Assignment

Given Algorithm :

```
function x = f(n) :
   x = 1;
   for i = 1:
         for j = 1:n :
              x = x + 1;
```

1. Find the runtime of the algorithm mathematically (I should see summations).

- The given algorithm involves two nested loops iterating from 1 to n,performing a constant time operation inside the loop.
- Each loop will be denoted as $\sum_{\square}^{\square=1} \blacksquare$ in summations from 1 to n.
- And the value statement inside the loops is 1
- The time complexity can be expressed mathematically as:

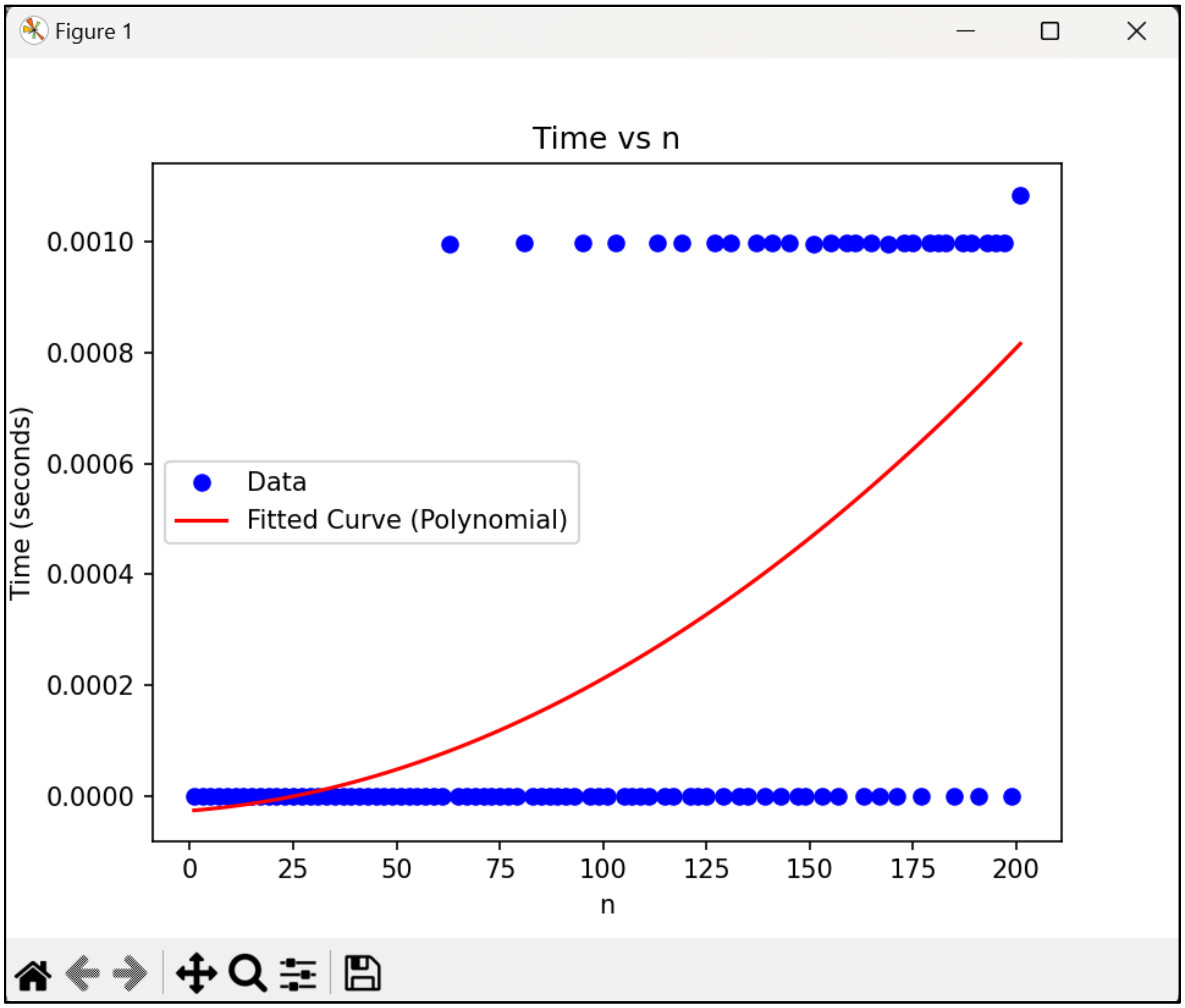
$$T(n)_{\blacksquare} = \sum_{n}^{i=1} \blacksquare \sum_{n}^{j=1} \blacksquare\, 1 = n^{\blacksquare} \times n^{\blacksquare} = n^2$$

2.Time this function for various n e.g. n = 1,2,3.... You should have small values of n all the way up to large values. Plot "time" vs "n" (time on y- axis and n on x-axis). Also, fit a curve to your data, hint it's a polynomial.

