# AMATH 585 - Homework 5 - Plots, Code, and Discussions

Nate Whybra

February 2025

## Problem 1 (a)

```python
# Problem 1 (a).
a, b = 4, 5
tol = 1e-6
n = int(np.ceil(np.log((b-a) / tol) / np.log(2)))

print("Problem 1(a): Bisection Method")
for i in range(n):
    x = (a + b) / 2

    if f(x) > 0:
        a = x
    else:
        b = x

    # Print interval.
    print("Iteration " + str(i) + ":", (a, b))
```

Figure 1: Implementation

```
Problem 1(a): Bisection Method
Iteration 0: (4.5, 5)
Iteration 1: (4.75, 5)
Iteration 2: (4.875, 5)
Iteration 3: (4.9375, 5)
Iteration 4: (4.9375, 4.96875)
Iteration 5: (4.953125, 4.96875)
Iteration 6: (4.9609375, 4.96875)
Iteration 7: (4.96484375, 4.96875)
Iteration 8: (4.96484375, 4.966796875)
Iteration 9: (4.96484375, 4.9658203125)
Iteration 10: (4.96484375, 4.96533203125)
Iteration 11: (4.965087890625, 4.96533203125)
Iteration 12: (4.965087890625, 4.9652099609375)
Iteration 13: (4.965087890625, 4.96514892578125)
Iteration 14: (4.965087890625, 4.965118408203125)
Iteration 15: (4.9651031494140625, 4.965118408203125)
Iteration 16: (4.965110778808594, 4.965118408203125)
Iteration 17: (4.965110778808594, 4.965114593505859)
Iteration 18: (4.965112686157227, 4.965114593505859)
Iteration 19: (4.965113639831543, 4.965114593505859)
```

Figure 2: Smallest known interval containing root after each iteration, accurate to 6 decimal places. Here iteration 0 is not the starting point, but rather the first step in the iteration.

Without running the code further, from (4.33), with $tol = 10^{-12}$ and $b - a = 5 - 4 = 1$, we'd need $n = \lceil \frac{\log 10^{12}}{\log 2} \rceil = 40$ iterations total to achieve the tolerance. To get to 6 decimal place accuracy ($tol = 10^{-6}$), we had to go for $n = \lceil \frac{\log 10^{6}}{\log 2} \rceil = 20$ iterations.

# Problem 1 (b)

```python
# Problem 1 (b).
error = np.inf
tol = 1e-8
x = 5
k = 0

print("Problem 1(b): Newton's Method")
while error > tol:
    # Get function value.
    fofx = f(x)
    print("Iteration " + str(k) + ":" + " (x = " + str(x) + ", f(x) = " + str(fofx) + ")")

    # Update.
    x = x - (fofx / df(x))
    error = np.abs(fofx)
    k = k + 1
```

Figure 3: Implementation

```
Problem 1(b): Newton's Method
Iteration 0: (x = 5, f(x) = -5.0)
Iteration 1: (x = 4.966310265004573, f(x) = -0.16564277761091706)
Iteration 2: (x = 4.965115686301458, f(x) = -0.0002012018060986165)
Iteration 3: (x = 4.96511423174643, f(x) = -2.978897128969038e-10)
```

Figure 4: Here iteration 0 is the starting point.

From iteration $1 \to 2$ we improved by 3 decimal places. From $2 \to 3$ we improved 6 decimal places. If this trend continued, we'd improve 9 decimal places from $3 \to 4$, so I predict it would take 1 more step to achieve 16 decimal point accuracy.

# Problem 1 (c)

```python
# Problem 1 (c).
error = np.inf
tol = 1e-8
k = 0
x_prev = 4
x_curr = 5

print("Problem 1(c): Secant Method")
while error > tol:
    # Print before.
    print("Iteration " + str(k) + ":" + " (x = " + str(x_prev) + ", f(x) = " + str(f(x_prev)) + ")")

    # Assign temp values.
    f_curr = f(x_curr)
    f_prev = f(x_prev)

    # Update.
    x = x_curr - f_curr * (x_curr - x_prev) / (f_curr - f_prev)
    error = np.abs(f(x))
    k = k + 1
    x_prev = x_curr
    x_curr = x

# Print after.
print("Iteration " + str(k) + ":" + " (x = " + str(x) + ", f(x) = " + str(f(x)) + ")")
```

Figure 5: Implementation

```
Problem 1(c): Secant Method
Iteration 0: (x = 4, f(x) = 49.598150033144236)
Iteration 1: (x = 5, f(x) = -5.0)
Iteration 2: (x = 4.908421805556329, f(x) = 7.402024407938184)
Iteration 3: (x = 4.963079336311798, f(x) = 0.2808942198028612)
Iteration 4: (x = 4.965235312126352, f(x) = -0.016750499040933065)
Iteration 5: (x = 4.965114231713327, f(x) = 4.280998666672531e-09)
```

Figure 6: Here iteration 0 is the starting point.

