Homework 2 writeups

Name: Oorjit Chowdhary

Section: AMATH 301 B

Problem 1

In [ ]:
```
import numpy as np

# Coding Problem 2
## Part a
y1, y2, y3, y4 = 0, 0, 0, 0
term1, term2, term3, term4 = 0.1, 0.1, 0.25, 0.5

for k in range(100000):
    y1 += term1

for k in range(100000000):
    y2 += term2

for k in range(100000000):
    y3 += term3

for k in range(100000000):
    y4 += term4

## Part b
x1 = np.abs(10000 - y1)
x2 = np.abs(y2 - 10000000)
x3 = np.abs(25000000 - y3)
x4 = np.abs(y4 - 50000000)

print(x1, x2, x3, x4)
```

```
1.8848368199542165e-08 0.018870549276471138 0.0 0.0
```

Part (a)

In Problem 2 of the coding portion of the homework, I found the following values for x_1, x_2, x_3, and x_4.

x_1	x_2	x_3	x_4
1.8848368199542165e-08	0.018870549276471138	0.0	0.0

Ranking from smallest to largest, we get:

$$x_3 = x_4 < x_1 < x_2$$

Part (b)

The values for x_1 and x_2 show that adding the small truncation error, which comes from the finite memory space allocation to store floating point numbers, for multiple times increases the amount of the error. Both `term1` and `term2` for `y1` and `y2` = 0.1, but `x2` > `x1` because `term2` was added 10,000,000 times and `term` was added 10,000 times.

But these values also contain two outliers, x_3 and x_4, as they are exactly zero while they were also derived from floating point numbers algebra. I think the reason for this the choice of numbers used for `term3` and `term4` as they can be considered as multiples of 2, which makes it easier for the compiler.

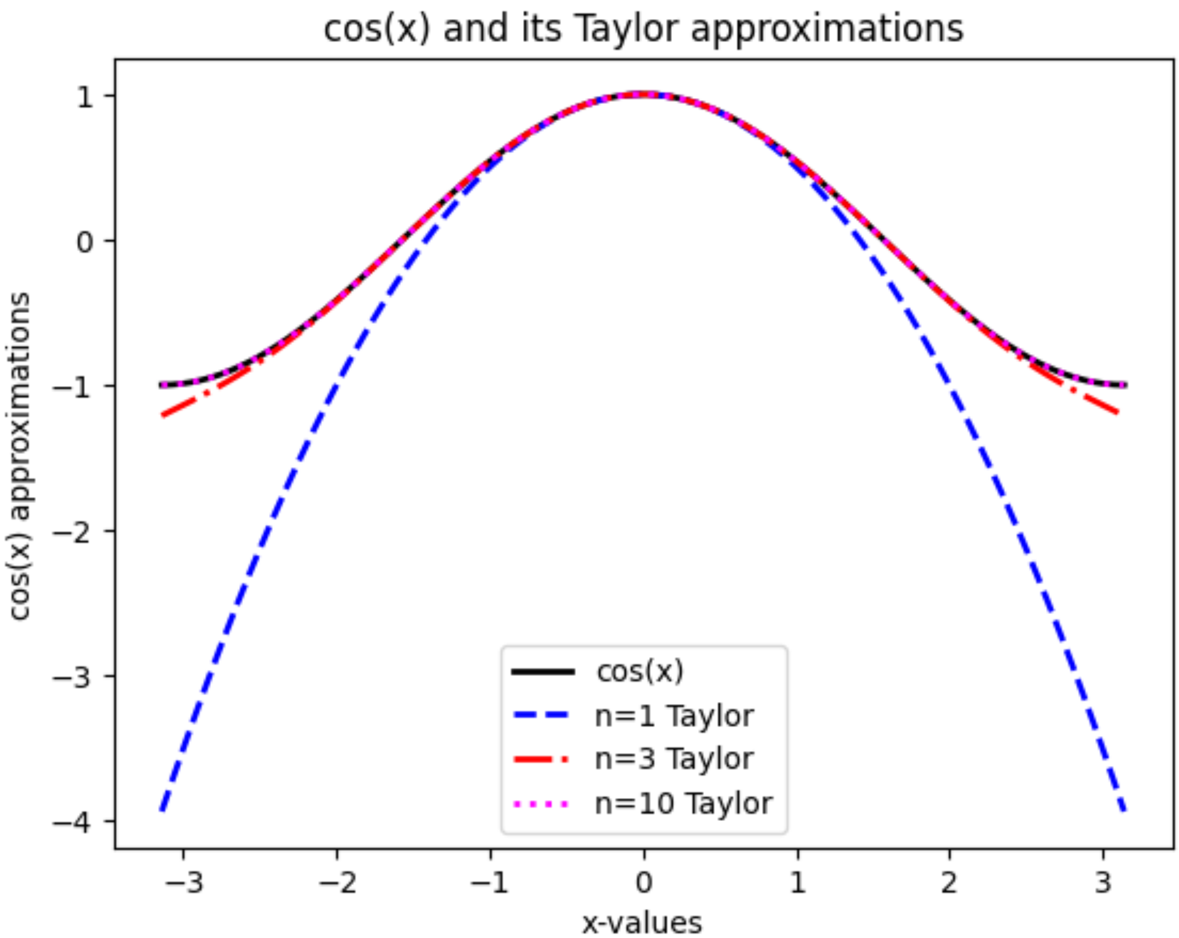
Part (c)

Going ahead with the information given in the hint, `term3 = 0.25` can be represented as 2^{-2} and `term4 = 0.5` can be represented as 2^{-1}. As the numbers are perfectly contained in base 2 notation, there will **NOT** be any truncation error. Hence x_3 and x_4 are exactly zero.

Problem 2

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-np.pi, np.pi, 100)
y = np.cos(x)
taylor_n1, taylor_n3, taylor_n10 = 0 * x, 0 * x, 0 * x

for k in range(2):
    taylor_n1 += (-1) ** k * x ** (2 * k) / np.math.factorial(2 * k)

for k in range(4):
    taylor_n3 += (-1) ** k * x ** (2 * k) / np.math.factorial(2 * k)

for k in range(11):
    taylor_n10 += (-1) ** k * x ** (2 * k) / np.math.factorial(2 * k)

plt.plot(x, y, color='k', linewidth=2)
plt.plot(x, taylor_n1, color='b', linewidth=2, linestyle='--')
plt.plot(x, taylor_n3, color='r', linewidth=2, linestyle='-.')
plt.plot(x, taylor_n10, color='magenta', linewidth=2, linestyle=':')
plt.legend(['cos(x)', 'n=1 Taylor', 'n=3 Taylor', 'n=10 Taylor'])

plt.xlabel('x-values')
plt.ylabel('cos(x) approximations')
plt.title('cos(x) and its Taylor approximations')
```

Out[ ]:
```
Text(0.5, 1.0, 'cos(x) and its Taylor approximations')
```


cos(x) and its Taylor approximations
cos(x) approximations
x-values
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3
cos(x)
n=1 Taylor
n=3 Taylor
n=10 Taylor