- Let's take an example n_values as in range 1,202 with set 2 (the values will be 1,3,5,....).
- Save the time in each iteration by calling the above Algorithm.
- And now plot the "time" vs "n" (time on y- axis and n on x-axis) by the obtained values.
- We can implement the above in python as:

```
import time
import numpy as np
import matplotlib.pyplot as plot
from scipy.optimize import curve_fit

# Define the above Algorithm as function
def AlgoF(n):
   x = 1
```

```
    for i in range(1, n+1):
        for j in range(1, n+1):
            x = x + 1
    return x

# define the range of n values as in range 1,202 with set 2
n_values = np.arange(1, 202,2)

# save time of the function for each value of n
times = []
for n in n_values:
    start_time = time.time()
    AlgoF(n)
    end_time = time.time()
    times.append(end_time - start_time)

# plot the results
plot.figure()
plot.plot(n_values, times, 'bo', label='Data')

# Fit a polynomial curve to the data
def polynomial_fit(x, a, b, c):
    return a * x**2 + b * x + c

popt, _ = curve_fit(polynomial_fit, n_values, times)
fitted_curve = polynomial_fit(n_values, *popt)
plot.plot(n_values, fitted_curve, 'r-', label='Fitted Curve (Polynomial)')

plot.xlabel('n')
plot.ylabel('Time (seconds)')
plot.title('Time vs n')
plot.legend()
plot.show()
```



3.Find polynomials that are upper and lower bounds on your curve from #2. From this specify a big-O, a big-Omega, and what big- theta is.

- To find upper and lower bounds on the curve obtained from the polynomial fit, we can consider polynomials that bound the data from above and below. Let's say we have the polynomial fit given by:

$$f(n) = an^2 + bn \text{[redacted]} + c$$

- We can find upper and lower bounds by considering polynomials that are of higher and lower degree, respectively. For example, we could use:
- An upper bound:

$$g(n) = dn^3 + en^2 + fn + g \text{[redacted]}$$

where d,e,f, and g are constants chosen such that g(n) is always greater than or equal to f(n) for all n

- A lower bound:

