AMS 326 Numerical Analysis HW 1 Problem Set 1

By: Mehadi Chowdhury (115112722)

Link to GitHub where README, PDF, IPYNB and Seperate Python Files will be hosted: https://github.com/EmceeCiao/AMS_326_HW1

Background For Problem 1.1 (Methods 1-4)

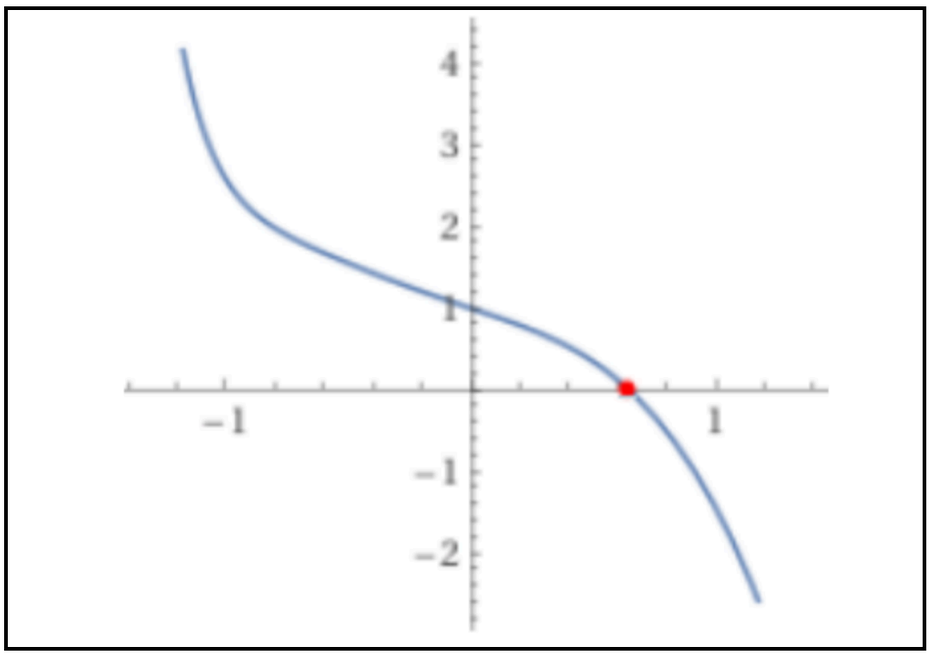
Given the following function:

$$f(x) = e^{-x^3} - x^4 - sin(x)$$

We know it has one root at

$$r \approx 0.641583$$

in the interval $x \in (0, 1)$, i.e., $f(r) \approx 0$ as shown,



Please write program(s) for each of the following methods to find the root correct within 4 decimal places, i.e., your root x_c should satisfy $|x_c - r| < 0.5 \times 10^{-4}$.

Also, please indicate how many iteration steps and estimate (either by hand count or by computer programming) how many floating-point operations needed to achieve your results from the initial (given) conditions:

Method 1 Use the bisection method with a given internal a_0 = −1 and b_0 = 1.
Method 2 Use Newton's method with the given initial root x_0 = 0.
Method 3 Use the Secant method with two given initial roots $x_{0,1}$ = −1, 1.
Method 4 Use Monte Carlo method for a given range [0.50, 0.75]. A narrowed range is given to not blow up your laptop.

Problem 1.1 - Method 1 - Bisection Method

Problem Description

We were tasked with finding the root for the function $f(x) = e^{-x^3} - x^4 - sin(x)$ using the bisection method with a given interval $a_0 = -1$ and $b_0 = 1$

Algorithm Description

The bisection method is an algorithm that finds the root of a function by choosing two initial values a_0, b_0 such that the following relation holds:

$$f(a_0) * f(b_0) < 0$$

This relation is necessary as it tells us that a root exists between these two points. Which then allows us to then iteratively take the midpoint, c, evaluate our function and cut off the half where the root does not lie until the root is found within a desired tolerance or until a maximum number of iterations is reached. In detail this is done by evaluating f(c), if f(c) is 0 then we know that is the root and we can return c. Otherwise we must check if $f(c) * f(a) < 0$ if it is we know that the root must lie in the interval [a, c], and b = c now. Else, the root is in [c, b] and c = a now.