From iteration $2 \to 3$ we improved by 1 decimal place. From $3 \to 4$ we improved by 1 decimal place. From $4 \to 5$ we improved by 7 decimal places. The rate of convergence increases dramatically here, if this trend were to continue I suspect it would take 1 or 2 more iterations to get to 16 decimal place accuracy.

## Problem 3

```python
# Problem 3.
c = 0.5
x = c * np.pi
n = 100

for i in range(n):
    x = np.cos(x)

    if i == n-1:
        print(x)
```

```
0.7390851332151607
```

Figure 7: Hitting cosine over and over for Problem 3.

# Problem 4

```
# Problem 4.
a0 = 1/4 + np.sqrt(6) / 36
a1 = 1/4 - np.sqrt(6) / 36
x0 = (6 + np.sqrt(6)) / 10
x1 = (6 - np.sqrt(6)) / 10

J = np.array([[1, 1, 0, 0], [x0, x1, a0, a1], [x0 ** 2, x1 ** 2, 2*a0*x0, 2*a1*x1], [x0 ** 3, x1 ** 3, 3 * a0 * (x0 ** 2), 3 * a1 * (x1 ** 2)]])
det_J = np.linalg.det(J)
print("Jacobian determinant:", det_J)
```

```
Jacobian determinant: -0.0033333333333333319
```

Figure 8: Computing the determinant of the Jacobian for Problem 4.