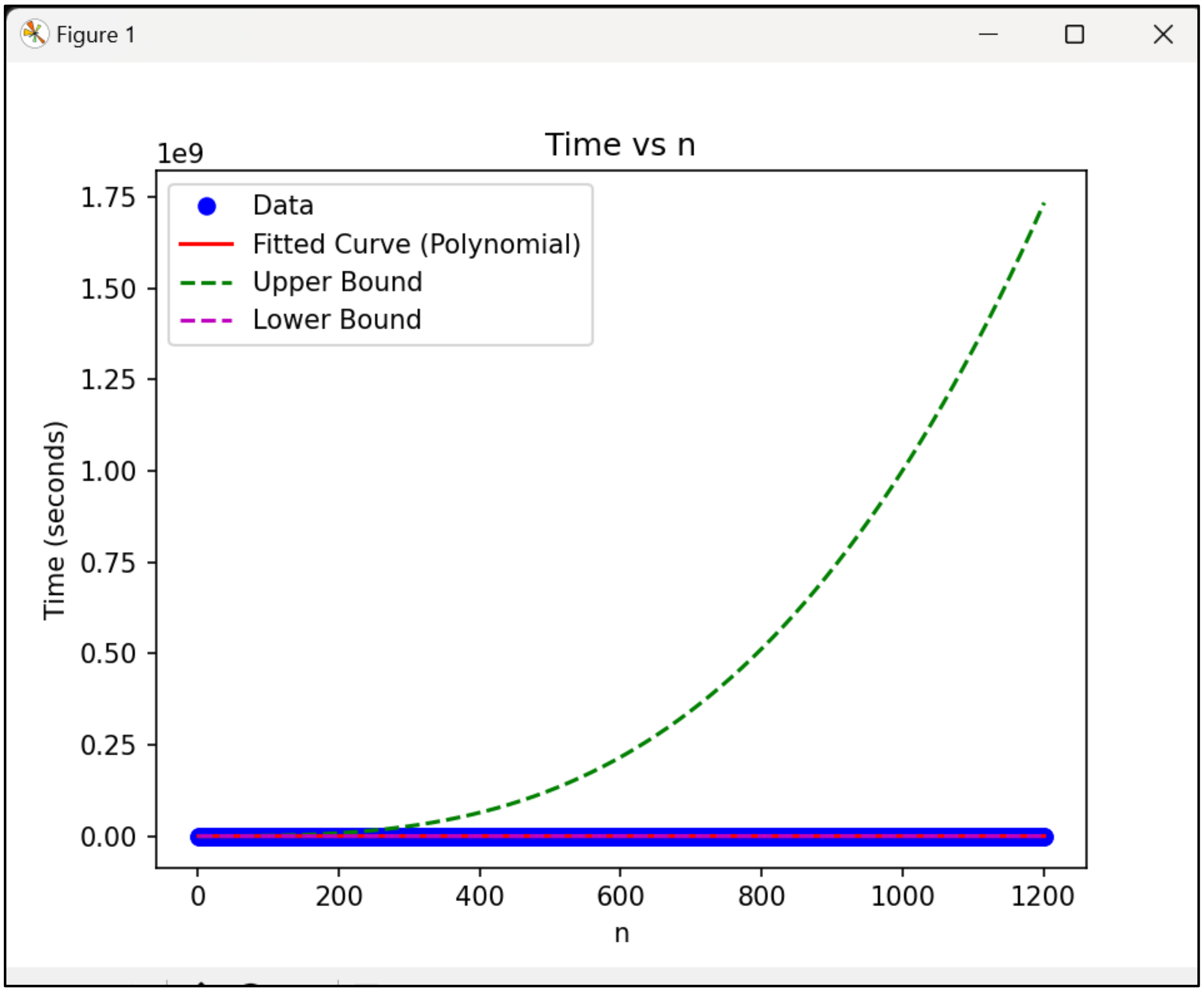
$$h(n) = hn + i$$

where h and i are constants chosen such that h(n) is always less than or equal to f(n) for all n.

- By choosing appropriate values for the constants in g(n) and h(n), we can create upper and lower bounds that encompass the behavior of the function f(n). These bounds will allow us to specify the big-O, big-Omega, and big-theta of the function.