In our program we will also be tracking the number of Floating Point Operations (FLOPS) and iterations as well. We also add an additonal check to break out as we know the actual root and want it to be within 0.5 * 10^-4.

Psuedocode

Bisection Method

$f(x) = e^{-x^3} - x^4 - \sin(x)$
$a \leftarrow -1$
$b \leftarrow 1$
$FLOPS \leftarrow 0$
$Iterations \leftarrow 0$
while $(b - a)/2 > tolerance$
  $Iterations+ = 1$
  $c \leftarrow (a + b)/2$
  $FLOPS+ = 2$
  $eval \leftarrow f(c)$
  $FLOPS+ = 18$
  if $abs(c - actualroot) < tolerance$
    return $c, FLOPS, Iterations$
  if $eval \cdot f(a) < 0$
    FLOPS += 18
    $b \leftarrow c$
  else
    FLOPS += 18
    $a \leftarrow c$
end

The final interval of [a,b] contains the root.

The approximate root is (a+b)/2

Code

In [ ]:
```
import numpy as np
import math

def f(x):
    """Function defined for f(x) evaluating the function for the x_value passed in"
    # The FLOPS of the below equation is around 17, 4 for sin,
        # 4 for e, 3 for -x^3 and 3 for x^4, 2 for -
    return np.exp(-x**3) - x**4 - np.sin(x)

def bisection_method(a, b, f, tolerance, actual_root):
    """
    Inputs:
```

```
    - a: Left starting point
    - b: Right starting point
    - f: function we are finding the root of
    - tolerance: The error we are tolerating for our break point
    - actual_root: The actual root of the function

    Output:
    - [(a+b)/2, flops, iterations]
    - (a+b)/2: root found
    - flops: Floating Point Operations done
    - iterations: The number of iterations the method took to find the answer
    """
    flops = 0
    iterations = 0
    while (b-a)/2 > tolerance:
        flops += 2
        iterations += 1
        c = (a+b)/2
        flops += 2
        val = f(c)
        flops += 18 # One more for the check right below
        if abs(c-actual_root) < tolerance:
            return [c, flops, iterations]

        # If f(c) * f(a) < 0, it lies in [a, c] range
        elif val * f(a) < 0:
            flops +=18
            b = c
        # Otherwise it lies in [c, b]
        else:
            flops += 18
            a = c
    return [(a+b)/2, flops, iterations]

# Using Bisection Method and Printing Formatted Results
results = (bisection_method(-1, 1, f, 0.5*10**-4, 0.641583))
print("Root: ", results[0])
print("Flops: ", results[1])
print("Iterations: ", results[2])
```

```
Root:  0.6416015625
Flops:  422
Iterations:  11
```

Results

The results of my program were the following:

Root $\approx$ 0.6416

Approximate Floating Point Operations: 422 (This is accounting for sin and e being 4 FLOPS and each individual operation being counted as FLOPS as well (IE: Exponentation is many multiplications, thus giving it more FLOPS)).

Iterations: 11

Performance

In terms of performance, this algorithm is approximately O(log n) where n is $|b-a|/tolerance$ because since we are dividing the search area in half each time, it only takes about O(log n) times to get to our answer as the convergence rate is known. In terms of space complexity, as we are not storing many results nor doing much complex calculations, our space complexity is minimal

Problem 1.1 - Method 2 - Newton's Method

Problem Description

We were tasked with finding the root for the function $f(x) = e^{-x^3} - x^4 - sin(x)$ using Newton's method with a given intial root $x_0 = 0$

Algorithm Description

In this case Newton's Method is an algorithm that finds the root of the function using 3 key things.

1. The Original Function f(x)
2. The Derivative of the Function f'(x)
3. An Initial Guess x_0

It uses these 3 key things to iteratively make a closer and closer approximation to the root. It hinges on the idea that we can use the tangent line at that point to find the next best point for our guess, this next best point being the x-intercept of the tangent line. More specifically the algorithm uses the following equation:

$x_{n+1} = x_n - f(x)/f'(x)$

We repeat the above until we reach our desired tolerance for the difference between our iterations, or our decided max iterations, or the limit for how small of a number we want to divide by.

