104A: Homework 2

Daniel Naylor

Question 1

We know that

$$P_n(x) = \sum_{j=0}^{n} \left(f_j \prod_{k \neq j} \frac{x - x_k}{x_j - x_k} \right)$$

So in our case,

$$\begin{aligned} P_2(x) &= 1 \cdot \left(\frac{x-1}{-1} \cdot \frac{x-3}{-3} \right) + \cdot 1 \left(x \cdot \frac{x-3}{1-3} \right) + -5 \cdot \left(\frac{x}{3} \cdot \frac{x-1}{3-1} \right) \\ &= (1-x) \cdot \frac{3-x}{3} + \frac{3x - x^2}{2} + \frac{5x - 5x^2}{6} \\ &= -x^2 + x + 1 \end{aligned}$$

And we find that $P_2(2) = -1 \approx f(2)$

Question 3 Comments

Interestingly, the cosine grid produces a far superior approximation than the normal grid.

Moreover with (c), it's interesting how the polynomial interpolations become very strong approximations for relatively small polynomials, unlike with (a).

Question 4

(a)

Suppose $f \in C^2[x_0, x_1]$, and let P_1 be the 1st degree polynomial interpolation for f at x_0, x_1. Then

$$||f - P_1||_\infty \leq \frac{1}{8}(x_0 - x_1)^2 M_2$$

Where $|f''(x)| \leq M_2 \; \forall \, x \in [x_0, x_1]$

Proof

Let $R(x) = f(x) - P_1(x)$ be the error term. Now fix $x \neq x_0, x_1$ and define

$$\psi(t) = R(t) - \frac{R(x)}{\omega(x)} \cdot \omega(t)$$

Where $\omega(t) = (t - x_0)(t - x_1)$.

Since $f(x_j) = P_1(x_j)$ and $\omega(x_j) = 0$, it follows that $\psi(x_j) = 0$ for $j = 0, 1$. Moreover, $\psi(x) = 0$. Hence ψ has 3 roots.

Since $\psi(t)$ has 3 roots, by Rolle's Theorem (applied twice) there exists two points such that $\psi'(c_1) = \psi'(c_2) = 0$. Applying Rolle's Theorem again we find that $\psi''(c) = 0$ for $c \in [x_0, x_1]$.

Yet we also find (note: ω is a 2-degree polynomial)

$$\psi''(t) = R''(t) - \frac{R(x)}{\omega(x)} \cdot 2!$$

Since $R(x) = f(x) - P_1(x)$, $R''(x) = f''(x)$. Hence

$$\psi''(t) = f''(t) - \frac{R(x)}{\omega(x)} \cdot 2$$

Plugging in c yields

$$\psi''(c) = 0 = f''(c) - \frac{R(x)}{\omega(x)} \implies f(x) - P_1(x) = \frac{f''(c)}{2} \cdot (x - x_0)(x - x_1)$$

We may take absolute values as they don't change equality

$$|f(x) - P_1(x)| = \frac{|f''(c)|}{2} \cdot |x - x_0| \, |x - x_1|$$

Note that we can find a number M_2 such that $|f''(x)| \leq M_2$ by the Extreme Value Theorem (since x is bounded). Moreover, since $x \in [x_0, x_1]$,

$$|x - x_0||x - x_1| \leq \frac{|x_1 - x_0|}{2} \cdot \frac{|x_0 - x_1|}{2} = \frac{(x_0 - x_1)^2}{4}$$

Hence

$$|f(x) - P_1(x)| \leq \frac{(x_0 - x_1)^2}{8} \cdot M_2$$

Since the RHS doesn't depend on x we may take the supremum

$$\sup_{[x_0,x_1]} |f(x) - P_1(x)| = ||f - P_1||_\infty \leq \frac{(x_0 - x_1)^2}{8} \cdot M_2$$

(b)

By the above result, we find that the P_1 maximum error for $\sin x$ is

$$||\sin x - P_1(x)||_\infty \leq \frac{(\pi/2)^2}{8} \cdot 1 = \frac{\pi^2}{32} \approx 0.30842513753$$