## Problem 5

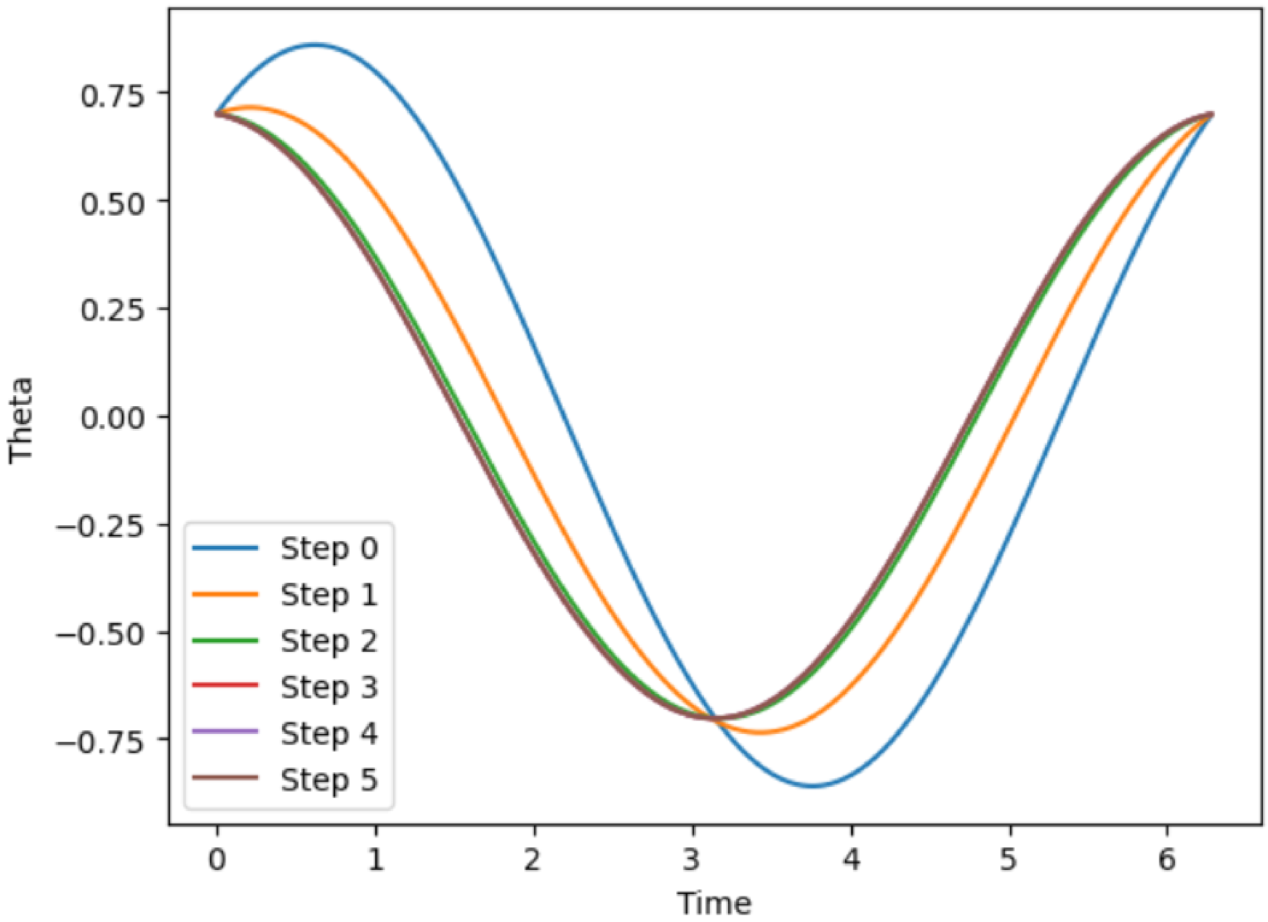


Figure 9: Solution 1. Iteration with $\theta^0 = 0.7\cos(t) + 0.5\sin(t)$ and $n = 1000$.

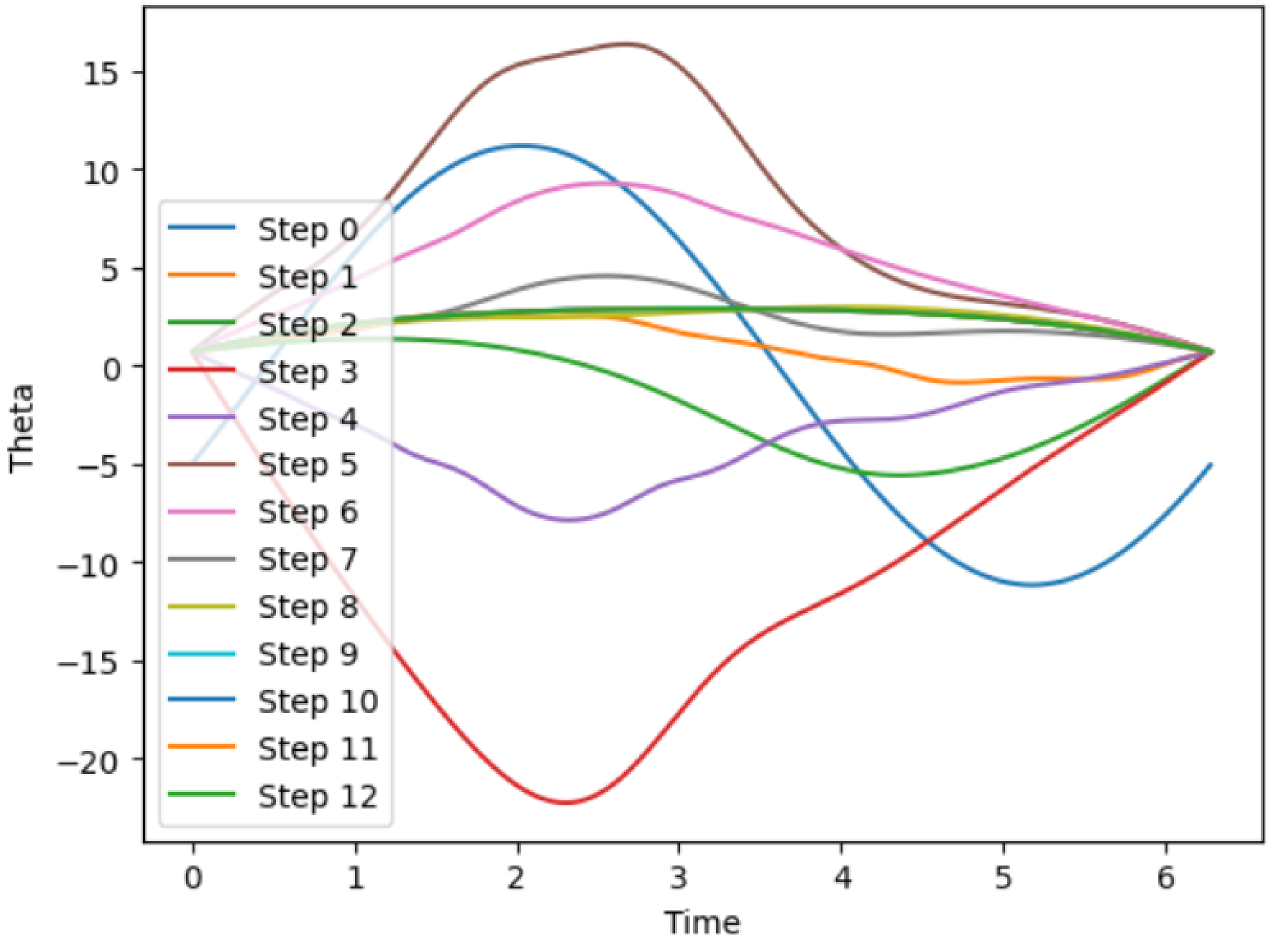


Figure 10: Solution 2. Iteration with $\theta^0 = -5\cos(t) + 10\sin(t)$ and $n = 1000$.