In our case we will be checking for the difference between our found root and the actual root being less than our tolerance rather than the difference between iterations. Additonally we will set a max iterations limit so it doesn't infinite loop.

Psuedocode

Newton's Method

$f(x) = e^{-x^3} - x^4 - \sin(x)$
$f'(x) = -3x^2e^{-x^3} - 4x^3 - cos(x)$
$x \leftarrow 0$
$FLOPS \leftarrow 0$
$Iterations \leftarrow 0$
while $Iterations < MaxIterations$:
  $Iterations \ += 1$
  $x = x - f(x)/f'(x)$
  $FLOPS \ += 38$
  if abs(x-actualroot) < tolerance:
    FLOPS += 1
    break

return x, FLOPS, Iterations

end

Code

In [21]:
```
import numpy as np
import math

def f(x):
    # The FLOPS of the below equation is around 17
    return np.exp(-x**3) - x**4 - np.sin(x)

def f_prime(x):
    # The FLOPS of the below equations is around 19
    # 4 for e, 4 for cos, 2 for 3x^2, 3 for -x^3, 3 for 4x^3, 3 for - and * between
    return -3 * x**2 * np.exp(-x**3) - 4*x**3 - np.cos(x)
```

```
def newton_method(x0, actual_root, f, f_prime, tolerance, max_iterations):
    """
    Inputs:
    - x0: Initial root used for Newton's Method
    - actual_root: Actual root of the function used to break early
    - f : function that newton's method is being called on
    - f_prime: derivative of function that newton's method is being called on
    - tolerance: Our tolerance for error from the actual root
    - max_iterations: Iteration limit set to prevent infinite looping

    Outputs:
    - [x_n, flops, iterations]
    - x_n: root found by newton's method
    - flops: Floating Point Operations done
    - iterations: The number of iterations the method took to find the answer
    """
    x_n = x0
    iterations = 0
    flops = 0
    while(iterations < max_iterations):
        iterations += 1
        x_n = x_n - (f(x_n)/f_prime(x_n))
        flops += 38 # 2 more flops from subtraction and division! (17 + 19 + 2)

        # Check if guess is close enough
        if(abs(x_n - actual_root) < tolerance):
            break

    return [x_n, flops, iterations]

results = newton_method(0, 0.641583, f, f_prime, 0.5*10**-4, 100)
print("Root: ", results[0])
print("Flops: ", results[1])
print("Iterations: ", results[2])
```

```
Root:  0.6415825512515503
Flops:  190
Iterations:  5
```

Results

The results of my program were the following:

Root $\approx$ 0.6416

Approximate Floating Point Operations: 190 (This is accounting for cos and e being 4 FLOPS and each individual operation being counted as FLOPS as well (IE: Exponentation is many multiplications, thus giving it more FLOPS)).

Iterations: 5

Performance

In terms of performance, this algorithm is a bit harder to approximate as it depends on both whether our intial guess is good and if the function is well behaved for newton's method. Assuming these conditions, we can say that newton's method is approximately O(log n) more efficient then bisection method as it has quadratic convergence and took less iterations in our above code (It took 6 compared to 11). Thus we can say, newton's method under the right conditions is around O(log(log(n))) where n represents 1/tolerance. In terms of space complexity, as we are not storing many results nor doing much complex calculations, our space complexity is minimal

Problem 1.1 - Method 3 - Secant Method

Problem Description

We were tasked with finding the root for the function $f(x) = e^{-x^3} - x^4 - sin(x)$ using Secant method with given intial roots $x_0 = -1$, $x_1 = 1$

Algorithm Description

Similar to Newton's method the secant method uses a straight line to make better and better approximations of the root. The main difference being that newton's method relies on using the slope of the function at a point as it needs the tangent line, the secant method relies on the secant line and requires two points to calculate rather than the slope (derivative) of the function at a point.

More specifically the algorithm uses the following equation:

$x_{i+1} = x_i - (x_i - x_{i-1}) * f(x_i)/(f(x_i) - f(x_{i-1}))$

We repeat the above until we reach our desired tolerance for the difference between our iterations, or our decided max iterations.