We know that $\sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}$, and that

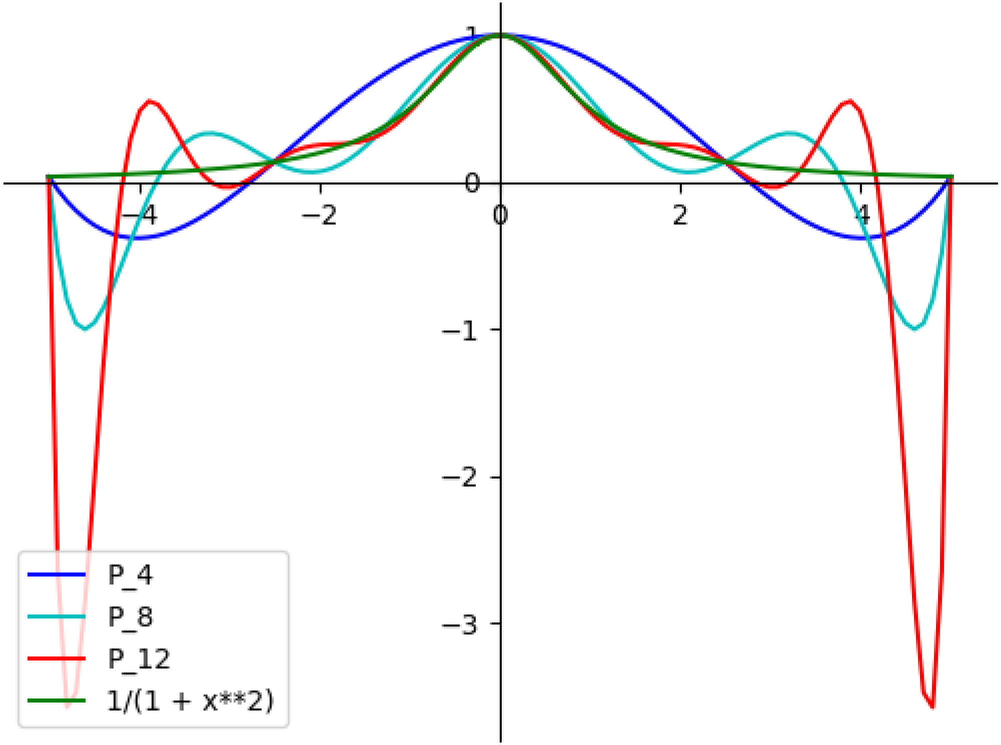
$$P_1(x) = f_0 + \frac{f_0 - f_1}{x_0 - x_1}(x - x_0) = 0 + \frac{0 - 1}{0 - \pi/2} x = \frac{2}{\pi} \cdot x$$

So $P_1\left(\frac{\pi}{4}\right) = \frac{1}{2}$ and the error is $\frac{\sqrt{2}-1}{2} \approx 0.20710678118$.

