STAT 31020: Homework 1

Caleb Derrickson

Wednesday, January 10, 2024

Contents

Problem 1

Let $f(x) : \mathbb{R} \to \mathbb{R}$ be an infinitely differentiable function. Let $k(x)$ be the smallest integer that satisfies $f^{(j)}(x) = 0, 1 \leq k \leq k-1, f^{(k)}(x) \neq 0$ (that is the smallest index of the derivative that is nonzero. Here, $f^{(l)}(x)$ is the l-th derivative of f at x.

Problem 1, Part 1

Prove that, if $k(x)$ is odd, then x is neither a local minimum nor a local maximum.

Solution:

Since f is given as arbitrarily differentiable, Taylor's Theorem can be applied. I will provide the theorem below, for reference.

Taylor's Theorem [1]

Let $k \geq 1$ be an integer and let the function $f : \mathbb{R} \to \mathbb{R}$ be k times differentiable at the point $a \in \mathbb{R}$. Then there exists a function $h_k : \mathbb{R} \to \mathbb{R}$ such that

$$f(x) = f(a) + f'(a) + \frac{1}{2!}\frac{d^2 f}{dx^2}\bigg|_{x=a}(x-a)^2 + \ldots + \frac{1}{k!}\frac{d^k f}{dx^k}\bigg|_{x=a}(x-a)^k + h_k(x)(x-a)^k,$$

where $\lim_{x \to a} h_k(x) = 0$.

Supposing k is odd, then $k = 2m+1$, for $m \in \mathbb{N} \cup \{0\}$.[2] Not also by assumption, all derivatives of f evaluated at a are zero, thus the Taylor expansion of $f(x)$ is simplified to

$$f(x) = f(a) + f'(a) + \frac{1}{(2m+1)!}\frac{d^{(2m+1)} f}{dx^{(2m+1)}}\bigg|_{x=a}(x-a)^{(2m+1)} + h_{(2m+1)}(x)(x-a)^{(2m+1)}.$$

Assuming we can truncate f (i.e., discard the function $h(x)$) around a neighborhood U_a of a, then we can approximate $f(x)$ for $x \in U_a$ as

$$f(x) \approx f(a) + \frac{1}{(2m+1)!} f^{(2m+1)}(x=a)(x-a)^{(2m+1)}.$$

Thus, we can analyze f according to the extremality of $f(a)$.

[1] This theorem is taken from the Wikipedia page for Taylor's Theorem, directly under the section titled "Taylor's theorem in one real variable."

[2] I am writing it like this to avoid ambiguity, since I am used to having the natural numbers without 0.

- **Case 1:** $f(a)$ is a local minimum.

  Then $f(x) - f(a) \geq 0$ should be true for all $x \in U_a$. This means $\frac{1}{(2m+1)!} f^{(2m+1)}(x = a)(x - a)^{(2m+1)} \geq 0$, so $(x - a)^{(2m+1)} \geq 0$. Note that this is an odd function, so for points less than a, this inequality cannot hold. Thus $f(a)$ cannot be a local minimum.

- **Case 2:** $f(a)$ is a local maximum.

  Then the same argument can be made in the previous case, replacing the inequality with $f(x) - f(a) \leq 0$. Then $(x - a)^{(2m+1)} \leq 0$, and since this is an odd function, we cannot find a neighborhood around a such that this inequality is satisfied for all points within the neighborhood. Thus $f(a)$ cannot be a local maximum.

Problem 1, Part 2

Prove that if $k(a)$ is even, then a is either a strict local minimum or a strict local maximum.

Solution:

Given we are in the same environment as in the previous part, we will truncate f so that around a neighborhood U_a of a, $f(x)$ can be reasonably approximated as

$$f(x \in U_a) \approx f(a) + \frac{1}{(k)!} f^{(k)}(x = a)(x - a)^{(k)}.$$

Since k is even, it can be expressed as $k = 2m$, for $m \in \mathbb{N} \cup \{0\}$. Then we can analyze $f(a)$ based on the value of $f(x) - f(a)$ Note that the case where $f(x) = f(a)$ is not considered, since this would imply $f^{(k)}(x = a) = 0$, which under our assumption of k cannot happen.

- **Case 1:** $f(x) - f(a) > 0$.

  This implies $\frac{1}{(2m)!} f^{(2m)}(x = a)(x - a)^{(2m)} > 0$, so $f^{(2m)}(x = a) > 0$, since $\frac{1}{(2m)!}(x - a)^{(2m)} \geq 0$ for any $x \in U_a, m \in \mathbb{N} \cup \{0\}$. Therefore, since $f^{(2m)}(x = a) > 0$, a is a strict local minimum.

- **Case 2:** $f(x) - f(a) < 0$.

  Then $\frac{1}{(2m)!} f^{(2m)}(x = a)(x - a)^{(2m)} < 0$, so $f^{(2m)}(x = a) < 0$, since $\frac{1}{(2m)!}(x - a)^{(2m)} \geq 0$ for any $x \in U_a, m \in \mathbb{N} \cup \{0\}$. Therefore, since $f^{(2m)}(x = a) < 0$, a is a strict local maximum.

Therefore, a can either be a strict local maximum or minimum, depending on the sign of $f(x) - f(a)$.

Problem 1, Part 3

In the same situation, prove that a is an isolated extremum.

