104A: Homework 3

Daniel Naylor

Question 1

(a)

We know that

$$\begin{aligned} P_n(x) &= \sum_{j=0}^{n} \left(f_j \prod_{k \neq j} \frac{x - x_k}{x_j - x_k} \right) \\ &= \sum_{k=1}^{n} \left(f[x_0, \dots, x_k] \prod_{j=0}^{k-1} (x - x_j) \right) + f[x_0] \end{aligned}$$

The coefficient for x^n in Newton's DD is $f[x_0, \dots, x_n]$, and each Lagrange Polynomial is degree n. Hence, if we differentiate both equations n times we find

$$\begin{aligned} f[x_0, \dots, x_n] \cdot n! &= n! \cdot \sum_{j=0}^{n} \left(f_j \prod_{k \neq j} \frac{1}{x_j - x_n} \right) \\ \implies f[x_0, \dots, x_k] &= \sum_{j=0}^{n} \frac{f_j}{\prod_{k \neq j} x_j - x_k} \end{aligned}$$

(b)

Consider any permutation of $x_0, \dots, x_n$, and call it $\sigma(x_0, \dots, x_n)$. We find that the sum of the Lagrangian Polynomials remains unchanged since each data point must reappear at some step in construction. Proceeding in a manner similar to above (differentiating n times) we find

$$f[\sigma(x_0, \dots, x_n)] = \sum_{j=0}^{n} \frac{f_j}{\prod_{k \neq j} x_j - x_k}$$

Hence $f[x_0, \dots, x_n] = f[\sigma(x_0, \dots, x_n)]$

Question 2

See attached PDF.

Question 3

We know that

$$f[x_0, \ldots, x_k] = \frac{f[x_1, \ldots, x_k] - f[x_0, \ldots, x_{k-1}]}{x_k - x_0}$$

Hence

$$f^{-1}[y_0, y_1] = \frac{f^{-1}[y_1] - f^{-1}[y_0]}{y_1 - y_0}$$

Yet $f^{-1}[y_j] = f^{-1}(y_j) = x_j$, so

$$f^{-1}[y_0, y_1] = \frac{x_1 - x_0}{y_1 - y_0}$$

Forming a P_1 approximation from this we find

$$\begin{aligned} P_1(0) &= f^{-1}[y_0] + f^{-1}[y_0, y_1](0 - y_0) \\ &= x_0 - y_0 \cdot \frac{x_1 - x_0}{y_1 - y_0} \end{aligned}$$

Given that $f(0.5) = -0.106530659712633$ and $f(0.6) = 0.05118836390597$ (for $f(x) = x - e^{-x}$), we find that

$$\begin{aligned} P_1(0) &= 0.5 + 0.106530659712633 \cdot \frac{0.6 - 0.5}{0.05118836390597 + 0.106530659712633} \\ &= 0.5675445848373328 \end{aligned}$$

Which gives us an approximation for $\bar{x}$, a zero of f.

Question 4

We know that

$$f[\underbrace{x_j, \ldots, x_j}_{n+2 \text{ nodes}}] = \frac{f^{(n+1)}(x_j)}{(n+1)!}$$

Hence $f[0,0] = f'(0) = 0$ and $f[1,1] = f'(1) = 3$. Using this we find

$$\begin{aligned} f[0] &= 0 \\ f[0,0] &= 0 \\ f[0,0,1] &= 2 \\ f[0,0,1,1] &= -1 \end{aligned}$$

Hence we obtain our 3rd degree polynomial by

$$P_3(x) = 0 + 0(x-0) + 2(x-0)^2 - 1(x-0)^2(x-1) = 3x^2 - x^3$$

Question 5

See attached PDF.

