Homework Set 1: Root Approximation and Extrapolation

AMS 326-01: Numerical Analysis, Spring 2025
Author: Joe Martinez

Due Febuary 21, 2025

1 Root Approximation (Problem 1.1)

For Problem 1.1, given the following function:

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

Our goal is to find the root $r \approx 0.641583$ with an accuracy of 4 decimal places using the bisection method, Newton's method, the Secant method and a Monte Carlo method. Additionally, for each method, we report the number of iterations, the runtime of the algorithm, and the total number of floating point (FP) operations it takes to reach the answer with the desired level of accuracy. We will use the following inequality $|x_c - r| < 0.5 * 10^{-4}$ and $|x_n - x_{n-1}| < 10^{-4}$ to determine the correct level of accuracy of our answer, where x_c is our final answer from a method and x_n is the value of our guess after the nth iteration.

The majority of FP operations in an iteration of a method will come from the evaluation of exponential and trigonometry functions, since modern computers use a partial sum of the appropriate Taylor series to approximate values. Since the range of values for x remains rather small from a range of [0, 1] with our given inputs, we will use the following assumptions:

1. Only a partial polynomial sum of a Taylor series is done to find the value of $\sin(x)$, $\cos(x)$, and e^x.
2. The calculation of $\sin(x)$ only goes up to the 13th degree term in the series (7 terms). This results in 19 FP operations per evaluation.
3. The calculation of $\cos(x)$ only goes up to the 12th degree term (7 terms). This results in 18 FP operations per evaluation.
4. The calculation of e^x only goes up to the 7th degree term (8 terms). This results in 19 FP operations per evaluation.

```python
# f(x) = e^(-x^3) - x^4 - sin(x)
def f(x: float) -> float:
    double_x = x * x
    exponent = -1 * double_x * x
    term1 = math.exp(exponent)
    term2 = double_x * double_x
    term3= math.sin(x)
    return term1 - term2 - term3

# f'(x) = (-3x^2)*e^(-x^3) - 4x^3 - cos(x)
def df(x: float) -> float:
    double_x = x * x
    exponent = -1 * double_x * x
    term1 = (-3 * double_x) * math.exp(exponent)
    term2 = 4 * (double_x * x)
    term3 = math.cos(x)
    return term1 - term2 - term3
```

Figure 1: Python Implementation of $f(x)$ and $f'(x)$

These assumptions should be consistent with the IEEE 754 standards for "double-precision" in floating-point values as used in the Python's math library, which aims for 15-decimal-place accuracy. With optimal code (see Figure 1), we estimate that the computation of $f(x)$ takes 44 FP operations, and for its derivative $f'(x)$ 46 FP operations are required. All algorithms, calculations, and plots are done using Python with the help of the Math, Numpy, Random, Time, and Matplotlib libraries and with predefined constants $ROOT = 0.641583$, $TOL = 10^{-4}$, $P\_ERROR = 0.5 * 10^{-4} = 0.5 * TOL$, and $ITER\_MAX = 10000$ used for our algorithms.

1.1 Bisection Method

The main idea behind the bisection method is given a range $[a, b]$, which contains a root, we divide the range in half. If the root is on the left half of the range, we shrink the range to the left half. If the root is on the right half, we shrink the range to the right half. We repeat this process until satisfied with the approximate answer. Since we divide the range in half for each iteration, the error to our solution after the nth iteration is:

$$|x_n - r| < \frac{|b - a|}{2^{n+1}}$$

Once this value is less than $0.5 * 10^{-4}$ we are sure to have a solution with 4-decimal accuracy.

