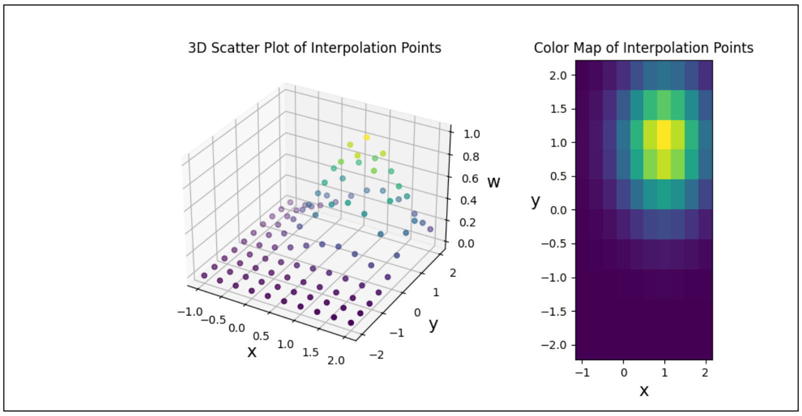

Figure 5: Generated Figures Before Lagrange Interpolation

The code below generates the figures after performing Lagrange Interpolation with a $M \times N$ grid of points:

- The x-coordinates are spaced over the interval $[-1, 2]$, and the y-coordinates are spaced over the interval $[-2, 2]$, with M and N grid points, respectively.
- A set of two subplots:
  - A color map displaying the result of the Lagrange interpolation for the $M \times N$ grid, where the values W_{lagrange} represent the interpolated values.
  - A 3D scatter plot showing the relationship between the grid points $(X_{\text{lagrange}}, Y_{\text{lagrange}})$ and their corresponding interpolated values W_{lagrange}, with a color map applied based on the values.

The variables M and N specify the resolution of the grid used in the interpolation process. The interpolated values W_{lagrange} are obtained using the 2D Lagrange interpolation function.

```
"""
Create the figures after performing Lagrange Interpolation with M
    x N gridpoints
"""

M = 100
N = 100

x_lagrange = np.linspace(-1, 2, M)
y_lagrange = np.linspace(-2, 2, N)
X_lagrange,Y_lagrange = np.meshgrid(x_lagrange,y_lagrange)
W_lagrange = lagrange2D(x_lagrange, y_lagrange,
    x_interpolation_nodes, y_interpolation_nodes,
    W_interpolation_nodes)

```

```python
fig = plt.figure(figsize=plt.figaspect(.5))
fig.tight_layout()
gs = gridspec.GridSpec(1, 2, width_ratios=[3, 1])

ax = fig.add_subplot(gs[1])
ax.pcolormesh(X_lagrange, Y_lagrange, W_lagrange)
ax.set_xlabel('x',fontsize = 15); ax.set_ylabel('y',rotation = 0, fontsize = 15)
ax.set_title(f'Color Map of {M} x {N} Grid Points')

ax = fig.add_subplot(gs[0],projection='3d')
ax.scatter(X_lagrange, Y_lagrange, W_lagrange, c=W_lagrange, cmap='viridis')
ax.set_xlabel('x',fontsize = 15); ax.set_ylabel('y',fontsize = 15); ax.set_zlabel('w',linespacing=3.4,fontsize = 15)
ax.set_title(f'3D Scatter Plot of {M} x {N} Grid Points')
plt.show()
```

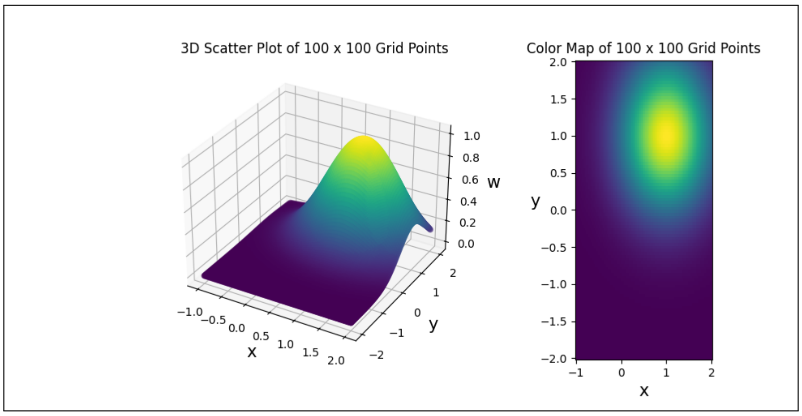

Figure 6: Generated Figures After Performing Lagrange Interpolation

ps4item3.py

The function `NormalNCWeights(n)` calculates the normalized weights for a numerical integration method using equidistant nodes in the interval $[-1, 1]$. The implementation of this function and the use of LU decomposition to solve the linear system were derived from the previous problem set 3, where the function `LU_solve()` was introduced.