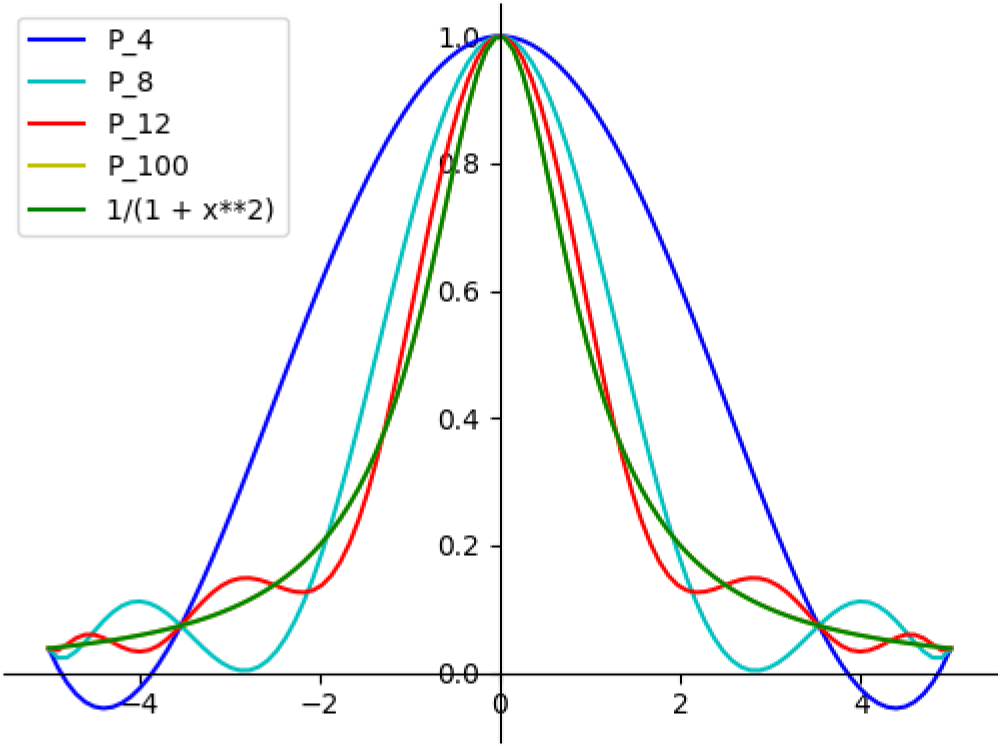
```python
'''
Daniel Naylor
5094024
10/20/2022
'''

import math
from typing import Tuple, List, Callable


stored_lambdas: dict[Tuple[int, int], float] = {
     # to allow for efficient recursion
     # entries are of the form
     # (n, j): lambda value
     (0, 0): 1.0
}

data_points: List[Tuple[float, float]] = [
    # all relevant data, stored in the form
    # j: (x_j, f_j)

    (0.00, 0.0000),
    (0.25, 0.7071),
    (0.50, 1.0000),
    (0.75, 0.7071),
    (1.25, -0.7071),
    (1.50, -1.0000)
]

def compute_lambda_direct(*, n: int, j: int, data: List[Tuple[float, float]] = data_points) ->
float:
    '''
    Computes a Barycentric weight (lambda) directly.
    1/product(xj - xk) for j != k

    `data` is the dataset that these lambdas will be computed from.
    '''
    prod = 1.0
    for k in range(n+1): # 0, 1, ..., n
        if k==j: continue
        prod *= (data[j][0] - data[k][0])

    return 1.0/prod

def compute_lambda(*, n: int, j: int, data: List[Tuple[float, float]] = data_points) -> float:
    '''
    Computes a Barycentric weight (lambda) recursively-ish.
    This uses `stored_lambdas` (dynamic programming) for efficiency purposes.
    '''
    if (n, j) in stored_lambdas:
        return stored_lambdas[(n, j)]
    elif (n - 1, j) in stored_lambdas:
        stored_lambdas[(n, j)] = stored_lambdas[(n-1, j)]/(data[j][0] - data[n][0])
        return stored_lambdas[(n, j)]
    else:
        stored_lambdas[(n, j)] = compute_lambda_direct(n=n, j=j, data=data)
        return stored_lambdas[(n, j)]

def create_poly_interp(data: List[Tuple[float, float]] = data_points) -> Callable[[float],
float]:
    '''
    Returns a polynomial interpolation based on the
    given dataset using Barycentric Weights.

    The returned callable will act as a mathematical function
    that is defined everywhere except at the datapoints themselves.
    '''
    def wrap(x: float):
        top_sum = 0.0
        bottom_sum = 0.0
        n = len(data) - 1
        for j in range(n + 1):
            current_lambda = compute_lambda(n=n, j=j, data=data)
            top_sum += (current_lambda * data[j][1])/(x - data[j][0])
            bottom_sum += current_lambda/(x - data[j][0])

        return top_sum/bottom_sum

    return wrap

P_5 = create_poly_interp(data=data_points)

print(P_5(2))

# Output: 0.8519999999999989
```