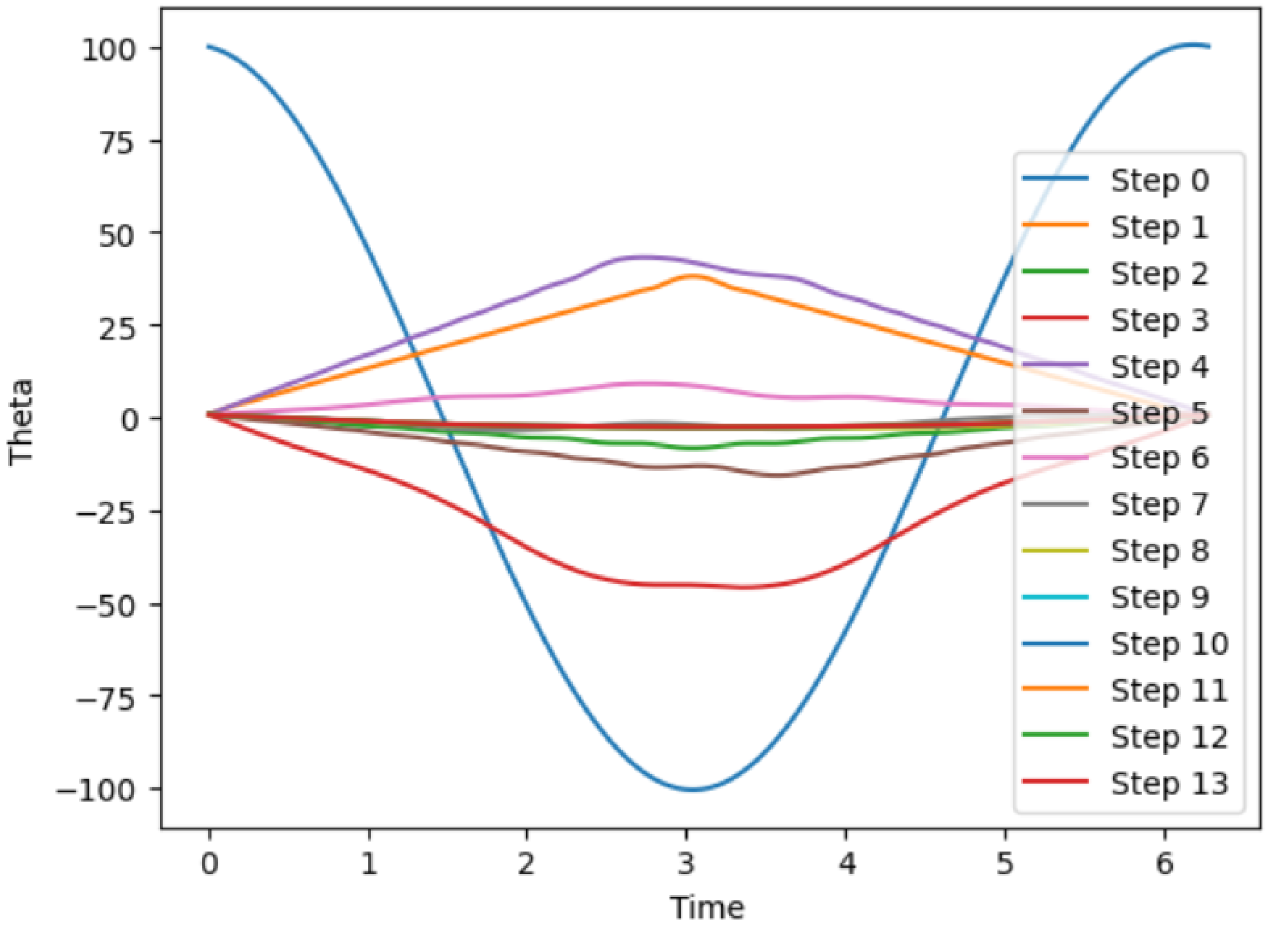


Figure 11: Solution 3. Iteration with $\theta^0 = 100\cos(t) - 10\sin(t)$ and $n = 1000$.