Question 6

See attached PDF.

```python
'''
Daniel Naylor
5094024
10/30/2022
'''

import math
import matplotlib.pyplot as plt
import numpy as np
from typing import List, Tuple

# --------------------------------------------------
#                       Problem 2
# --------------------------------------------------

def compute_divided_diff_array(data: List[Tuple[float, float]]) -> List[float]:
    '''
    Given a dataset, return a 1-dimensional array of Divided Difference coefficients.

    The returned array will be of the form:
    ```
    [
        f[x_0],
        f[x_0, x_1],
        ...
        f[x_0, x_1, ..., x_n]
    ]
    '''
    n_plus_1 = len(data)
    c = [] # the array to return.
    # the name 'c' is used since c[0] corresponds to the first coefficient; easier to read
    for j in range(n_plus_1): # j = 0, ... n
        c.append(data[j][1])

    for k in range(1, n_plus_1): # k = 1, ..., n
        for j in range(n_plus_1-1, k-1, -1): # j = n, n-1, ..., k
            c[j] = (c[j] - c[j-1])/(data[j][0] - data[j-k][0])

    return c

test_data = [
    (0, 1),
    (1, 2),
    (2, 3),
    (3, 4)
]

print(compute_divided_diff_array(test_data))

output: [1, 1.0, 0.0, 0.0]. correct since this is the polynomial x+1

--

class poly_approx:
    '''
    Creates a polynomial approximation given a dataset.
    Acts like a mathematical function.

    ### Attributes
    `data`: The given data
    `coeffs`: The Divided Difference Coefficients, computed with `data`.
