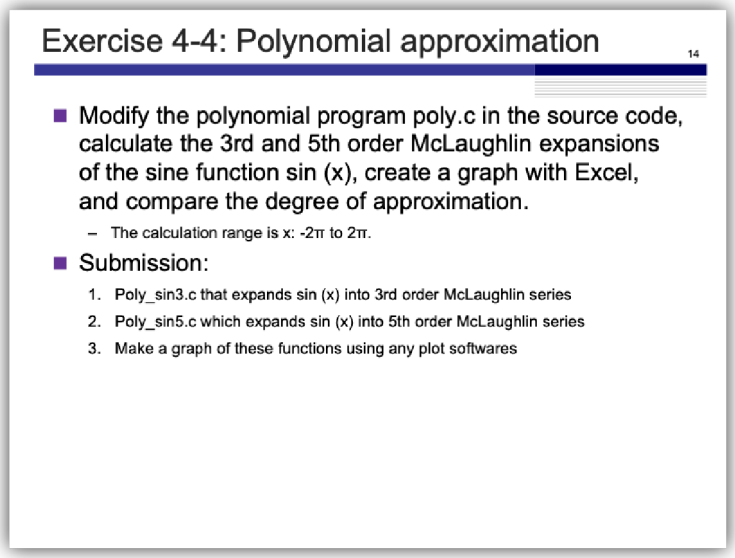
Exercise 4-4

Maximilian Fernaldy - C2B1702



To evaluate the Maclaurin expansions, we first need to define it. The Maclaurin series is an approximation of a function that can be defined as follows:

$$f(x) = \frac{f(0)}{0!} + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \ldots$$

$$\therefore\ f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!}x^n$$

Since we're going to use factorials, we should also define the factorial as a function, as it's not included in the `<math.h>` header file:

```
int factorial(int number) {
    if (number < 1) { // if zero is passed, return 1 as the factorial
        return 1;
    }
    for (int i = number-1; i > 1; i--) {
        number *= i; // multiplies number by i and assigns that new value to number
    }
    return number;
}
```

Note that this function does not error handle factorials of integers less than 0 (it still returns 1 as the factorial, even though factorials of negative integers are undefined), but since we are going to use it four our own purposes, it's not a problem.

Using the newly-defined function `factorial`, we can define the coefficients to use in the Maclaurin series.

In poly_sin3.c:

```
double  c[N] = {sin(0)/factorial(0), cos(0)/factorial(1), -sin(0)/factorial(2), -cos(0)/factorial(3)};
```

and in poly_sin5.c:

```
double  c[N] = {sin(0)/factorial(0), // evaluates to 0
                cos(0)/factorial(1), // evaluates to 1
                -sin(0)/factorial(2), // evaluates to 0
                -cos(0)/factorial(3), // evaluates to 1/6
                sin(0)/factorial(4), // evaluates to 1/24
                cos(0)/factorial(5)}; // evaluates to 1/120
```

Even though it is possible to just use the evaluated values of the coefficients, it is good practice to define where the coefficients come from, for better code readability and comprehension.

After setting the coefficients, we use the following formula for the Maclaurin series of degree n:

$$f(x) = (((c_{n-1}x + c_{n-2})x + c_{n-3})x + ... + c_1)x + c_0$$

translating this formula to code, we have

```
/**** (2) processing contents ****/
for(x = -M_PI; x <= M_PI; x += M_PI/100){
    for(y = c[N-1], i = N-2; i >= 0; --i) {
        y = y * x + c[i];
    }
    printf("%f %f\n", x, y);
}
```