Solution:

Assuming that a is a local extremum of f, then we can apply the first order necessary conditions to get that $f'(a) = 0$. Applying Taylor's Theorem to $f'(x)$ then gives

$$\frac{df}{dx}(x) = \frac{df}{dx}\bigg|_{x=a} + \frac{d^2f}{dx^2}\bigg|_{x=a}(x-a) + \frac{1}{2!}\frac{d^3f}{dx^3}\bigg|_{x=a}(x-a)^2 + \ldots + \frac{1}{(k-1)!}\frac{d^kf}{dx^k}\bigg|_{x=a}(x-a)^{(k-1)} + h_k(x)$$

This is allowed in some neighborhood U_a of a. Note that all derivatives prior to the k-th are zero. Suppose further that U_a also satisfies the approximation performed in the previous parts, such that

$$\frac{df}{dx}(x \in U_a) \approx \frac{1}{(k-1)!}\frac{d^kf}{dx^k}\bigg|_{x=a}(x-a)^{(k-1)}.$$

Note that if $x' \in U_a$ is an extremum, then $\frac{df}{dx}(x') = 0$. When evaluating $\frac{df}{dx}$ at $x = x'$ and setting it equal to zero, we get

$$\frac{df}{dx}(x') = 0 = \frac{1}{(k-1)!}\frac{d^kf}{dx^k}\bigg|_{x=a}(x'-a)^{(k-1)}.$$

By assumption, $\frac{d^kf}{dx^k}(x=a) \neq 0$. We can cancel this out, plus the factor of $(k-1)!$ to get $(x'-a)^{(k-1)} = 0$. This then implies $x' = a$. Thus there are no other extrema within the neighborhood U_a, meaning a is an isolated extremum.

Problem 1, part 4

Give an example where $k(a) = \infty$, (i.e. all derivatives vanish at a), but a is a strict isolated local minimum.

Solution:

If we consider the function

$$f(x) = \begin{cases} \exp\left(-\frac{1}{x^2}\right), & x \neq 0 \\ 0, & x = 0 \end{cases},$$

Then we can note that $f(x)$ is piecewise continuous and infinitely differentiable, where evaluating at the point $a = 0$ gives an isolated minimum. Note the inclusion of $f(0) = 0$ is necessary, as evaluating the function $\exp\left(-\frac{1}{x^2}\right)$ at $x = 0$ is undefined. Note also the function "levels out" near the origin, but always has a nonzero slope. I have included an image of the function below in Figure 1.

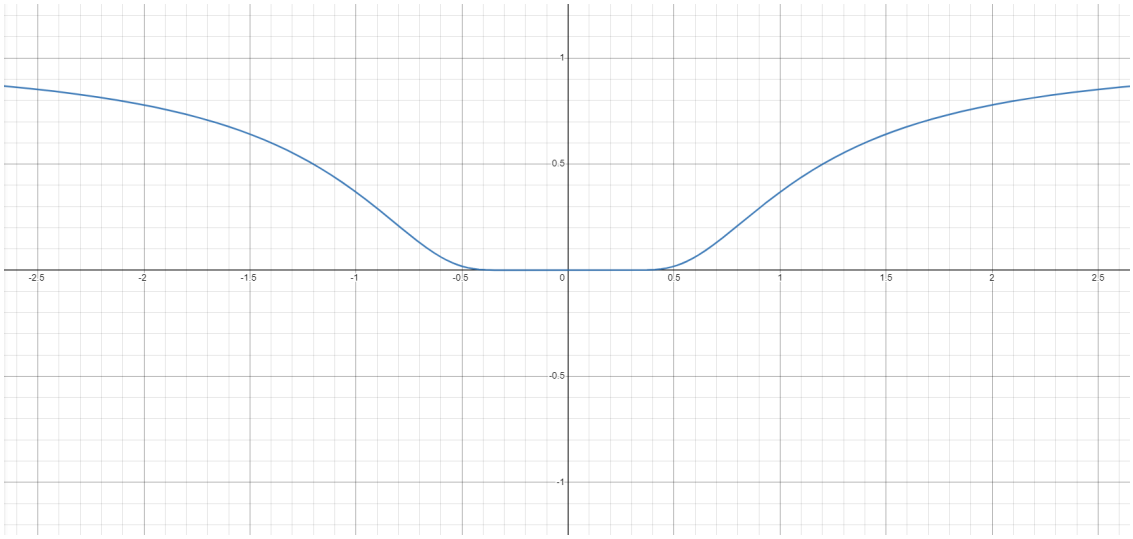

Figure 1: Plot of $f(x)$. This was generated using the Desmos browser software (desmos.com/calculator).

Problem 2

As we discussed in class, an important issue in comparing algorithms is the asymptotic order of convergence. This problems exposes several rates of convergence and examples concerning them. I suggest you also read the subsection "Rates of Convergence" of the Appendix A2 of the textbook (pages 619 - 620 in the 2006 edition). The following points refer to the textbook problem numbering.

Problem 2.13

Show that the sequence $x_k = 1/k$ is not Q-linearly convergent, though it does converge to zero. (This is called *sublinear convergence*.)

Solution:

Plugging in the sequence to the definition of Q-convergence, we get

$$\frac{1/(k+1)}{1/k} = \frac{k}{k+1} = 1 - \frac{1}{k+1},$$

where the limit $x^* = 0$. By definition also, we require the ratio to be bounded by a number $r \in (0, 1)$ for all k. Note the ratio found above limits to 1, for any $r \in (0, 1)$ we can always find a k such that the ratio is larger than r, thus $x_k = 1/k$ is not Q-linearly convergent.