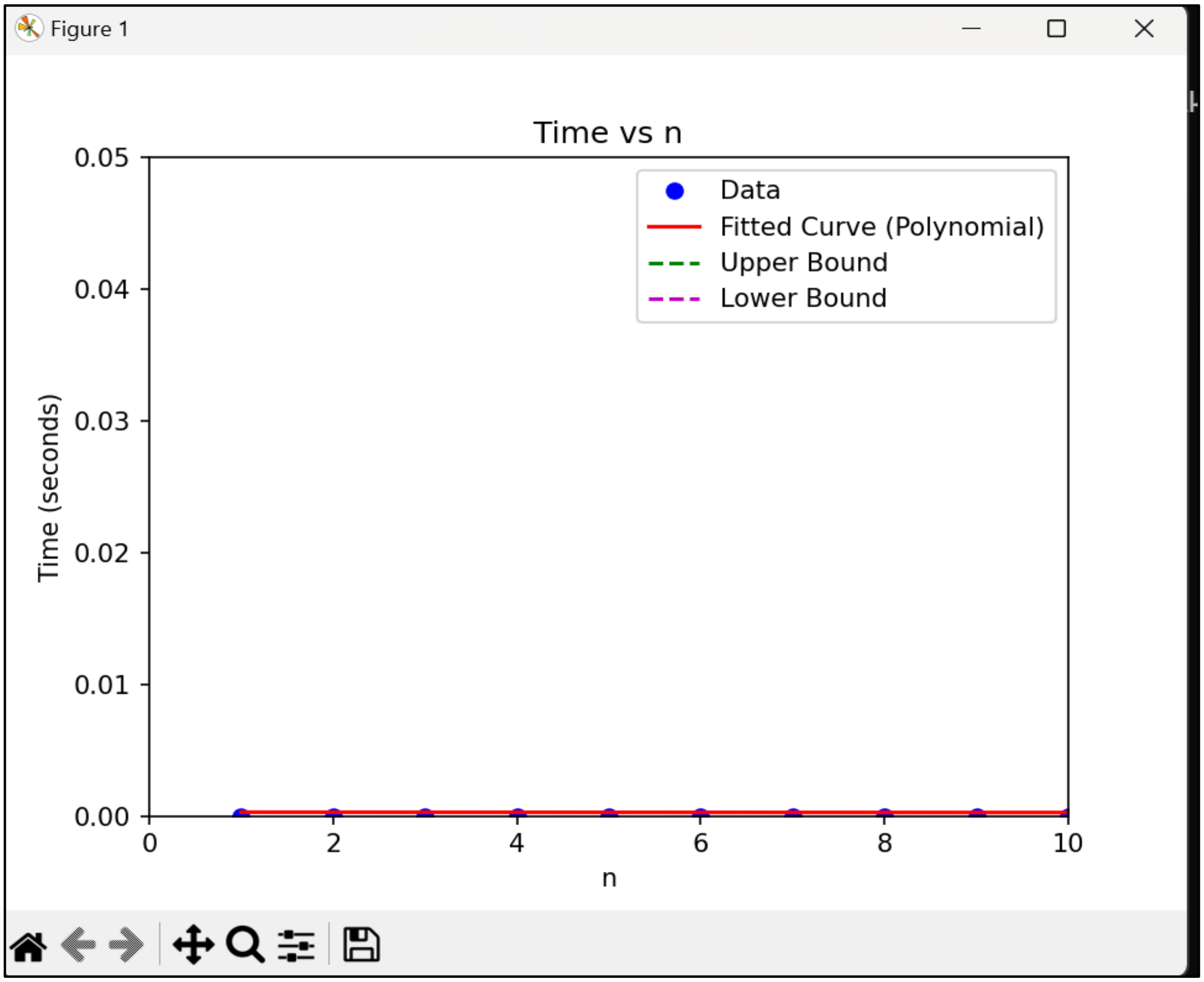
```
import numpy as np
import time
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
# Define the above Algorithm as function
def AlgoF(n):
   x = 1
   y=1
   for i in range(1, n+1):
      for j in range(1, n+1):
         x = x + 1

   return x

# define the range of n values as in range 1,202 with set 2
n_values = np.arange(1, 1202,1)

# save time of the function for each value of n
times = []
for n in n_values:
   start_time = time.time()
   AlgoF(n)
   end_time = time.time()
   times.append(end_time - start_time)
```

```
# Define the function for fitting a polynomial
def polynomial_fit(x, a, b, c):
    return a * x**2 + b * x + c

# Fit the polynomial curve to the data
popt, _ = curve_fit(polynomial_fit, n_values, times)

# Define the polynomial function
def f(n, a, b, c):
    return a * n**2 + b * n + c

# Upper bound function (higher degree polynomial)
def upper_bound(n, d, e, f, g):
    return d * n**3 + e * n**2 + f * n + g

# Lower bound function (lower degree polynomial)
def lower_bound(n, h, i):
    return h * n + i

# Extract coefficients of the polynomial fit
a, b, c = popt

# Define constants for upper and lower bounds
d_upper, e_upper, f_upper, g_upper = 1, 1, 1, 1  # Adjust as needed
h_lower, i_lower = 1, 1  # Adjust as needed

# Calculate upper and lower bounds
upper_bounds = upper_bound(n_values, d_upper, e_upper, f_upper, g_upper)
lower_bounds = lower_bound(n_values, h_lower, i_lower)

# Plot the original curve, upper and lower bounds
plt.figure()
plt.plot(n_values, times, 'bo', label='Data')
plt.plot(n_values, f(n_values, a, b, c), 'r-', label='Fitted Curve (Polynomial)')
plt.plot(n_values, upper_bounds, 'g--', label='Upper Bound')
plt.plot(n_values, lower_bounds, 'm--', label='Lower Bound')

plt.xlabel('n')
plt.ylabel('Time (seconds)')
plt.title('Time vs n')
plt.legend()
plt.show()
```



4.Find the approximate (eye ball it) location of "n_0". Do this by zooming in on your plot and indicating on the plot where n_0 is and why you picked this value. Hint: I should see data that does not follow the trend of the polynomial you determined in #2

To find the approximate location of n_0 which represents the point where the data starts to deviate from the trend of the polynomial curve, we can visually inspect the plot and identify the point where this deviation occurs. Typically, $n_0$corresponds to the point where the data starts to exhibit behavior that is inconsistent with the polynomial fit.



5.Will this increase how long it takes the algorithm to run (e.x. you are timing the function like in #2)?

- Adding the variable y = i + j inside the nested loops introduces a constant time operation, which does not change the overall time complexity of the function. Therefore, the time complexity remains O(n^2).Therefore, the time complexity remains O(n^2), and the additional variable y does not increase the runtime of the algorithm.