Implementing Algorithm 1 in Python and using the given range of $[-1, 1]$, we get a final result $x_c = 0.64154052734375$ after 15 iterations, approximately 1490

Algorithm 1: Bisection Method 4-decimal Approximation

Input: Values a and b representing a range $[a, b]$ such that $f(a) * f(b) < 0$, $a < b$, and a root r $\in [a, b]$

Output: A root to the given function $f(x)$ with 4 decimal places accuracy and the number of iterations i

```
i ← 0;
x ← (a + b)/2;
range_size ← |b − a|;
error ← range_size/2;
while error ≥ 0.5 * 10^−4 do
    x ← (a + b)/2;
    if f(a) * f(x) < 0 then
        b ← x;
    else
        a ← x;
    end
    i ← i + 1;
    error ← error/2;
end
OUTPUT i;
return x
```

FP operations, and 0.126625 milliseconds of runtime (see Figure 2 for source code). The final error $|b - a|/2^{n+1}$ was found to be $3.0517578125 * 10^{-5}$ and $|x_c - r|$ was $4.24726 * 10^{-5}$. The values of each iteration can be found in Table 1.

1.2 Newton's Method

The logic behind Newton's method is to travel backwards along the curve of the graph to the x-intercept of the tangent line. Since $f(x)/f'(x)$ describes the distance traveled along the x-axis from the x-intercept, subtracting this value from our current guess gives us our new root estimate. The closer our guess is to the root, the smaller this distance traveled along the curve. If given a good guess, this method converges very quickly after a few iterations, but given a bad guess, the method can diverge. In order to find 4-decimal accuracy in our solution, we will use the inequality:

$$|x_n - x_{n-1}| < 10^{-4}$$

When the difference between consecutive iterations below 10^{-4}, we can infer that $|x_c - r| < 0.5 * 10^{-4}$ since the true error is approximately around half the step size as $\Delta x = x_n - x_{n-1}$ approached 0 as the number of iterations n approaches infinity in a converging method.

```python
def bisection_method(a: float, b: float) -> float:
    if a >= b:
        print(f"ERROR: a must be smaller than b")
        return

    initial_condition = f(a) * f(b)
    if initial_condition >= 0:
        print(f"ERROR: f(a)*f(b) must be below 0")
        return

    i = 0
    x = (a + b) / 2
    range_size = math.fabs(b - a)
    error = range_size / 2
    while (error >= P_ERROR):
        x = (a + b) / 2
        if f(a) * f(x) < 0: b = x
        else: a = x
        i += 1
        error = error / 2
    print(f"Number of iterations: {i}")
    print(f"Number of estimated FLOP needed: ~{95 + 93 * i}")
    print(f"Final Result: {x}")
    print(f"Final Error: {error}")
    return x
```

Figure 2: Python Implementation of the Bisection Method

Iterations	a	b	x	$f(a)$	$f(x)$
0	-1	1	0.0	2.55975	1.0
1	0.0	1	0.0	1.0	1.0
2	0.5	1	0.5	0.34057	0.34057
3	0.5	0.75	0.75	0.34057	-0.3422
4	0.625	0.75	0.625	0.04569	0.04569
5	0.625	0.6875	0.6875	0.04569	-0.1354
6	0.625	0.65625	0.65625	0.04569	-0.0418
7	0.640625	0.65625	0.640625	0.00268	0.00268
8	0.640625	0.648437	0.6484375	0.00268	-0.0193
9	0.640625	0.644531	0.64453125	0.00268	-0.0083
10	0.640625	0.642578	0.642578125	0.00268	-0.0027
11	0.640625	0.641601	0.6416015625	0.00268	-0.00005
12	0.641113	0.641601	0.64111328125	0.00131	0.00131
13	0.641357	0.641601	0.641357421875	0.00063	0.00063
14	0.641479	0.641601	0.6414794921875	0.00028	0.00028
15	0.641540	0.641601	0.64154052734375	0.000117	0.000117

Table 1: Calculations using the Bisection Method

Algorithm 2: Newton's Method 4-decimal Approximation

Input: Initial guess x_0

Output: A root to the given function $f(x)$ with 4 decimal places accuracy and the number of iterations i

$i \leftarrow 0$;

do

  $prev\_x \leftarrow x$;

  $x \leftarrow x - \frac{f(x)}{f'(x)}$;