Problem 2.14

Show that the sequence $x_k = 1 + (0.5)^{2^k}$ is Q-quadratically convergent to 1.

Solution:

If we consider the sequence x_k to be a summation of two sequences, namely $y_k = 1, z_k = (0.5)^{2^k}$, then y_k converges to 1. Note that z_k is a subsequence of the sequence $z'_k = (0.5)^k$, which was shown in class to limit to zero, thus z_k limits to zero. Since x_k is a summation of converging sequences, then its limit is just the sum of the limits of the individual sequences, so $x_k \to 1$ as $k \to \infty$.

Plugging this into the definition for Q-quadratic convergence, we see

$$\frac{|1 + (0.5)^{2^{(k+1)}} - 1|}{|1 + (0.5)^{2^k} - 1|^2} = \frac{(0.5)^{2^{(k+1)}}}{(0.5)^{2^{(k+1)}}} = 1.$$

Note that the bound need to just be a positive constant, so setting $M = 1$, we see that the sequence x_n is Q-quadratically convergent.

Problem 2.15

Does the sequence $x_k = 1/k!$ converge Q-superlinearly? Q-quadratically?

Solution:

It's easy to see $x_k \to 0$ as $k \to \infty$. Checking the sequential ratio gives

$$\frac{x_{k+1}}{x_k} = \frac{1/(k+1)!}{1/k!} = \frac{k!}{(k+1)!} = \frac{1}{k+1}$$

Therefore, the ratio limits to zero as $k \to \infty$ meaning x_k is Q-superlinearly convergent. For Q-quadratic convergence,

$$\frac{x_{k+1}}{x_k} = \frac{1/(k+1)!}{(1/k!)^2} = \frac{(k!)^2}{(k+1)!} = \frac{k!}{k+1}$$

This is not bounded above by any $M > 0$, so the sequence is not Q-quadratic convergent.

Problem 2.16

Consider the sequence x_k defined by

$$x_k = \begin{cases} (\frac{1}{4})^{2^k}, & \text{k even,} \\ (x_{k-1})/k, & \text{k odd.} \end{cases}$$

Is the sequence Q-superlinearly convergent? Q-quadratically convergent? R-quadratically convergent?

Solution:

It's easy to see that $x_k \to 0$ as $k \to \infty$. We just need to check each Q-convergence.

- Q-superlinear:

  For k even,

  $$\frac{x_{k+1}}{x_k} = \frac{x_k/(k+1)}{x_k} = \frac{1}{k+1}.$$

  For k odd,

  $$\frac{x_{k+1}}{x_k} = k\frac{x_{k+1}}{x_{k-1}} = k\frac{(0.25)^{2^{k+1}}}{(0.25)^{2^{k-1}}} = k\frac{(1/256)^{2^{k-1}}}{(1/4)^{2^{k-1}}}.$$

  Although the odd ratio is not as simple as the even ratio, we can see it goes to zero rapidly, as the numerator decreases faster then the denominator, the multiple of k does not change this much. Thus the sequence x_k is Q-superlinear.

- Q-quadratic:

  For k even,

  $$\frac{x_{k+1}}{(x_k)^2} = \frac{x_k/(k+1)}{(x_k)^2} = \frac{1}{x_k(k+1)}.$$

  Since the sequence limits to zero, there is no chance that there is a bound for the even sequences. Thus there is no need to check the odd sequences, since we cannot find a bound $M > 0$ for the even sequences.

- R-Quadratic:

  With inspiration form Problem 2.14, we notice that $y_k = (0.5)^{2^k} > x_k$ for all k. Note we don't have to worry about absolute values and limits since both sequences limit to zero. We can show this explicitly:

  For k even,

  $$x_k = (0.25)^{2^k} = (0.5)^{2\cdot 2^k} = (0.5)^{2^{(k+1)}} < (0.5)^{2^k} = y_k$$

  for k odd,

  $$x_k = \frac{(0.25)^{2^{(k-1)}}}{k} = \frac{(0.5)^{2\cdot 2^{(k-1)}}}{k} = \frac{(0.5)^{2^k}}{k} < (0.5)^{2^k} = y_k$$

We can then show that y_k is Q-quadratically convergent.

$$\frac{y_{k+1}}{y_k} = \frac{(0.5)^{2^{(k+1)}}}{((0.5)^{2^k})^2} = \frac{(0.5)^{2^{(k+1)}}}{(0.5)^{2^{(k+1)}}} = 1$$

Therefore, setting $M = 1$, we see that y_k is quadratically convergent, thus x_k is R-quadratic.

Problem 3

This problem shows an example of where Newton's method diverges.

Problem 3, part 1

Implement Newton's method. Allow the number of iterations to be a parameter.

Problem 3, part 2

Apply the method to the following funciton (Fenton's function);

$$f(x) = \frac{1}{10}\left[12 + x_1^2 + \frac{1 + x_2^2}{x_1^2} + \frac{x_1^2 x_2^2 + 100}{(x_1 x_2)^4}\right]$$

If you code the derivatives by hand, then implement the function (in Matlab) [f, g, H] = fentonfgH(x), where x is the point of the function, gradient, and Hessian evaluation.

Solution:

The reason I'm grouping the first two parts is that my implementation makes the two inseparable/missing information. This is my FIRST time writing code in MATLAB, so just to confirm everything was correct, I rewrote the code in Python. I am relying on scipy, sympy, and numpy packages to do the heavy lifting. My MATLAB code requires the Fixed-Point Designer package to implement back substitution. I will provide my code for Python first, since it's easier to read and digest, and then MATLAB. I don't know the usual structure of a MATLAB project, so I have structured it as a main function, which then runs the subsequent functions. The main function is the last code block shown in the list.