In our case we will be checking for the difference between our found root and the actual root being less than our tolerance rather than the difference between iterations. Additonally we will set a max iterations limit so it doesn't infinite loop.

Psuedocode

Secant's Method

$f(x) = e^{-x^3} - x^4 - \sin(x)$
$x_0 \leftarrow -1$
$x_1 \leftarrow 1$
$FLOPS \leftarrow 0$
$Iterations \leftarrow 0$
while $Iterations < MaxIterations$:
  $Iterations \ += 1$
  $x_1 = x_1 - (x_1 - x_0) * (f(x_1))/(f(x_1) - f(x_0))$
  $FLOPS \ += 56$
  if abs(x-actualroot) < tolerance:
    FLOPS += 1
    break

Code

In [ ]:
```python
import numpy as np
import math

def f(x):
# The FLOPS of the below equation is around 17
    return np.exp(-x**3) - x**4 - np.sin(x)

def secant_method(x0, x1, actual_root, f, tolerance, max_iterations):
    """
    Inputs:
    - x0: one of two intial guesses for root used in secant method
    - x1: one of two intial guesses for root used in secant method
    - actual_root: actual root of the function
    - f: function that secant method is being called on
    - tolerance: Our tolerance for error from the actual root
    - max_iterations: Iteration Limit set to prevent infinite looping

    Outputs:
    - [x1, flops, iterations]
    - x1: root found by secant's method
    - flops: Floating Point Operations done
    - iterations: The number of iterations the method took to find the answer
```

```
    """
    x0 = x0
    x1 = x1
    iterations = 0
    flops = 0
    while(iterations < max_iterations):
        iterations += 1
        temp = x1

        # Formula
        x1 = x1 - ((x1-x0) * (f(x1)/(f(x1)-f(x0))))
        x0 = temp
        flops += 56

        # Break when reacing desired tolerance
        if(abs(x1 - actual_root) < tolerance):
            break
    return [x1, flops, iterations]

results = secant_method(-1, 1, 0.641583, f, 0.5*10**-4, 100)
print("Root: ", results[0])
print("Flops: ", results[1])
print("Iterations: ", results[2])
```

```
Root:  0.6415908941839591
Flops:  336
Iterations:  6
```

Results

The results of my program were the following:

Root $\approx$ 0.6416

Approximate Floating Point Operations: 336 (This is accounting for sin and e being 4 FLOPS and each individual operation being counted as FLOPS as well (IE: Exponentation is many multiplications, thus giving it more FLOPS)). Additionally, each individual call to the function incremented the FLOPS as well.

Iterations: 6

Performance

In terms of performance, this method is comparable to newton's method as we don't have the added complexity of finding a derivative. However it's time complexity algorithmically is approximately O(log n) where n is 1/tolerance, it's only slightly faster due to it's convergence rate of the golden ratio (mentioned on https://en.wikipedia.org/wiki/Secant_method). For space complexity, it is the same as newton's method and the bisection method, our space complexity is minimal.

Problem 1.1 - Method 4 - Monte Carlo Method

Problem Description

We were tasked with finding the root for the function $f(x) = e^{-x^3} - x^4 - sin(x)$ using Monte Carlo's Method for a given range [0.50, 0.75]

Algorithm Description

While there was no explicit algorithm provided, the essential idea was to randomly generate numbers using a good psuedo random number generator within the range and check to see if our result reached the condtions we seek. In this case the condtion I set was the root being

within 4 decimal points of the actual_root which was given to us in the background of the problems. This will otherwise be known as our tolerance.