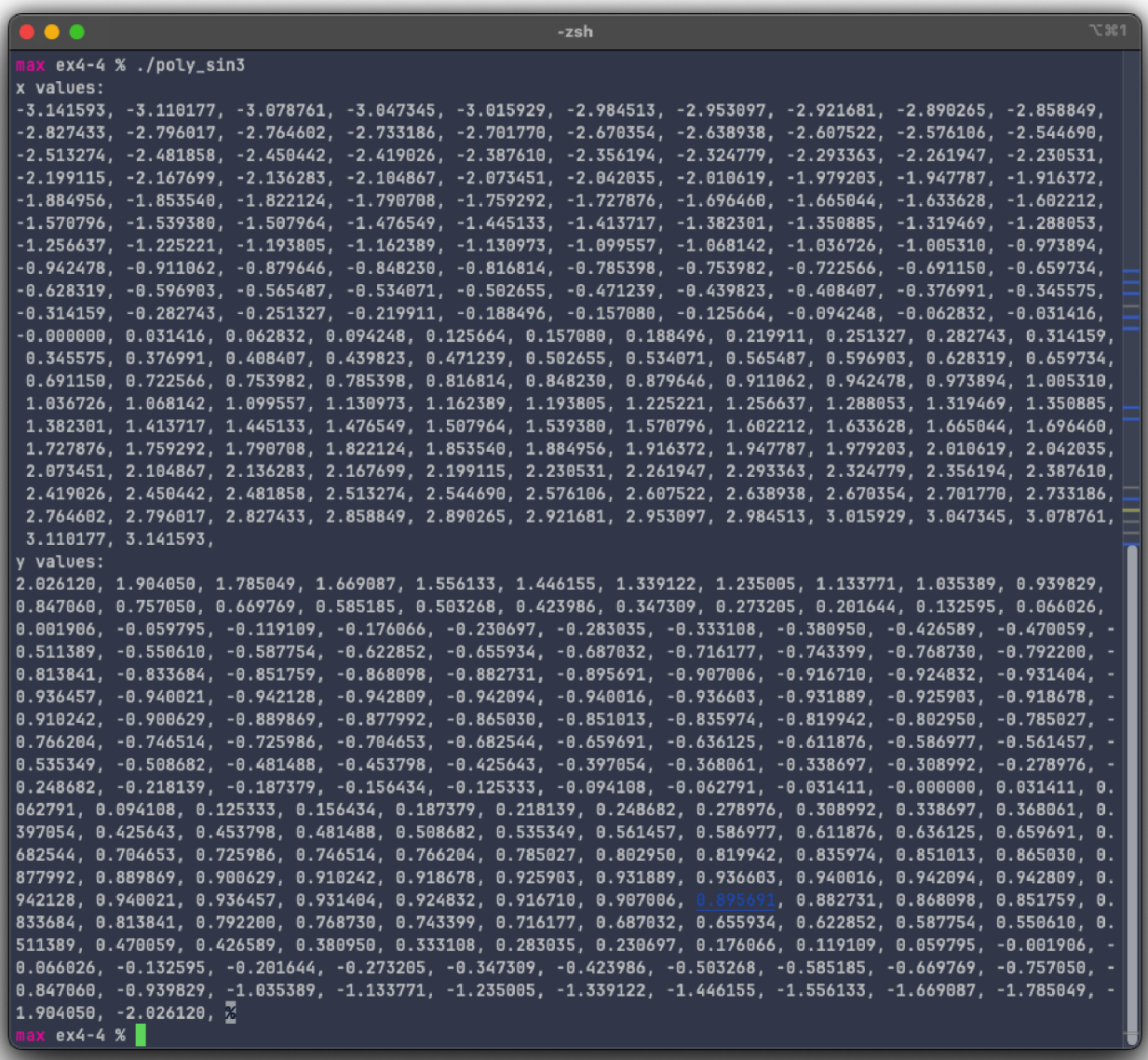
the `x` level `for` loop iterates through the linear space of the x-axis, giving the function an x-value to work with. The `y` level `for` loop is being used to sum the terms of the formula. We can see that at the first iteration, `y = c[N-1]` which corresponds to c_{n-1} gets multiplied by `x` and added by `c[i]`, which corresponds to c_{n-2}. Then the whole of this corresponds to $c_{n-1}x + c_{n-2}$, and it all gets multiplied by x and added by `c[i]` again in the next iteration, but now `c[i]` corresponds to c_{n-3}. The operation will repeat itself, changing `c[i]` until it is `c[0]`, before printing the result for the specific value of x, and continuing with the next value of x. This will happen until the `x` level `for` loop also ends, ending the whole program.

Since we want to plot the values using matplotlib later, we should modify the program to print out the x and y values separated by a comma and space.

```
/**** (2) processing contents ****/
printf("x values:\n");
for(x = -M_PI; x <= M_PI; x += M_PI/100){
    printf("%f, ", x);
}
printf("\ny values:\n");
for(x = -M_PI; x <= M_PI; x += M_PI/100){
    for(y = c[N-1], i = N-2; i >= 0; --i) {
        y = y * x + c[i];
    }
    printf("%f, ", y);
}
```

The first `for` loop iterates from $-\pi$ to π in $\frac{\pi}{100}$ increments and prints out the x-values. The second calculates the y-values and prints them out. This part of the code stays the same for whichever degree of polynomial we are using.