    '''
    def __init__(self, data: List[Tuple[float, float]]) -> None:
        '''
        Creates a polynomial given a dataset. The data should be of the form
        ```
        [
```

```
            (x_0, f_0),
            (x_1, f_1),
            ...
            (x_n, f_n)
        ]
        ```
        For example, `[(0, 1), (1, 2), (2, 3)]` for `x+1`
        would be an appropriate dataset.
        '''
        self.data = data
        self.coeffs = compute_divided_diff_array(data)

    def __call__(self, x: float, /) -> float:
        p = self.coeffs[-1]
        for j in range(len(self.data) - 1, -1, -1): # j = n, n-1, ..., 0
            p = self.coeffs[j] + p * (x - self.data[j][0])

        return p

def f(x: float) -> float:
    '''
    `f(x) = e^(-x^2)`
    '''
    return math.exp(-x**2)

p_10_data = [
    (j/5 - 1, f(j/5 - 1))
    for j in range(11) # j = 0, ..., 10
]

p_10 = poly_approx(p_10_data)

graphing everything...

domain = np.linspace(-1, 1, 100)

error_graph = np.array([
    f(x) - p_10(x)
    for x in domain
])

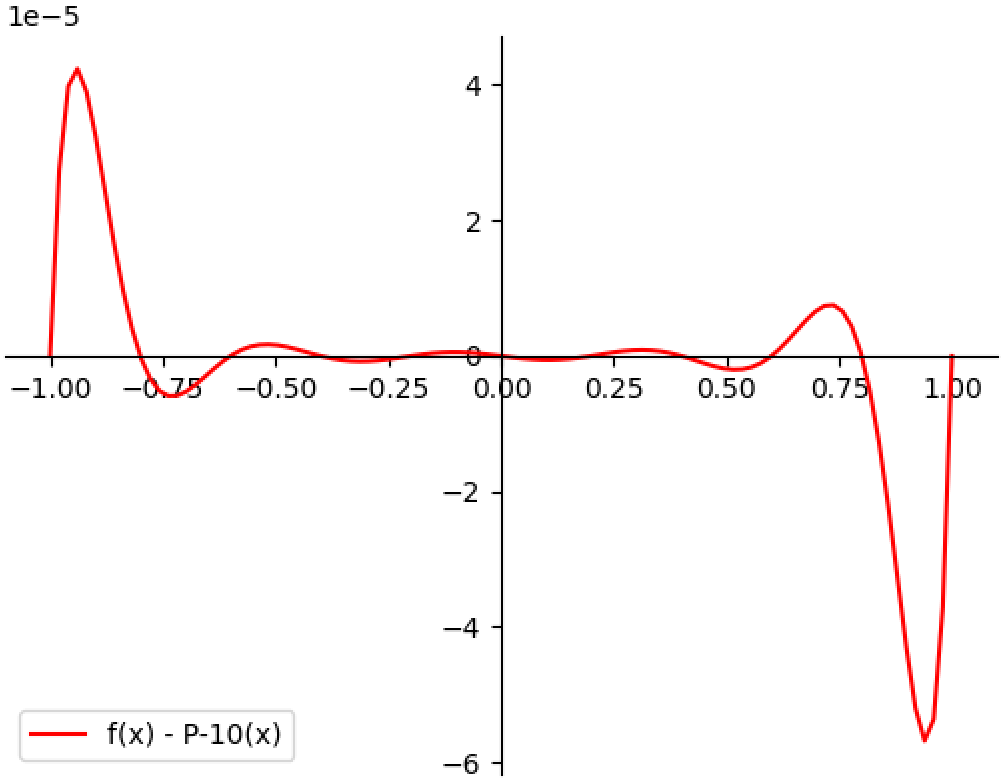
fig = plt.figure()

ax = fig.add_subplot(1, 1, 1)
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')

plt.plot(domain, error_graph, 'r', label='f(x) - P-10(x)')

plt.legend(loc='lower left')

plt.show()
```
```



1e−5
4
2
0
−2
−4
−6
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
f(x) - P-10(x)

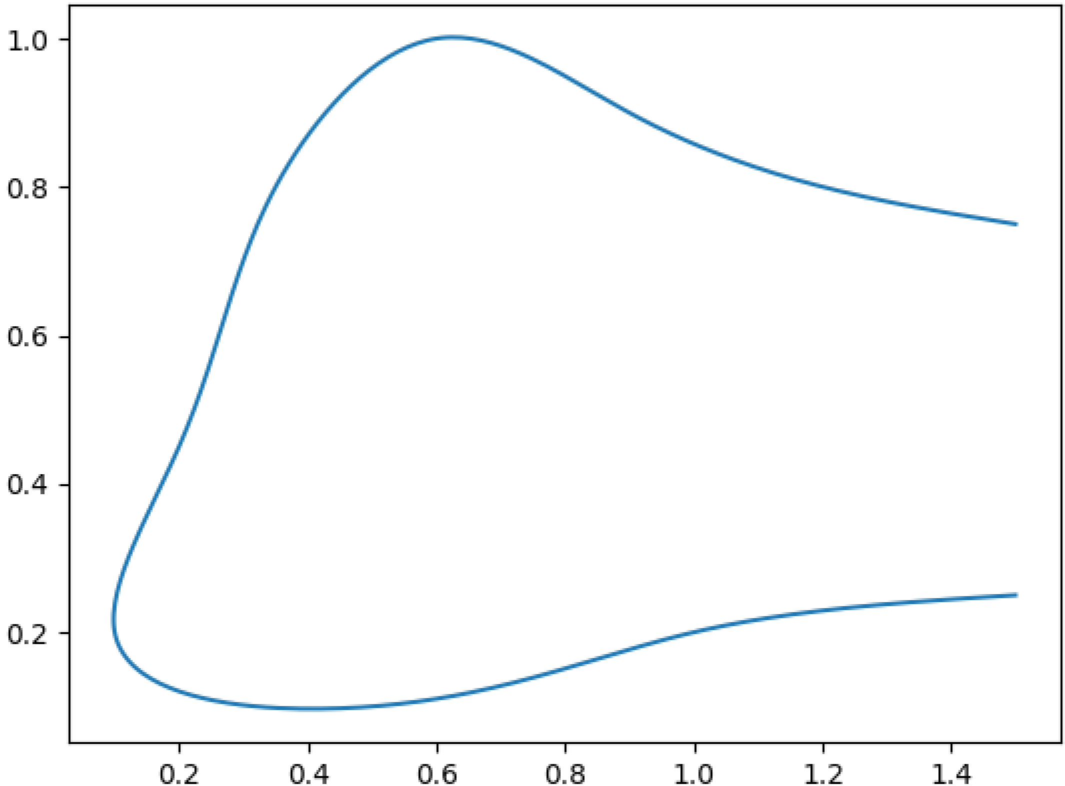
```python
'''
Daniel Naylor
5094024
11/09/2022
'''