Psuedocode

Monte Carlo

$lowerbound \leftarrow 0.50$
$upperbound \leftarrow 0.75$
$lowerbound \leftarrow lowerbound * 1000000$
$upperbound \leftarrow upperbound * 1000000$
$FLOPS \leftarrow 2$
$Iterations \leftarrow 0$
while $True$:
  $Iterations \ += 1$
  $FLOPS \ += 6$
  guess = random number
  if abs(guess-actualroot) < tolerance:
    FLOPS += 1
    break

Code

In [7]:
```python
import numpy as np
import random

def monte_carlo_method(actual_root, lower_bound, upper_bound, tolerance):
    """
    Monte Carlo Method of producing root through known actual root

    Input:
    - actual_root: Known root of function we are trying to approximate to
    - lower_bound: Inclusive lower bound of random number generation
    - upper_bound: Inclusive upper bound of random number generation
    - tolerance: tolerance given for difference between guess and actual_root

    Output:
    - [guess, flops, iterations]
    - guess: Successful guess/approximation of root
    - flops: Floating Point Operations Used for the method
    - iterations: Total number of iterations the method used
    """
    iterations = 0
    flops = 0
    guess = 0
    lower_bound = int(lower_bound * 1000000)
    upper_bound = int(upper_bound * 1000000)
    flops += 2
    while True:
        flops+= 6 # according to ChatGPT the PRNG of Python randrange takes around
        iterations += 1
        guess = random.randrange(lower_bound, upper_bound, 1)/1000000
        if (abs(guess-actual_root) < tolerance):
            break

    return [guess, flops, iterations]

results = monte_carlo_method(0.641583, 0.50, 0.75, 0.5*10**-4)

print("Root: ", results[0])
print("Flops: ", results[1])
print("Iterations: ", results[2])
```

```
Root:  0.641618
Flops:  386
Iterations:  64
```

Results

While the results for this program aren't consistent due to the nature of the method when it comes to the iterations and floating point operations; below I will be reporting one set of results from running the program:

Root ≈ 0.6416

Approximate Floating Point Operations: 386 (This is counting for random number generation taking 6 flops at a time, and two flops added on for creating an upper and lower bound to use the python library)

Iterations: 64

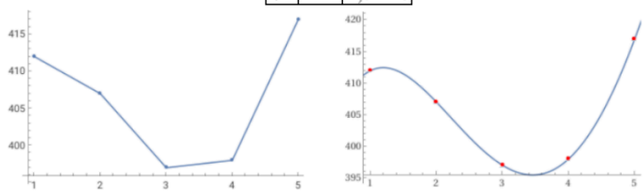
Performance

In terms of performance, this algorithm is much worse off when it comes to iterations and FLOPS as it's based on randomness compared to the other 3 methods above. The program can solve the problem in one iteration with an extremely lucky guess, but most of the time, it takes thousands. So, for time complexity it's quite hard to say but on average it should be approximately O((upperbound - lowerbound)/ (2 * tolerance)) as this is around the expected number of iterations needed assuming this follows a geometric distribution. For space complexity, it is again minimal as we aren't really storing much.

Background For Problem 1.2

Problem 1.2 (30 points, 15 for each sub-problem): The following table is the Tesla stock closings during five consecutive trading sessions. Please complete the following

t	y	Date
05	417	Jan 29
04	398	Jan 28
03	397	Jan 27
02	407	Jan 24
01	412	Jan 23



(Note: I made up the last day's stock value to make the problem more interesting!)

Please complete the following:

(1) Interpolate the data in a polynomial $P_4(t)$ and compute $P_4(t = 6)$ using your $P_4(t)$

(2) Make a quadratic fit of the data $Q_2(t) = a_0 + a_1 t + a_2 t^2$ and compute $Q_2(t = 6)$ using $Q_2(t)$.

Problem 1.2 - Task 1

Problem Description

Given a table of tesla stock closings during five consecutive trading sessions please interpolate the data in a polynomial $P_4(t)$ and compute $P_4(t = 6)$ using your $P_4(t)$

Algorithm Description

While there are many interpolation algorithms that can be used, I had interpolated using the Lagrange Interpolation method which is a solution to polynomial interpolation and uses the following idea:

$\sum_{j=0}^{k}(y_j * \prod_{m\neq j}^{0\leq m\leq k} \frac{x-x_m}{x_j-x_m})$

This formula serves as a solution to the polynomial interpolation as it ensures that the equation passes through all the points perfectly by essentially making it so that the only term that survives when plugging in a specific point's x valuye is that point's y value as a coefficient being multiplied by 1.