  $error \leftarrow |x - prev\_x|$;

  $i \leftarrow i + 1$

while $error \geq 10^{-4}$;

OUTPUT i;

return x

```python
def newton_method(x: float) -> float:
    i = 0
    while True:
        prev_x = x
        x = x - (f(x) / df(x))
        error = math.fabs(x - prev_x)
        i += 1
        if error < TOL: break
        elif (i > ITER_MAX):
            print("ERROR: Reached maximum iterations. Converging too slow")
            print(f"Result: {x} \n Error: {error}\n")
            return
    print(f"Number of iterations: {i}")
    print(f"Number of estimated FLOP: ~{94 * i}")
    print(f"Final Result: {x}")
    print(f"Final Error: {error}")
    return x
```

Figure 3: Python Implementation of Newton's Method

Iterations	x
0	0
1	1.0
2	0.7389073272732694
3	0.6508467811726749
4	0.641675261983784
5	0.6415825512515503

Table 2: Calculations using Newton's Method

After implementing Algorithm 2 in Python and giving an initial guess of $x_0 = 0$, we get a result of $x_c = 0.6415825512515503$ after only 5 iterations, approximately 470 FP operations, and 0.0466250 milliseconds of runtime (see Figure 3 for source code). The final error $|x_n - x_{n-1}|$ was found to be $9.271073223371395 * 10^{-5}$ and $|x_c - r|$ is $4.48748 * 10^{-7}$. The values of each iteration can be found in Table 2.

1.3 Secant Method

The Secant method finds the roots of a function by approximating the derivative of $f(x)$ finding the slope of the secant that goes through both two root guesses. When the computation of $f'(x)$ is too expensive or impossible, this method is usually used to approximate roots. As an approximation of Newton's Method, the performance is slightly worse than Newton's method in comparison, susceptible to the same pitfalls, but overall a quickly converging method. In order to find 4-decimal accuracy in our solution, we will use the same inequality as in Newton's method for measuring the difference between consecutive iterations:

$$|x_n - x_{n-1}| < 10^{-4}$$

Implementing Algorithm 3 in Python an giving the two initial guesses of $x_0 = -1$ and $x_1 = 1$, we get a result of $x_c = 0.6415825351325788$ after only 7 iterations, approximately 666 FP operations, and a runtime of 0.0545420 milliseconds (See Figure 4). The final error $|x_n - x_{n-q}|$ was found to be $8.35905138030224 * 10^{-6}$ and $|x_c - r|$ was $4.64867 * 10{-}7$. The values of each iteration can be found in Table 3.

1.4 Monte Carlo Method

The Monte Carlo method relies on the repeated generation of random numbers to guess a solution. Given such a random nature, a large number of iterations are needed to find the solution, but with a strike of incredibly luck one can find the solutions in a very short amount of iterations as well. It is incredibly hard to guess the root and estimate the error of a guess, therefore in order to find 4-decimal place accuracy in our solution we use a predefined value of $r = 0.641583$ to find the error in each iteration.

Algorithm 3: Secant Method 4-decimal Approximation

Input: Two initial guesses x_0 and x_1

Output: A root to the given function $f(x)$ with 4 decimal places accuracy and the number of iterations i

do

$y1 \leftarrow f(x1)$;

$y2 \leftarrow f(x2)$;

$denom \leftarrow \frac{y2-y1}{x2-x1}$;

$x3 = x2 - \frac{y2}{denom}$;

$x1 = x2$;

$x2 = x3$;

$error \leftarrow |x2 - x1|$;

$i \leftarrow i + 1$;

while $error \geq 10^{-}4$;