The steps involved in the function are as follows:

1. **Initialization of the Interval**: The interval is set to $[-1, 1]$ (this can be modified through the variables `left_endpoint` and `right_endpoint`).

2. **Equidistant Node Generation**: The function generates $n + 1$ equidistant nodes within the interval $[-1, 1]$ using the `np.linspace()` function. These nodes will be the points at which the integration will be evaluated.

3. **Vandermonde Matrix Construction**: A Vandermonde matrix $V_{\text{transposed}}$ is created, where each entry V_{ij} represents x_j^i, with x_j being the j-th node.

4. **Integral Evaluation**: The integral of the polynomial $p_n(x)$ over the interval $[-1, 1]$ is computed for each power i, and stored in the vector y. The integral for each power is computed as:
$$y_i = \frac{(x_{\text{right}}^{i+1} - x_{\text{left}}^{i+1})}{i + 1}$$
where x_{left} and x_{right} are the left and right endpoints of the interval, respectively.

5. **Normalization Factor**: A normalization factor is calculated as:
$$\text{normalization\_factor} = \frac{n}{x_{\text{right}} - x_{\text{left}}}$$
which is used to adjust the weights.

6. **LU Decomposition**: The function uses LU decomposition (`LU_solve()`) to solve for the unnormalized weights. The `LU_solve()` function and its associated dependencies were introduced in the previous problem set 3.

7. **Weight Normalization**: The unnormalized weights are multiplied by the normalization factor to obtain the final normalized weights.

```python
def NormalNCWeights(n):
    # Initialize an arbitraty interval
    left_endpoint = -1
    right_endpoint = 1

    # Generate the equidistant nodes at the interval [
        left_endpoint, right_endpoint]
    x = np.linspace(left_endpoint, right_endpoint, num = n + 1)

    # Generate the Vandermonde matrix using the nodes
    V_transposed = [[0 for _ in range(n + 1)] for _ in range(n +
        1)]
    for i in range(n + 1):
        for j in range(n + 1):
            V_transposed[i][j] = (x[j])**(i)

    # Initialize the matrix to store the evaluated integral of
        polynomial p_n(x) on [left_endpoint, right_endpoint]
    y = [0 for _ in range(n + 1)]

    # Evaluate the integral of polynomial p_n(x) on [left_endpoint
        , right_endpoint]
```

```
    for i in range(n + 1):
        y[i] = ((right_endpoint)**(i + 1) - (left_endpoint)**(i +
            1)) /(i + 1)

    # Calculate the normalization factor
    normalization_factor = n / (right_endpoint - left_endpoint)

    # Solve for the unnormalized weights
    unnormalized_weights = LU_solve(V_transposed, y)

    # Normalized the weights
    normalized_weights = unnormalized_weights *
        normalization_factor

    return normalized_weights
```

The function `NCQuad(f, a, b, w)` implements the Newton-Cotes quadrature method to approximate the integral of a given function f over the interval $[a, b]$ using $n + 1$ nodes with corresponding weights w. The function steps are as follows:

1. **Node Count**: The number of nodes is determined by the length of the weight vector w, where $\text{node\_count} = \text{len}(w)$.
2. **Degree of Precision**: The degree of precision is calculated as $n = \text{node\_count} - 1$.
3. **Step Size Calculation**: The step size h is computed as:

$$h = \frac{b - a}{n}$$

   where a and b are the left and right bounds of the integration interval, respectively.
4. **Weights Adjustment**: The actual weights are obtained by multiplying the normalized weights w by the step size h, i.e., $\text{weights} = w \times h$.
5. **Integral Calculation**: The integral is then computed as the sum of the function evaluations at the quadrature nodes weighted by the corresponding weights:

$$I = \sum_{k=0}^{n} w_k \cdot f(a + h \cdot k)$$

   where w_k is the weight corresponding to the node at $a + h \cdot k$.

```
def NCQuad(f, a, b, w):
    # Calculate the number of nodes
    node_count = len(w)

    # Calculate the degree of precision by subtracting 1 to the
        number of nodes
```

```python
    n = node_count - 1

    # Calculate the step-size 'h'
    h = (b - a) / n

    # Obtain the actual weights from the normalized weights
    weights = w * h

    # Initialize the variable to store the integral
    integral = 0

    # Calculate the the integral quadrature of f over the interval
        [a, b]
    for k in range(node_count):
        integral += weights[k] * f(a + h*k)

    return integral
```