Psuedocode

Lagrange Interpolate

$tvalues \leftarrow [t_1, t_2, \ldots . t_n]$
$yvalues \leftarrow [y_1, y_2, \ldots . y_n]$
$point \leftarrow p$
$terms \leftarrow []$
$result \leftarrow 0$
$flops \leftarrow 0$ for i in range(n) :
  $coeff = y_i$
  for j in range(n)
    if i $\neq$ j :
      coeff =* $(point - t_j)/(t_i - t_j)$
      flops += 4
  result += coeff
return result, flops

Code

In [ ]:
```python
import numpy as np

def lagrange_interpolate(t_values, y_values, point):
    """
    Function to use create and use a lagrange interpolation

    Inputs:
    t_values: Array of t values passed in for the points
              (A point is of the form (t, y))
    y_values: Array of y values passed in for the points
              (A point is of the form (t, y))
    point: t value we are attempting to interpolate for

    Outputs:
    - prints the FLOPS, Interpolating Function along with the value for it
      at the point passed in
    - result: the result of the point found by the interpolating function
    - flops: the number of Floating Point Operations done
    """
    n = len(t_values)
    terms = []
    result = 0
    flops = 0

    for i in range(len(t_values)):
        coeff = y_values[i]
        str = f"{y_values[i]}"
        for j in range(len(t_values)):
            if i != j:
                coeff *= (point - t_values[j])/(t_values[i]-t_values[j])
                str += f" * (t - {t_values[j]} / {t_values[i]} - {t_values[j]})"
                flops += 4 # Conservatively, there are 4 operations here which may
        terms.append(str)

        result += coeff

    print(f"Final Lagrange Polynomial of Order {n}:")
    print(" +\n".join(terms))

    print(f"\nP(t = {point}) = {result}")
    print(f"Floating Point Operations: {flops}")
```

```
    return result, flops

t_values = [5, 4, 3, 2, 1]
y_values = [417, 398, 397, 407, 412]

lagrange_interpolate(t_values, y_values, 6)
print("")
```

```
Final Lagrange Polynomial of Order 5:
417 * (t - 4 / 5 - 4) * (t - 3 / 5 - 3) * (t - 2 / 5 - 2) * (t - 1 / 5 - 1) +
398 * (t - 5 / 4 - 5) * (t - 3 / 4 - 3) * (t - 2 / 4 - 2) * (t - 1 / 4 - 1) +
397 * (t - 5 / 3 - 5) * (t - 4 / 3 - 4) * (t - 2 / 3 - 2) * (t - 1 / 3 - 1) +
407 * (t - 5 / 2 - 5) * (t - 4 / 2 - 4) * (t - 3 / 2 - 3) * (t - 1 / 2 - 1) +
412 * (t - 5 / 1 - 5) * (t - 4 / 1 - 4) * (t - 3 / 1 - 3) * (t - 2 / 1 - 2)

P(t = 6) = 451.99999999999994
Floating Point Operations: 80
```

Results

$$P_4(t) = 417 * (t - 4/5 - 4) * (t - 3/5 - 3) * (t - 2/5 - 2) * (t - 1/5 - 1) +$$
$$398 * (t - 5/4 - 5) * (t - 3/4 - 3) * (t - 2/4 - 2) * (t - 1/4 - 1) +$$
$$397 * (t - 5/3 - 5) * (t - 4/3 - 4) * (t - 2/3 - 2) * (t - 1/3 - 1) +$$
$$407 * (t - 5/2 - 5) * (t - 4/2 - 4) * (t - 3/2 - 3) * (t - 1/2 - 1) +$$
$$412 * (t - 5/1 - 5) * (t - 4/1 - 4) * (t - 3/1 - 3) * (t - 2/1 - 2)$$

$$P_4(t = 6) \approx 452$$

Approximate Floating Point Operations: 80

Performance

O(n^2) where n is the number of points as we have to iterate through n y_values for each t_values which there is also n of to create the interpolation formula, thus having n^2 operations done. Once the formula is created however, the operation is O(1) to use it as multiplication, division takes O(1) time. In terms of space complexity, to build out the function I required an array of length n where n is the number of points, so the space complexity is O(n).