OUTPUT i;

return x2

```python
def secant_method(x1: float, x2: float) -> float:
    if x1 == x2:
        print("ERROR: x1 cannot equal x2")
        return

    i = 0
    while True:
        fx2 = f(x2)     # Calculate f(x2) once to save FP
        denom = (fx2 - f(x1)) / (x2 - x1)
        x3 = x2 - (fx2 / denom)
        x1 = x2
        x2 = x3
        i += 1
        error = math.fabs(x2 - x1)
        if error < TOL: break
        elif (i >= ITER_MAX):
            print("ERROR: Reached maximum iterations. Converging too slow")
            print(f"Result: {x2} \n Error: {error}\n")
            return
    print(f"Number of iterations: {i}")
    print(f"Number of estimated FLOP: ~{95 * i + 1}")
    print(f"Final Result: {x2}", math.fabs(x2 - ROOT) < P_ERROR)
    print(f"Final Error: {error}")
    return x2
```

Figure 4: Python implementation of the Secant Method

Iteration	x_1	x_2
0	-1	1
1	1	0.2692954465371731
2	0.2692954465371731	0.5067386384603219
3	0.5067386384603219	0.7094891148394559
4	0.7094891148394559	0.6312838607577576
5	0.6312838607577576	0.6408442416804365
6	0.6408442416804365	0.6415908941839591
7	0.6415908941839591	0.6415825351325788

Table 3: Calculations using the Secant method

Iteration	Final x_c	Final Error $\lvert x_c - r \rvert$	Runtime (ms)
482	0.6415490513804902	$3.3948619 * 10^{-05}$	0.117582 ms
2528	0.6416025426123784	$1.9542612 * 10^{-05}$	0.450499 ms
1641	0.6416033923862334	$2.0392386 * 10^{-05}$	0.295959 ms
4967	0.6415358056603104	$4.7194339 * 10^{-05}$	0.880541 ms
1313	0.6415872067335091	$4.2067335 * 10^{-06}$	0.236041 ms
227	0.6415643235656251	$1.8676434 * 10^{-05}$	0.043667 ms

Table 4: Calculations using the Monte Carlo Method

Implementing Algorithm 4 in Python as shown in Figure 5 with the help of the random numbers library Random, specifically the library function "random.uniform(a, b)" to produce a random number in the range $[a, b]$, a solution is found in a varying number of iterations with varying runtime.

Using statistics, we can determine that the expected number of iterations using the Monte Carlo method is around 2500. This is derived from observing that the number of iterations it takes to find an acceptable answer follows a geometric probability distribution, with a probability of success $p = 0.0004$. This probability is derived by computing the probability of a random number x_c being in the range approximately $[r - 0.5 * 10^{-4}, r + 0.5 * 10^{-4}]$ over a uniform probability distribution from $[0.5, 0.75]$. The expected value of a geometric distribution is $1/p$, therefore $1/0.0004 = 2500$ iterations are needed.

The number of FP operations needed are 3 times $i + 1$ iterations plus 1 given our implementation of this algorithm. Therefore, the excepted number of FP operations using this Monte Carlo method is 7504. Table 4 demonstrates 6 simulations using this method along with their iterations. Overall, the runtime using the given range of values is less than millisecond.

Algorithm 4: Monte Carlo Method 4-decimal Approximation

Input: Values a and b representing a range $[a, b]$ where ROOT $\in [a, b]$
Output: A root to the given function $f(x)$ with 4 decimal places correct and the number of iterations i

$i \leftarrow 0$;
$x \leftarrow$ a random value in the range $[a, b]$;
$error \leftarrow |x - ROOT|$;
while $error \geq 0.5 * 10^{-4}$ **do**
    $x \leftarrow$ a random value in the range $[a, b]$;
    $error \leftarrow |x - ROOT|$;
    $i \leftarrow i + 1$;
end
OUTPUT i;
return x

```python
def monte_carlo_method(a: float, b: float, verbose = False) -> float:
    if a >= b:
        print("ERROR: a cannot be greater than or equal to b")
        return

    i = 1
    x = random.uniform(a, b)
    error = math.fabs(x - ROOT)
    while (error >= P_ERROR):
        if (verbose): print(f"\tIteration {i}: {x}")
        x = random.uniform(a, b)
        error = math.fabs(x - ROOT)
        i += 1
    print(f"Number of iterations: {i}")
    print(f"Number of estimated FLOP: ~{3*i + 4}")
    print(f"Final Result: {x}")
    print(f"Final Error: {error}")
    return x
```

Figure 5: Python implementation of the Monte Carlo Method

t	y
05	417
04	398
03	397
04	407
01	412

Table 5: Tesla Stocks Closing Data

2 Extrapolation (Problem 1.2)

For Problem 1.1, we were given a table of Tesla stock closing (Table 5) during five consecutive trading sessions.