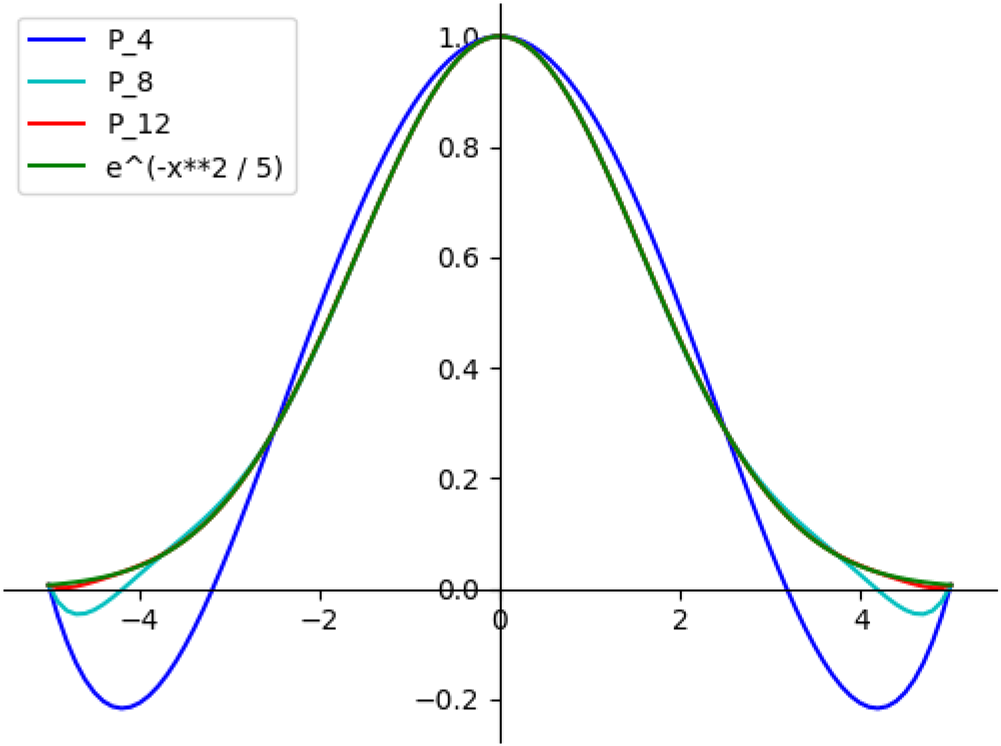
```python
'''
Daniel Naylor
5094024
10/20/2022
'''

import math
import matplotlib.pyplot as plt
import numpy as np

from typing import Tuple, List, Callable

def f(x: float) -> float:
    '''
    Our function f for parts a and b
    '''
    return 1/(1 + x**2)

def uniform_lambda(*, n: int, j: int) -> int:
    '''
    Computes a lambda assuming that the grid is uniform (all points are equally spaced).

    `(-1)^j n choose j`
    '''
    return (-1)**j * math.comb(n, j)

def cos_lambda(*, n: int, j: int) -> float:
    '''
    Computes a lambda assuming it's of the cosine form shown in the homework question 3b 
remark.
    '''
    scalar = 0.5 if j==0 or j==n else 1
    return scalar * (-1)**j

def create_poly_interp(
        data: List[Tuple[float, float]],
        *,
        lambda_formula: Callable[[int, int], int | float]
    ) -> Callable[[float], float]:
    '''
    Returns a polynomial interpolation based on the
    given dataset using Barycentric Weights.

    `lambda_formula` refers to the formula (function) that
    should be used to compute lambda values.

    The returned callable will act as a mathematical function
    that is defined everywhere except at the datapoints themselves.
    '''
    def wrap(x: float):
        top_sum = 0.0
        bottom_sum = 0.0
        n = len(data) - 1
        for j in range(n + 1):
            # if trying to compute over a given node point, return its value
            if x == data[j][0]: return data[j][1]
            current_lambda = lambda_formula(n=n, j=j)
            top_sum += (current_lambda * data[j][1])/(x - data[j][0])
            bottom_sum += current_lambda/(x - data[j][0])

        return top_sum/bottom_sum

    return wrap


# --------------------------------------------
#                     Part A
# ---------------------------------------------

# all datasets are lists of the form
# j: (xj, fj)
# where j is the jth element in the list

def create_dataset_for_a(*, n: int, func: Callable[[float], float]) -> List[Tuple[float,
float]]:
    '''
    Creates a suitable dataset of n + 1 (nodes)
    for question 3a depending on the given n
    and function
    '''
    return [
        (
            -5 + j * (10/n),
            func(-5 + j * (10/n))
        )
        for j in range(n + 1) # j = 0, ..., n
    ]

P_a1 = create_poly_interp( # polynomial approximation for 3a n=4
    data=create_dataset_for_a(n=4, func=f),
    lambda_formula=uniform_lambda
)

P_a2 = create_poly_interp(
    data=create_dataset_for_a(n=8, func=f),
    lambda_formula=uniform_lambda
)

P_a3 = create_poly_interp(
    data=create_dataset_for_a(n=12, func=f),