from typing import List, Tuple

# --------------------------------------------------
#                       Problem 5
# --------------------------------------------------

# natural spline: z_0 = z_n = 0

class NaturalSplineInterpolation:
    '''
    A natural spline interpolation (piecewise cubic polynomial)
    of a given dataset.

    This class acts like a mathematical function,
    i.e. it can be called with a number as input
    and gives a number as output.
    '''
    def __init__(self, data: List[Tuple[float, float]]) -> None:
        '''
        Initializes a spline interpolation based on the given data.
        Data should be of the form:
        ```
        [
            (x_0, f_0),
            (x_1, f_1),
            ...
            (x_n, f_n)
        ]
        ```
        Each `x` value should be sorted. For example, `[(0, 1), (-1, 0)]` is an invalid data
input,
        and instead should be `[(-1, 0), (0, 1)]`.
        '''
        n = len(data) - 1
        assert n > 0, 'Please provide a valid dataset (at least 2 points)'
        self._data = data

        z = { # natural spline, set z_0 and z_n first
            0: 0,
            n: 0
        }
        h = [ # compute values for h_j, length of each sub-interval.
            data[j+1][0] - data[j][0]
            for j in range(n) # 0, ..., n-1
        ]

        # solve for l, m values
        # use a dictionary for ease of assigning values
        l={} # using 1, ..., n-2
        m={1: 2 * (h[0] + h[1])} # 1, ..., n-1
        for j in range(1, n-1): # 1, ..., n-2
            l[j] = h[j]/m[j]
            m[j+1] = 2 * (h[j] + h[j+1]) - (l[j] * h[j])

        # compute values of y (what the sums of z add to)
        y={
            j: -6 * (data[j][1] - data[j-1][1])/(h[j-1]) + 6 * (data[j+1][1] - data[j]
[1])/h[j]
            for j in range(1, n) # 1, ..., n-1
        }
```

```python
        # compute the values of k (what the matrix product of right-upper and z's produce)
        k={1: y[1]}
        for j in range(2, n):
            k[j] = y[j] - l[j-1] * k[j-1]

        # finally, compute the remaining z values using k
        # note: this abuses the lower triangular property
        z[n-1] = k[n-1]/m[n-1]
        for j in range(n-2, 0, -1): # n-2, ..., 1
            z[j] = (k[j] - h[j] * z[j+1])/m[j]

        # FINALLY, compute a, b, c, d coefficients
        a, b, c, d = {}, {}, {}, {}

        for j in range(n):
            a[j] = (z[j+1] - z[j]) / (6 * h[j])
            b[j] = z[j]/2
            c[j] = (data[j+1][1] - data[j][1])/h[j] - h[j]/6 * (z[j+1] + 2 * z[j])
            d[j] = data[j][1]

        self._a, self._b, self._c, self._d = a, b, c, d

    def __call__(self, x: float, /) -> float:
        # determine suitable piecewise spline S_j using the following heuristic:
        # find maximum j such that x <= x_(j+1).
        # if no such j (for whatever reason), then use j=n-1.
        if (x >= self._data[-2][0]):
            j=len(self._data) - 2
        else:
            j=0
            while (x > self._data[j+1][0]):
                j=j+1

        shifted_x = x - self._data[j][0]

        return self._a[j] * shifted_x**3 + self._b[j] * shifted_x**2 + self._c[j] * shifted_x
+ self._d[j]

# test data

data = [
    (0, 5),
    (1, 3),
    (3, 8),
    (6, -1)
]

approx = NaturalSplineInterpolation(data=data)

print(
    f'1: {approx(1)}',
    f'2: {approx(2)}',
    f'3: {approx(3)}',
    f'5: {approx(5)}',
    sep='\n'
)

# Output
# 1: 3.0
# 2: 5.125
# 3: 8.0
# 5: 4.0
```

```python
'''
Daniel Naylor
5094024
11/09/2022
'''

import numpy as np
import matplotlib.pyplot as plt
from tabulate import tabulate as tab
from typing import List, Tuple

# --------------------------------------------------
#                       Problem 6
# --------------------------------------------------

class NaturalSplineInterpolation:
    '''
    A natural spline interpolation (piecewise cubic polynomial)
    of a given dataset.

    This class acts like a mathematical function,
    i.e. it can be called with a number as input
    and gives a number as output.
    '''
    def __init__(self, data: List[Tuple[float, float]]) -> None:
        '''
        Initializes a spline interpolation based on the given data.
        Data should be of the form:
        ```
        [
            (x_0, f_0),
            (x_1, f_1),
            ...
            (x_n, f_n)
        ]
        ```
        Each `x` value should be sorted. For example, `[(0, 1), (-1, 0)]` is an invalid data
input,
        and instead should be `[(-1, 0), (0, 1)]`.
        '''
        n = len(data) - 1
        assert n > 0, 'Please provide a valid dataset (at least 2 points)'
        self._data = data

        z = { # natural spline, set z_0 and z_n first
            0: 0,
            n: 0
        }
        h = [ # compute values for h_j, length of each sub-interval.
            data[j+1][0] - data[j][0]
            for j in range(n) # 0, ..., n-1
        ]

        # solve for l, m values
        # use a dictionary for ease of assigning values
        l={} # using 1, ..., n-2
        m={1: 2 * (h[0] + h[1])} # 1, ..., n-1
        for j in range(1, n-1): # 1, ..., n-2
            l[j] = h[j]/m[j]
            m[j+1] = 2 * (h[j] + h[j+1]) - (l[j] * h[j])

        # compute values of y (what the sums of z add to)
        y={
            j: -6 * (data[j][1] - data[j-1][1])/(h[j-1]) + 6 * (data[j+1][1] - data[j]
[1])/h[j]
            for j in range(1, n) # 1, ..., n-1
        }
```