Our goal is to find a polynomial function $P_4(t)$ that interpolates the data, and then a second polynomial function that fit the data in quadratic form quadratic $Q_4(t) = c_2t^2 + c_1t + c_0$.

2.1 Polynomial Interpolation

We can find a polynomial equation that interpolates the data by making a system of equations with shared coefficients for each data point and then solving for the coefficients.

$$\begin{bmatrix} t_0^n & t_0^{n-1} & ... & t_0 & 1 \\ t_1^n & t_1^{n-1} & ... & t_1 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ t_n^n & t_n^{n-1} & ... & t_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

$$P_n(t) = a_nt^n + a_{n-1}t^{n-1} + ... + a_1t + a_0$$

Plugging in our values of of t and y, we get the following system of equations

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 16 & 8 & 4 & 2 & 1 \\ 81 & 27 & 9 & 3 & 1 \\ 256 & 64 & 16 & 4 & 1 \\ 625 & 125 & 25 & 5 & 1 \end{bmatrix} \begin{bmatrix} a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} 412 \\ 407 \\ 397 \\ 398 \\ 417 \end{bmatrix}$$

Using the Python code in Figure 6 with the help of a numerical calculation library Numpy to solve this matrix equation of the form $Ax = b$, we find the following vector of coefficients in 0.516457948833704 milliseconds of runtime:

$$\begin{bmatrix} a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \approx \begin{bmatrix} -0.375 \\ 6.41666667 \\ -31.625 \\ 50.5833333 \\ 387 \end{bmatrix}$$

```python
def interpolate_polynomial(x: list, y: list) -> np.ndarray:
    equations = []
    num_points = len(x)
    for i in range(0, num_points):
        polynomial = []
        point = x[i]
        for j in range(num_points - 1, -1, -1):
            polynomial.append(math.pow(point, j))
        equations.append(polynomial)

    A = np.array(equations)
    coeff = np.linalg.solve(A, y)
    print("Coefficient Matrix:\n" + str(A))
    print("Results:\n" + str(coeff))
    return coeff
```

Figure 6: Python code for calculating the coefficients of the interpolation equation from an array of x and y values using Numpy

$$\therefore P_4(t) \approx -0.375t^4 + 6.41666667t^3 - 31.625t^2 + 50.5833333t + 387$$

$$P_4(t) = -\frac{3}{8}t^4 + \frac{77}{12}t^3 - \frac{253}{8}t^2 + \frac{607}{12}t + 387$$

Plugging in for $t = 6$ of $P_4(t)$, we get the following a $P_4(6) = 462$. A plot of $P_4(t)$ with the point $(6, P_4(6))$ in Figure 8.

2.2 Quadratic Fit

We can find a quadratic equation that fits the data by fitting each data point into a degree 2 polynomial and building a system of equations.

$$A = \begin{bmatrix} t_0^2 & t_0 & 1 \\ t_1^2 & t_1 & 1 \\ \vdots & \vdots & \vdots \\ t_n^2 & t_n & 1 \end{bmatrix} \quad x = \begin{bmatrix} x_2 \\ x_1 \\ x_0 \end{bmatrix} \quad b = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

$$Ax = b$$

Plugging in our values of of t and y, we get the following system of equations.

$$\begin{bmatrix} 1 & 1 & 1 \\ 4 & 2 & 1 \\ 9 & 3 & 1 \\ 16 & 4 & 1 \\ 25 & 5 & 1 \end{bmatrix} \begin{bmatrix} c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} 412 \\ 407 \\ 397 \\ 398 \\ 417 \end{bmatrix}$$