    lambda_formula=uniform_lambda
)

# ---------------------------------------------
#                    Part B
# ---------------------------------------------

def create_cos_nodes(*, n: int, func: Callable[[float], float]) -> List[Tuple[float, float]]:
    '''
    Creates nodes using the cosine formula given in part b
    '''
    return [
        (
            5 * math.cos(math.pi * j / n),
            func(5 * math.cos(math.pi * j / n))
        )
        for j in range(n+1)
    ]

P_b1 = create_poly_interp(
    data=create_cos_nodes(n=4, func=f),
    lambda_formula=cos_lambda
)

P_b2 = create_poly_interp(
    data=create_cos_nodes(n=8, func=f),
    lambda_formula=cos_lambda
)

P_b3 = create_poly_interp(
    data=create_cos_nodes(n=12, func=f),
    lambda_formula=cos_lambda
)
```

```python
P_b4 = create_poly_interp(
    data=create_cos_nodes(n=100, func=f),
    lambda_formula=cos_lambda
)

# --------------------------------------------
#                    Part C
# --------------------------------------------

def f_c(x: float) -> float:
    '''
    Our function f for part c
    '''
    return math.exp(-x**2 / 5)


P_c1 = create_poly_interp( # polynomial approximation for 3a n=4
    data=create_dataset_for_a(n=4, func=f_c),
    lambda_formula=uniform_lambda
)

P_c2 = create_poly_interp(
    data=create_dataset_for_a(n=8, func=f_c),
    lambda_formula=uniform_lambda
)

P_c3 = create_poly_interp(
    data=create_dataset_for_a(n=12, func=f_c),
    lambda_formula=uniform_lambda
)

# ---------------------------------------------------
#             graphing all the functions
#    (i only included the code for graphing part c)
#  (the code for parts a and b are almost identical)
#        (all graphs are attached to this PDF)
# ---------------------------------------------------

domain = np.linspace(-5, 5, 100)

graph_of_P_c1 = np.array([
    P_c1(x)
    for x in domain
])

graph_of_P_c2 = np.array([
    P_c2(x)
    for x in domain
])

graph_of_P_c3 = np.array([
    P_c3(x)
    for x in domain
])

graph_of_f_c = np.array([
    f_c(x)
    for x in domain
])

# setting up the graphs...
fig = plt.figure()

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

# plot the functions
plt.plot(domain, graph_of_P_c1, 'b', label='P_4')
plt.plot(domain, graph_of_P_c2, 'c', label='P_8')
plt.plot(domain, graph_of_P_c3, 'r', label='P_12')
plt.plot(domain, graph_of_f_c, 'g', label='e^(-x**2 / 5)')

plt.legend(loc='upper left')

# show the plot
plt.show()
```



1
0
−1
−2
−3
−4
−2
0
2
4
P_4
P_8
P_12
1/(1 + x**2)


P_4
P_8
P_12
P_100
1/(1 + x**2)
1.0
0.8
0.6
0.4
0.2
0.0
−4
−2
0
2
4


P_4
P_8
P_12
e^(-x**2 / 5)
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−4
−2
0
2
4