Problem 1.2 - Task 2

Problem Description

Given a table of tesla stock closings during five consecutive trading sessions please make a quadratic fit of the data $Q_2(t) = a_0 + a_1t + a_2t^2$ and compute $Q_2(t = 6)$ using $Q_2(t)$

Algorithm Description

For creating a quadratic fit, the algorithm we used was Least Square Curve Fitting, were we can take an inconsistent system and find the least squares solution for it. The formula and reasoning is as follows:

Given an inconsistent system of form Ax = b, we can solve $A^TA\bar{x} = A^Tb$ for the least squares solution. In this case since we are trying to do a quadratic fit of the form $a_0 + a_1t + a_2t^2$, our Ax = b will be the system of equations created by plugging in our points into the quadratic function we were given.

Specifically we are passing in $1, t, t^2$ to create matrix A as this matrix multiplied by our coefficients represented by x should lead to the y values of our points, otherwise represented by b here. We can now solve our inconsistent system here for what the coefficients of this quadratic function will be as we can isolate for $\bar{x}$ giving us the values for

a_0, a_1, a_2. With this we now know the quadratic fit function and can compute it at a specific point, which in this case would be at 6.

Pseudocode

Quadratic Fit

$tvalues \leftarrow [t_1, t_2, \ldots . t_n]$
$yvalues \leftarrow [y_1, y_2, \ldots . y_n]$
$A \leftarrow$ empty matrix of size n x 3
for i from 1 to n:
  $A[i, 1] \leftarrow 1$
  $A[i, 1] \leftarrow t_i$
  $A[i, 1] \leftarrow (t_i)^2$

$A^T \leftarrow$ tranpose A

coeffs = solution to $A^t A = A^T *$ y values

$quadraticfit \leftarrow$ coeffs[0] + coeffs[1]t + coeffs[2]t^2

$ans \leftarrow$ quadraticfit(t = point)

return ans

Code

In [ ]:

```
import numpy as np

def quadratic_fit(t_values, y_values, point_value):
    """
    Function to both create the quadratic fit function and
    print out the results achieved when evaluating the function
    at a specific point_value

    Input:
    t_values: Array of t values passed in for the points
              (A point is of the form (t, y))
    y_values: Array of y values passed in for the points of the form
              (A point is of the form (t, y))
    point: t value we are evaluating our quadratic fit for

    Output:
    prints the quadratic fit equation and the value of computing the quadratic fit
    for the t value passed in
    """

    t_values = np.array(t_values)
    y_values = np.array(y_values)

    n = len(t_values)

    A = np.vstack([np.ones(n), t_values, t_values**2]).T

    # Need to create A^TA and A^Tb and then solve these

    A_TA = np.dot(A.T, A)

    A_Ty = np.dot(A.T, y_values)

    #Solves for the coefficients
    coeffs = np.linalg.solve(A_TA, A_Ty)

    a_0 = coeffs[0]
    a_1 = coeffs[1]
    a_2 = coeffs[2]

    print(f"Q_2(t) = {a_0} + {a_1}t + {a_2}t^2\n")
```

```
    ans = a_0 + (a_1*point_value) + (a_2 * point_value**2)
    print(f"Q_2(t = {point_value}) = {ans}")

t_values = [5, 4, 3, 2, 1]
y_values = [417, 398, 397, 407, 412]

quadratic_fit(t_values, y_values, 6)
```

```
Q_2(t) = 435.39999999999765 + -25.18571428571237t + 4.214285714285411t^2

Q_2(t = 6) = 435.9999999999982
```

Results

$Q_2(t) \approx 435.39999999999765 + -25.18571428571237t + 4.214285714285411t^2$

$Q_2(t = 6) \approx 436$

Performance

The time complexity of this code generically is around O(m * n^2) where we are given the m * n matrix A. This is because multipling and m * n matrix by it's transpose which is an n * m matrix will require (m * n * n) operations, which will dominate the time complexity. However in this case talking about the code above, we can simplify the time complexity a bit more, since n is a constant 3 here due to the quadratic function only having 3 terms, we can say the time complexity of the code is O(m * 9). The space complexity of this code is O(m * n) as that's the amount of space needed to store these matrices.