In [ ]:
```
import numpy as np
import sympy as sm
import scipy as sp

#For Plotting and prettyness
import matplotlib.pyplot as plt
from matplotlib import colors as color
import matplotlib
from mpl_toolkits.axes_grid1 import make_axes_locatable
```

In [ ]:
```
x1, x2 = sm.symbols('x1 x2')

class func(sm.Function):
    @classmethod
    def eval(cls, x1, x2):

        res = (1/10) * (12 + x1**2 + (1 + x2**2)/x1**2 + (x1**2 * x2**2 + 100) / (x1**4 * x2**4) )
        return res

#Computing function, gradient and hessian in sympy
f = func(x1, x2)
g = [sm.diff(f, k) for k in [x1, x2]]
H = sm.hessian(f, (x1, x2))
```

In [ ]:
```
def newtonseq(x0, k, stepsize) :

    def fentonfgH(x_eval) :
        f_eval = f.subs([(x1, x_eval[0]), (x2, x_eval[1])])
        g_eval = [g[i].subs([(x1, x_eval[0]), (x2, x_eval[1])]) for i in range(len(g))]
        H_eval = H.subs([(x1, x_eval[0]), (x2, x_eval[1])])

        return np.array(f_eval).astype(np.float64), np.array(g_eval).astype(np.float64), np.array(H_eval).astype(np.float64)

    newtonseq = np.zeros((k, 2))
    newtonseq[0, :] = x0

    for i in range(0, k - 1):
        _, g_step, H_step = fentonfgH(newtonseq[i, :])

        #Computing QR of H_step for inverse
        H_Q, H_R = np.linalg.qr(H_step)

        #Solving QRx = b => Rx = Q^T b
        b = H_Q.T @ g_step
        x = sp.linalg.solve_triangular(H_R, b)

        newtonseq[i+1, :] = newtonseq[i, :] - stepsize * x.T

    return newtonseq
```

In [ ]:
```
stepsize = 1
xk = newtonseq([3, 4], 20, stepsize)
fig, axs = plt.subplots(3, 1, figsize = (8, 10))

#I wanted a color gradient that goes between two dark colors.
#winter was the only one I could find.
cm = matplotlib.colormaps['winter']
norm = color.Normalize(vmin=0, vmax=len(xk[:, 1]))

titles = ['X', 'Y']
for i in [0, 1, 2]:
    if i != 2:
        axs[i].plot(xk[:, i], c = "#A23BEC")
        axs[i].set_xlabel("Step")
        axs[i].set_ylabel(f'{titles[i]} - Position')
        axs[i].set_title(f'{titles[i]} - Position of sequence, stepsize = {stepsize}')
    else :

        for j in range(len(xk[:, 0])):
            axs[i].scatter(xk[j, 0], xk[j, 1], c=[cm(norm(j))], zorder = 2)
        axs[i].plot(xk[:, 0], xk[:, 1], c="#A23BEC", linewidth=1.5, zorder = 1)
        axs[i].set_title(f'Convergence of Sequence, stepsize = {stepsize}')
        axs[i].set_xlabel("X - Position")
        axs[i].set_ylabel("Y - Position")
        axs[i].annotate('Initial', xy = (xk[0, 0], xk[0, 1]))
        axs[i].annotate('Final', xy = (xk[-1, 0], xk[-1, 1]))

    axs[i].set_facecolor("#e1e2e3")
    axs[i].grid(True)


divider = make_axes_locatable(axs[-1])
cax = divider.append_axes('right', size='5%', pad=0.05)
fig.tight_layout(pad = 0.3)
fig.colorbar(plt.cm.ScalarMappable(norm = norm, cmap = cm), ax = axs[-1], label = 'Step', cax=cax)
```

```
function [f, g, H] = fentonfgH(x)

    %Implementation of evaluating the fenton function at a given point
    %as well as its gradient and Hessian.

    %The parameter x should be an array of size 2.

    %Evaluating f at x
    func = @(x) (1/10) * (12 + x(1)^2 + (1 + x(2)^2)/x(1)^2 + (x(1)^2 *
x(2)^2 + 100) / (x(1)^4 * x(2)^4) );
    f = func(x);

    %Hard coding the gradient, and evaluating at x
    g1 = @(x) 0.2*x(1) - 0.2*(x(2)^2 + 1)/x(1)^3 + 0.2/(x(1)^3*x(2)^2) -
0.4*(x(1)^2*x(2)^2 + 100)/(x(1)^5*x(2)^4);
    g2 = @(x) 0.2*x(2)/x(1)^2 + 0.2/(x(1)^2*x(2)^3) - 0.4*(x(1)^2*x(2)^2 +
100)/(x(1)^4*x(2)^5);

    g = [g1(x), g2(x)];

    %Hard Coding the Hessian, then evaluating
    H11 = @(x)  0.2 + 0.6*(x(2)^2 + 1)/x(1)^4 - 1.4/(x(1)^4*x(2)^2) +
2.0*(x(1)^2*x(2)^2 + 100)/(x(1)^6*x(2)^4);
    H12 = @(x) -0.4*x(2)/x(1)^3 - 1.2/(x(1)^3*x(2)^3) + 1.6*(x(1)^2*x(2)^2 +
100)/(x(1)^5*x(2)^5);
    H21 = @(x) -0.4*x(2)/x(1)^3 - 1.2/(x(1)^3*x(2)^3) + 1.6*(x(1)^2*x(2)^2 +
100)/(x(1)^5*x(2)^5);
    H22 = @(x) 0.2/x(1)^2 - 1.4/(x(1)^2*x(2)^4) + 2.0*(x(1)^2*x(2)^2 + 100)/
(x(1)^4*x(2)^6);

    H = [
        H11(x), H12(x);
        H21(x), H22(x)
    ];
end
```