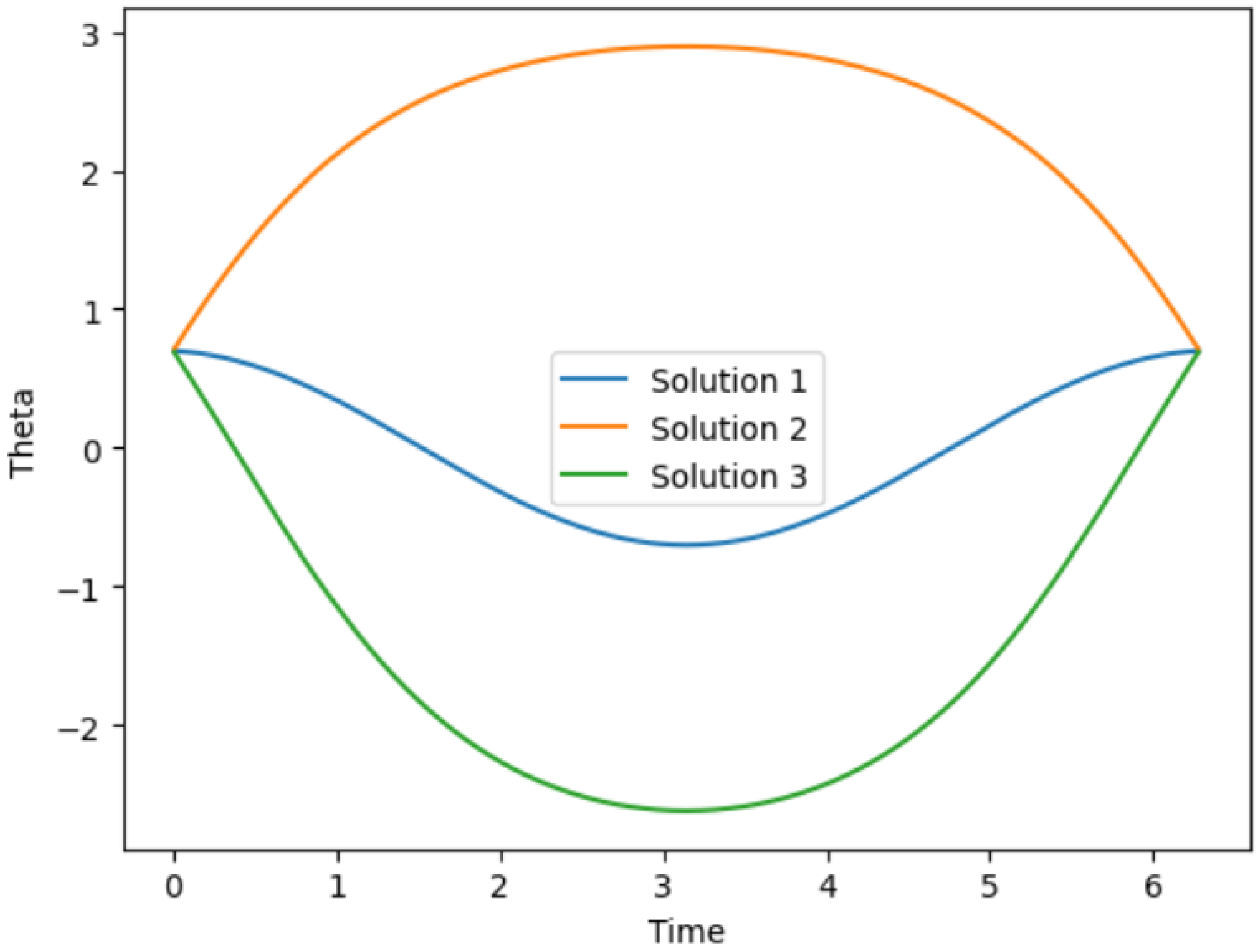


Figure 12: All solutions plotted.

By changing the initial conditions, I was able to find 3 solutions. If $\theta = 0$ corresponds to the pendulum being in the center position, Solution 1 would correspond to 1 single oscillation of a pendulum (back and forth over the center point) whereas Solutions 2 and 3 correspond to the pendulum swinging up through and back down to its starting position without passing through the center point. You can also notice that if the initial guess had a larger magnitude (appropriate norm), then it generally took more steps to converge. I used $\sum_i |\Delta_{n,i}| \leq 10^{-6}$ as my stopping criterion (when the sum of absolute changes was small enough).