The function `CompositeNCQuad(f, a, b, n, m)` implements the Composite Newton-Cotes quadrature method to approximate the integral of a given function f over the interval $[a, b]$ by dividing the interval into m subintervals, and applying the Newton-Cotes quadrature method with $n + 1$ nodes for each subinterval. The function steps are as follows:

1. **Interval Subdivision**: The interval $[a, b]$ is divided into m subintervals, with each subinterval having an equal length.
2. **Subinterval Boundaries**: For each subinterval, the left and right endpoints are calculated as:
$$\text{left\_endpoint} = a + \frac{(b-a)}{m} \cdot i, \quad \text{right\_endpoint} = \text{left\_endpoint} + \frac{(b-a)}{m}$$
where i ranges from 0 to $m - 1$ for the subintervals.
3. **Quadrature on Subintervals**: The Newton-Cotes quadrature method `NCQuad` is applied to each subinterval using $n + 1$ nodes and corresponding weights, with the integral over each subinterval computed individually. The results are then summed to obtain the final integral.

```python
def CompositeNCQuad(f, a, b, n, m):
    # Initialize the variable to store the integral
    integral = 0

    # Calculate the number of subintervals
    for i in range(m):
        left_endpoint_of_current_subinterval = a + (b / m) * i
        right_endpoint_of_current_subinterval =
            left_endpoint_of_current_subinterval + (b / m)
        integral += NCQuad(f, left_endpoint_of_current_subinterval
            , right_endpoint_of_current_subinterval,
            NormalNCWeights(n))
```

```python

    return integral
```

We approximate the integral of the given function over the interval $[0, 3]$ using the composite Newton-Cotes quadrature method. The method is applied with different values of n and m, where n represents the number of nodes in the Newton-Cotes formula and m is the number of subintervals into which the interval $[0, 3]$ is divided.

$$f(x) = \frac{1}{x^4 - 3x^2 + 4}$$

The approximations are computed as follows:

1. For $n = 3$ and $m = 100$, the result is approximately:

$$\text{Approximation} = \text{CompositeNCQuad}(f, 0, 3, 3, 100)$$

2. For $n = 4$ and $m = 50$, the result is approximately:

$$\text{Approximation} = \text{CompositeNCQuad}(f, 0, 3, 4, 50)$$

3. For $n = 5$ and $m = 25$, the result is approximately:

$$\text{Approximation} = \text{CompositeNCQuad}(f, 0, 3, 5, 25)$$

```python
# Given function to integrate
def given_function(x):
    return 1 / (x**4 - 3*x**2 + 4)

'''
Use CompositeNCQuad to approximate the given function at interval
    [0, 3] with [n, m] given by:
'''

# (i) [3, 100]
print(CompositeNCQuad(given_function, 0, 3, 3, 100))

# (ii) [4, 50]
print(CompositeNCQuad(given_function, 0, 3, 4, 50))

# (iii) [5,25]
print(CompositeNCQuad(given_function, 0, 3, 5, 25))
```

The results of the composite Newton-Cotes quadrature approximation for the given function over the interval $[0, 3]$ with different values of n (degree of precision) and m (number of subintervals) are as follows:

```
0.7702480820563341    (n = 3, m = 100)
0.7702480820720065    (n = 4, m = 50)
0.7702480820711671    (n = 5, m = 25)
```

These results show that even with a lower degree of precision ($n = 3$) and a higher number of subintervals ($m = 100$), the approximation is very close to the exact integral value. Similarly, with a higher degree of precision ($n = 5$) and a smaller number of subintervals ($m = 25$), the approximation remains similarly accurate. This illustrates the relationship between the degree of precision and the number of subintervals:

- For $n = 3$ and $m = 100$, the approximation is already close to the exact value.
- For $n = 4$ and $m = 50$, the result remains close to the first, demonstrating that reducing the number of subintervals doesn't significantly impact accuracy when the degree of precision is increased.
- For $n = 5$ and $m = 25$, the approximation is nearly identical, showing that a higher degree of precision can make up for fewer subintervals.

Thus, the accuracy of the composite Newton-Cotes quadrature method is influenced by both the degree of precision and the number of subintervals, and the results show that balancing these parameters effectively leads to accurate approximations.