Published with MATLAB® R2023b

```
function newtonseq = newtonseq(x0, k, stepsize)
    %Funcition to calculate the newton's method sequence up to step k
    %of the initial conditions x0 given.

    %k should be an integer > 1
    %x0 should be a 1x2 array of real numbers

    % Initialize the array with zeros
    newtonseq = zeros(k, 2);

    newtonseq(1, :) = x0;

    % Populate the array element-wise
    for i = 1:k-1
        [~, g_step, H_step] = fentonfgH(newtonseq(i, :));

        % Computing QR of H_step for inverse
        [Q, R] = qr(H_step);

        % Solving QRx = b => Rx = Q^T b
        b = Q' * g_step';
        x = fixed.backwardSubstitute(R, b);

        newtonseq(i+1, :) = newtonseq(i, :) - stepsize * x';
    end
end
```

Published with MATLAB® R2023b

```
function plotting(seq, stepsize)

    numsteps = size(seq, 1);
    k = length(seq(:, 1));
    xaxis = 1:k;
    colorMap = turbo(numsteps);

    subplot(1, 3, 1);
    plot(xaxis, seq(:, 1) );
    xlabel('step'), ylabel('x - position'), title(strcat('X - Position of
sequence, stepsize =  ', num2str(stepsize)))
    grid on

    subplot(1, 3, 2);
    plot(xaxis, seq(:, 2) );
    xlabel('step'), ylabel('y - position') ,title(strcat('Y - Position of
sequence, stepsize =  ',  num2str(stepsize)))
    grid on

    subplot(1, 3, 3);
    % Plot each step with a changing color. Also a Solid line for reference
    hold on;
    plot(seq(:, 1), seq(:, 2), 'Color', 'k', 'LineWidth', 1.5);
    scatter(seq(:, 1), seq(:, 2), 50, colorMap, 'filled');
    hold off;

    clim([0, numsteps]);
    c = colorbar;
    c.Label.String = 'Step';
    xlabel('x - position'), ylabel('y - position'),
title(strcat('Convergence of Sequence, stepsize =  ',  num2str(stepsize)))
    grid on


end
```

Published with MATLAB® R2023b

```
function mainf(x, k, stepsize)

    %This will act as a pseudo main file to run everything
    %The inputs are the initial conditions and the number of steps in
    %the newton's method. This will plot the sequence in successive steps.
    %Only works since we're in 2D

    % A correct input for this code is:
    %   mainf([3, 4], 20, 1)

    % x - initial conditions
    %   Should be a 1x2 array
    % k - number of steps
    %   Should be an integer > 1

    clc
    seq = newtonseq(x, k, stepsize);

    plotting(seq, stepsize);

end
```

Published with MATLAB® R2023b

Problem 3, part 3

Initialize the method at x = [3, 2]. Describe what you observe (and quantify it, eg why you think you observe convergence or divergence, the result should be pretty obvious in a few iterations).

Solution:

I will provide the plots below, first in Python, then in MATLAB. Each implementation shows the same plots - the iteration in X, iteration in Y, then the iteration in $\mathbb{R}^2$. The Python code has these stacked vertically, whereas the MATLAB code shows these stacked horizontally. Note that in my initial approach, I included a stepsize to allow for a better approach to the minimum. I have set the stepsize in both to one for this reason. It seems the initial condition $x_0 = [3, 2]$ gives us as converging sequence, as we see it takes less than 10 iterations to get to the minimum, then stays there for the remaining. This is also reflected in the X and Y position plots flattening out after some time.