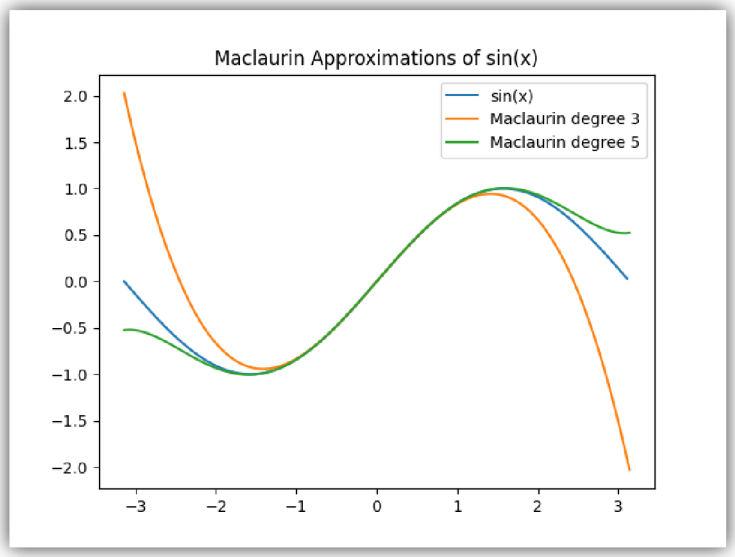
Compiling and running poly_sin3.c gives:



and poly_sin5.c gives:

```
-zsh
max ex4-4 % ./poly_sin5
x values:
-3.141593, -3.110177, -3.078761, -3.047345, -3.015929, -2.984513, -2.953097, -2.921681, -2.890265, -2.858849,
-2.827433, -2.796017, -2.764602, -2.733186, -2.701770, -2.670354, -2.638938, -2.607522, -2.576106, -2.544690,
-2.513274, -2.481858, -2.450442, -2.419026, -2.387610, -2.356194, -2.324779, -2.293363, -2.261947, -2.230531,
-2.199115, -2.167699, -2.136283, -2.104867, -2.073451, -2.042035, -2.010619, -1.979203, -1.947787, -1.916372,
-1.884956, -1.853540, -1.822124, -1.790708, -1.759292, -1.727876, -1.696460, -1.665044, -1.633628, -1.602212,
-1.570796, -1.539380, -1.507964, -1.476549, -1.445133, -1.413717, -1.382301, -1.350885, -1.319469, -1.288053,
-1.256637, -1.225221, -1.193805, -1.162389, -1.130973, -1.099557, -1.068142, -1.036726, -1.005310, -0.973894,
-0.942478, -0.911062, -0.879646, -0.848230, -0.816814, -0.785398, -0.753982, -0.722566, -0.691150, -0.659734,
-0.628319, -0.596903, -0.565487, -0.534071, -0.502655, -0.471239, -0.439823, -0.408407, -0.376991, -0.345575,
-0.314159, -0.282743, -0.251327, -0.219911, -0.188496, -0.157080, -0.125664, -0.094248, -0.062832, -0.031416,
-0.000000, 0.031416, 0.062832, 0.094248, 0.125664, 0.157080, 0.188496, 0.219911, 0.251327, 0.282743, 0.314159,
 0.345575, 0.376991, 0.408407, 0.439823, 0.471239, 0.502655, 0.534071, 0.565487, 0.596903, 0.628319, 0.659734,
 0.691150, 0.722566, 0.753982, 0.785398, 0.816814, 0.848230, 0.879646, 0.911062, 0.942478, 0.973894, 1.005310,
 1.036726, 1.068142, 1.099557, 1.130973, 1.162389, 1.193805, 1.225221, 1.256637, 1.288053, 1.319469, 1.350885,
 1.382301, 1.413717, 1.445133, 1.476549, 1.507964, 1.539380, 1.570796, 1.602212, 1.633628, 1.665044, 1.696460,
 1.727876, 1.759292, 1.790708, 1.822124, 1.853540, 1.884956, 1.916372, 1.947787, 1.979203, 2.010619, 2.042035,
 2.073451, 2.104867, 2.136283, 2.167699, 2.199115, 2.230531, 2.261947, 2.293363, 2.324779, 2.356194, 2.387610,
 2.419026, 2.450442, 2.481858, 2.513274, 2.544690, 2.576106, 2.607522, 2.638938, 2.670354, 2.701770, 2.733186,
 2.764602, 2.796017, 2.827433, 2.858849, 2.890265, 2.921681, 2.953097, 2.984513, 3.015929, 3.047345, 3.078761,
 3.110177, 3.141593,
y values:
-0.524044, -0.521131, -0.520097, -0.520826, -0.523202, -0.527114, -0.532453, -0.539115, -0.546995, -0.555995,
-0.566017, -0.576967, -0.588753, -0.601286, -0.614481, -0.628253, -0.642523, -0.657211, -0.672242, -0.687543,
-0.703043, -0.718674, -0.734371, -0.750069, -0.765709, -0.781232, -0.796580, -0.811702, -0.826544, -0.841058,
-0.855195, -0.868912, -0.882166, -0.894914, -0.907120, -0.918745, -0.929756, -0.940120, -0.949806, -0.958785,
-0.967030, -0.974517, -0.981223, -0.987125, -0.992204, -0.996444, -0.999826, -1.002337, -1.003965, -1.004697,
-1.004525, -1.003440, -1.001436, -0.998508, -0.994652, -0.989867, -0.984151, -0.977505, -0.969932, -0.961434,
-0.952017, -0.941686, -0.930449, -0.918313, -0.905288, -0.891386, -0.876616, -0.860994, -0.844531, -0.827243,
-0.809146, -0.790257, -0.770593, -0.750173, -0.729016, -0.707143, -0.684574, -0.661332, -0.637439, -0.612918,
-0.587793, -0.562089, -0.535830, -0.509044, -0.481755, -0.453992, -0.425780, -0.397148, -0.368125, -0.338738,
-0.309017, -0.278991, -0.248690, -0.218143, -0.187381, -0.156434, -0.125333, -0.094108, -0.062791, -0.031411,
-0.000000, 0.031411, 0.062791, 0.094108, 0.125333, 0.156434, 0.187381, 0.218143, 0.248690, 0.278991, 0.309017,
 0.338738, 0.368125, 0.397148, 0.425780, 0.453992, 0.481755, 0.509044, 0.535830, 0.562089, 0.587793, 0.612918,
 0.637439, 0.661332, 0.684574, 0.707143, 0.729016, 0.750173, 0.770593, 0.790257, 0.809146, 0.827243, 0.844531,
 0.860994, 0.876616, 0.891386, 0.905288, 0.918313, 0.930449, 0.941686, 0.952017, 0.961434, 0.969932, 0.977505,
 0.984151, 0.989867, 0.994652, 0.998508, 1.001436, 1.003440, 1.004525, 1.004697, 1.003965, 1.002337, 0.999826,
 0.996444, 0.992204, 0.987125, 0.981223, 0.974517, 0.967030, 0.958785, 0.949806, 0.940120, 0.929756, 0.918745,
 0.907120, 0.894914, 0.882166, 0.868912, 0.855195, 0.841058, 0.826544, 0.811702, 0.796580, 0.781232, 0.765709,
 0.750069, 0.734371, 0.718674, 0.703043, 0.687543, 0.672242, 0.657211, 0.642523, 0.628253, 0.614481, 0.601286,
 0.588753, 0.576967, 0.566017, 0.555995, 0.546995, 0.539115, 0.532453, 0.527114, 0.523202, 0.520826, 0.520097,
 0.521131, 0.524044,
max ex4-4 %
```