For this inconsistent system $Ax = b$, the least squares solution can be found by solving $A^T A\bar{x} = A^T y$. Using the least squares solution we compose our

```python
def quadratic_fit(x: list, y: list) -> np.ndarray:
    equations = []
    for n in x:
        array = []
        for p in range(2, -1, -1):
            array.append(math.pow(n, p))
        equations.append(array)
    A = np.array(equations)
    A_t = A.transpose()
    M = A_t @ A
    b = A_t @ y
    # Mx = b
    x_hat = np.linalg.solve(M, b)
    print(f"Coefficient Matrix:\n" + str(A))
    print(f"\nResults: " + str(x_hat))
    return x_hat
```

Figure 7: Python code for calculating the coefficients of the quadratic equation from an array of x and y values using Numpy

quadratic equation $Q_n(t) = c_2t^2 + c_1t + c_0$ where $\bar{x} = [c_2, c_1, c_0]$. Using the Python code in Figure 7 using Numpy to solve this matrix equation, we get three coefficients of the quadratic equation in 0.8623750181868672 milliseconds of runtime.

$$c = \begin{bmatrix} c_2 \\ c_1 \\ c_0 \end{bmatrix} \approx \begin{bmatrix} 4.21428571 \\ -25.18571429 \\ 435.4 \end{bmatrix}$$

$$\therefore Q_4(t) = 4.21428571t^2 - 25.18571429t + 435.4$$

Plugging in for $t = 6$, for we get a the follow $Q_4(6) = 436.0000000000007 \approx 436$. A plot of $Q_4(t)$ with the point $(6, Q_4(6))$ in Figure 8.

3 Appendix

The source code used for all of the calculations and graphs is found at as hw1.py:
https://github.com/joeyboi145/ams326_scripts
https://github.com/joeyboi145/ams326_scripts/blob/main/hw1.py

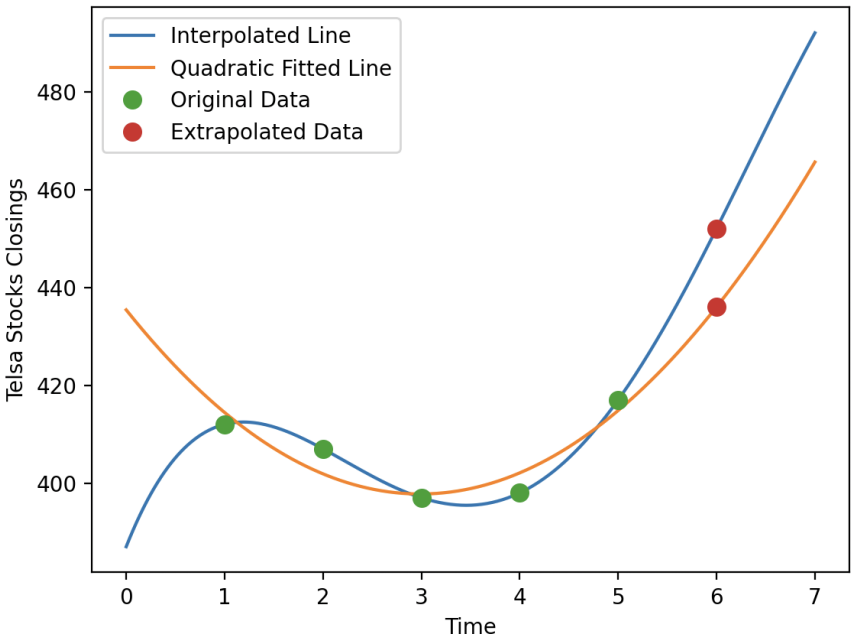

Figure 8: Plot of Tesla Stocks vs. Time with interpolation equation $P_4(t)$, fitted quadratic equation $Q_4(t)$, original data points, $(6, P_4(6))$ and $(6, Q_4(6))$