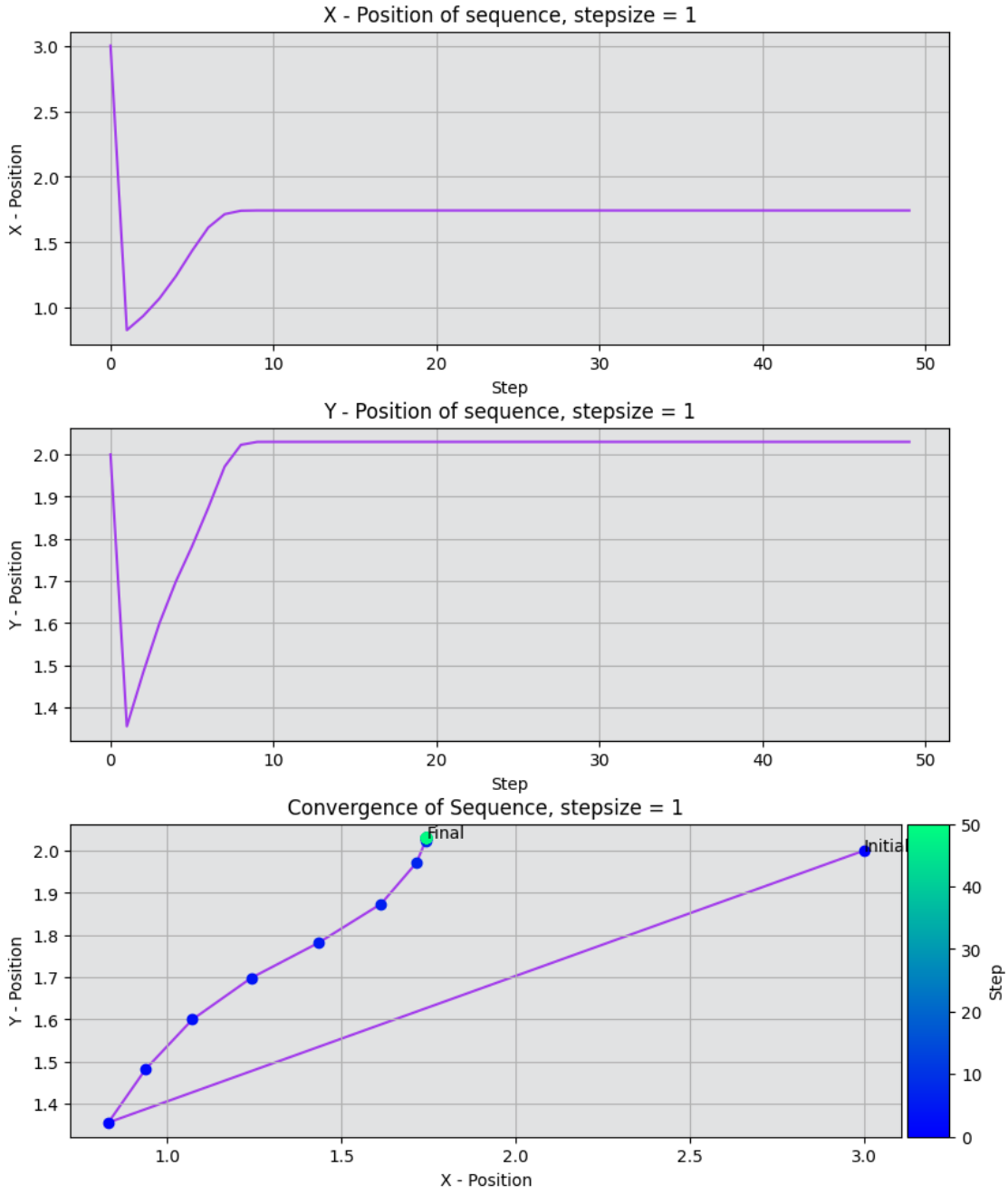

Figure 2: Showcase of Newton's Method for the Fenton Function in Python, with 50 steps at initial point [3, 2].

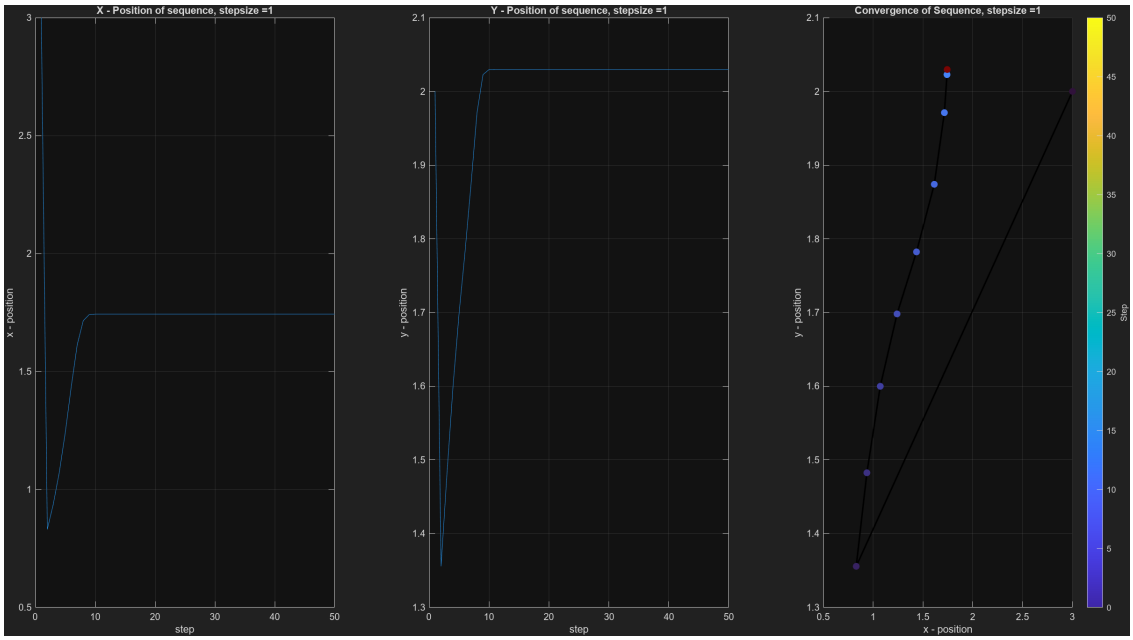

Figure 3: Showcase of Newton's Method for the Fenton Function in MATLAB, with 50 steps at initial point [3, 2]. Please excuse the darkness of this plot - I'm new to MATLAB and have yet found how to use dark mode AND have light mode plots.

Problem 3, part 4

Initialize the method at x = [3, 4]. Describe what you observe (and quantify it a bit as above).