We can use the data from the output and plug it into an array in python (see last page for python code) to plot it and see how close they are to the actual sin function.



Clearly, the 5th degree Maclaurin series approximates $sin(x)$ better than the 3rd degree series, as it stays close to the real function further from $x = 0$. It's still not a viable option for x-values higher than π or lower than $-\pi$ though, as we can see that it deviates quite severely after that point.

Python code for plotting

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import binom

# Actual sin function
lowerlimit = -np.pi; upperlimit = np.pi

x_values = list(np.arange(lowerlimit, upperlimit, np.pi/100))

print(x_values)

y_values = [np.sin(x) for x in x_values]

# Data obtained from poly_sin3
c_program_x_range=[-3.141593, -3.110177, -3.078761, -3.047345, -3.015929, -2.984513, ...]

sin3data=[2.026120, 1.904050, 1.785049, 1.669087, 1.556133, 1.446155, ...]

# Data obtained from poly_sin5
sin5data=[-0.524044, -0.521131, -0.520097, -0.520826, -0.523202, -0.527114, ...]


plt.plot(x_values, y_values, label = "sin(x)")
plt.plot(c_program_x_range, sin3data, label = "Maclaurin degree 3")
plt.plot(c_program_x_range, sin5data, label = "Maclaurin degree 5")
plt.title("Maclaurin Approximations of sin(x)")
plt.legend()
plt.show()
```