```
        # compute the values of k (what the matrix product of right-upper and z's produce)
        k={1: y[1]}
        for j in range(2, n):
            k[j] = y[j] - l[j-1] * k[j-1]

        # finally, compute the remaining z values using k
        # note: this abuses the lower triangular property
        z[n-1] = k[n-1]/m[n-1]
        for j in range(n-2, 0, -1): # n-2, ..., 1
            z[j] = (k[j] - h[j] * z[j+1])/m[j]

        # FINALLY, compute a, b, c, d coefficients
        a, b, c, d = {}, {}, {}, {}

        for j in range(n):
            a[j] = (z[j+1] - z[j]) / (6 * h[j])
            b[j] = z[j]/2
            c[j] = (data[j+1][1] - data[j][1])/h[j] - h[j]/6 * (z[j+1] + 2 * z[j])
            d[j] = data[j][1]

        self._a, self._b, self._c, self._d = a, b, c, d

    def __call__(self, x: float, /) -> float:
        # determine suitable piecewise spline S_j using the following heuristic:
        # find maximum j such that x <= x_(j+1).
        # if no such j (for whatever reason), then use j=n-1.
        if (x >= self._data[-2][0]):
            j=len(self._data) - 2
        else:
            j=0
            while (x > self._data[j+1][0]):
                j=j+1

        shifted_x = x - self._data[j][0]

        return self._a[j] * shifted_x**3 + self._b[j] * shifted_x**2 + self._c[j] * shifted_x
+ self._d[j]

# test data

x_data = [
    (0, 1.5),
    (0.618, 0.90),
    (0.935, 0.60),
    (1.255, 0.35),
    (1.636, 0.20),
    (1.905, 0.10),
    (2.317, 0.50),
    (2.827, 1.00),
    (3.330, 1.50)
]

y_data = [
    (0, 0.75),
    (0.618, 0.90),
    (0.935, 1.00),
    (1.255, 0.80),
    (1.636, 0.45),
    (1.905, 0.20),
    (2.317, 0.10),
    (2.827, 0.20),
    (3.330, 0.25)
]

x_approx = NaturalSplineInterpolation(data=x_data)
y_approx = NaturalSplineInterpolation(data=y_data)
```

```python
x_coeffs_table = {
    'j': [x for x in range(8)],
    'a_j': [*x_approx._a.values()],
    'b_j': [*x_approx._b.values()],
    'c_j': [*x_approx._c.values()],
    'd_j': [*x_approx._d.values()]
}

print(tab(x_coeffs_table, headers='keys'))

#   j         a_j         b_j        c_j    d_j
# ---  ----------  ----------  ---------  -----
#   0   0.0105369   0           -0.974898   1.5
#   1   0.102103    0.0195355  -0.962825   0.9
#   2   0.987167    0.116635   -0.919659   0.6
#   3  -1.77355     1.06432    -0.541755   0.35
#   4   5.39457    -0.962851   -0.503097   0.2
#   5  -3.39333     3.39057     0.149959   0.1
#   6   0.670643   -0.803594    1.21579    0.5
#   7  -0.147442    0.22249     0.919428   1

y_coeffs_table = {
    'j': [y for y in range(8)],
    'a_j': [*y_approx._a.values()],
    'b_j': [*y_approx._b.values()],
    'c_j': [*y_approx._c.values()],
    'd_j': [*y_approx._d.values()]
}

print(tab(y_coeffs_table, headers='keys'))

#   j        a_j          b_j         c_j    d_j
# ---  ---------  -----------  ----------  -----
#   0   0.276944   0             0.136947    0.75
#   1  -3.00101    0.513454      0.454261    0.9
#   2   2.43045   -2.34051     -0.124915    1
#   3  -0.273187  -0.00727664  -0.876207    0.8
#   4   2.1739    -0.31953     -1.00072     0.45
#   5  -0.784397   1.4348      -0.700711    0.2
#   6  -0.474226   0.465289     0.0821273   0.1
#   7   0.172483  -0.260277     0.186683    0.2

# graph the parametric function...

t = np.linspace(0, 3.33, 400)

x = np.array([
    x_approx(i)
    for i in t
])

y = np.array([
   y_approx(i)
   for i in t
])

fig, ax = plt.subplots()

ax.plot(x, y)

plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.2
0.4
0.6
0.8
1.0
1.2
1.4