Solution:

The plots are provided in the same manner as in the previous part. Here we see the opposite behavior of the previous part, since after some iteration both sequences in the X and Y coordinates diverge. However in Figure 6, with the inclusion of a stepsize = 0.01, we see a gradual approach to the minimum.

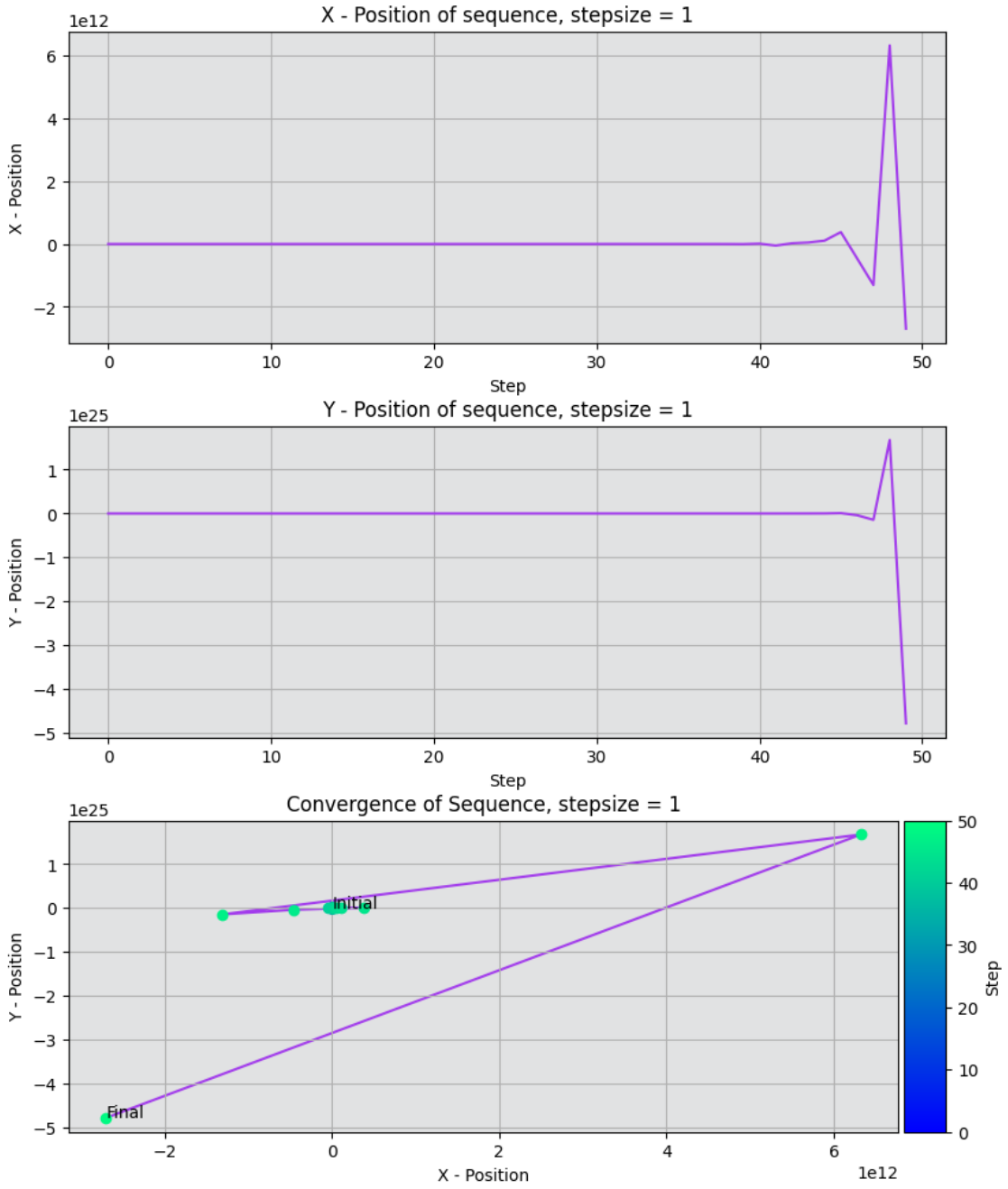

Figure 4: Showcase of Newton's Method for the Fenton Function in Python, with 50 steps at initial point [3, 4].

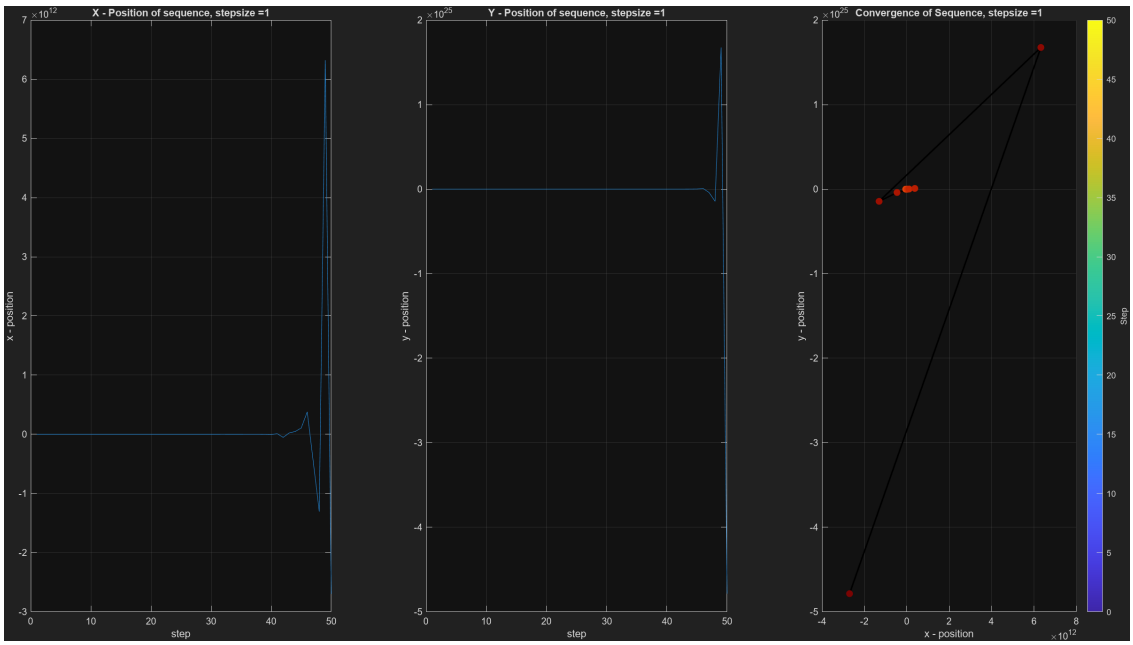

Figure 5: Showcase of Newton's Method for the Fenton Function in MATLAB, with 50 steps at initial point [3, 4]. Please excuse the darkness of this plot - I'm new to MATLAB and have yet found how to use dark mode AND have light mode plots.

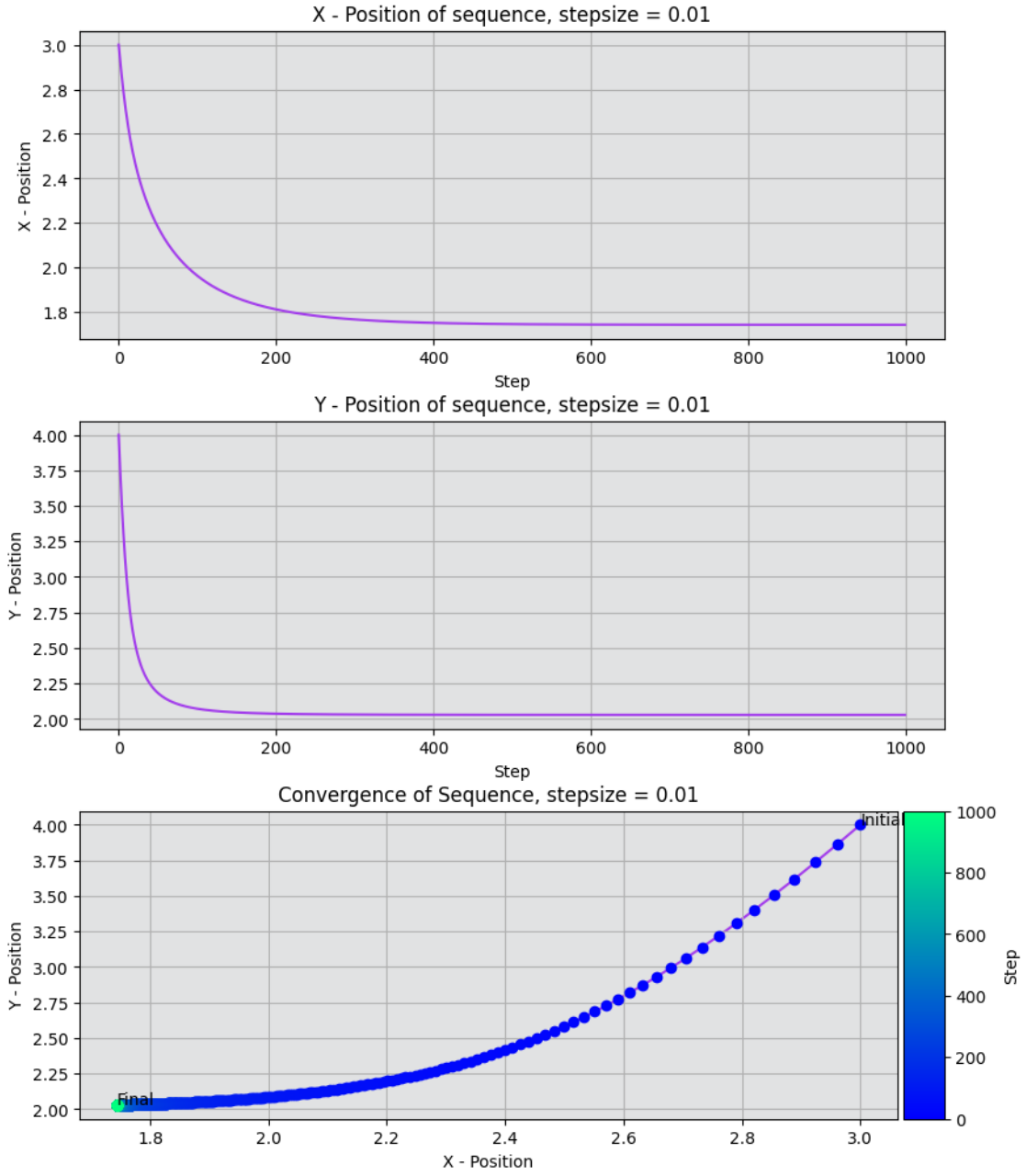

Figure 6: Introduction of a small stepsize, with many more iterations. This turns a diverging model into one which converges to a minimum.