MATH 458

Project #2

John Krone

In this write up I will walk you through relevant details of how I designed and modified the MATLAB script to perform Newton’s method for the given problem:

First, a bit of math. I wrote the above equation in complex form as the following:

Then, I framed this as a vector-valued function:

And set up the Newton’s method problem to find the values such that the function is equal to zero. Of course, we already know and can easily derive the roots of , so when designing the loop for the Newton’s method, I used a while loop with the condition to stop when the norm of the updated vector value was within a small value of the norm of one of the four roots of the given complex equation, which in vector values corollate with the following:

Additionally, I added a condition to the while loop to stop when the number of iterations it went through reached 100. The reasoning for this was to avoid infinite loops in the case that a certain point didn’t converge.

With my vector-valued function defined, it was easy to set up the rest of the problem. I defined the Jacobian and stored it as a function handle using preset MATLAB functions so that I could feed it inputs in the while loop for Newton’s method.

As you will see in my code, I created a new function file to run Newton’s method and to count the number of iterations each point required to achieve the convergence criteria or make the loop stop by maxing out the iterations. I then called this function (called “newton”) from my main script where I had defined the vector-valued function and its Jacobian. I also plotted the results in the main script.

I designed the “newton” method to count iterations so I could explore convergence rates. I ran into a key issue. All along both diagonals of the graph the points did not converge to one of the four known roots after 100 iterations of Newton’s method. In Figure 1, you can see a screenshot of the upper left corner of the grid I used to plot the iteration map (final version shown in Figure 7). The iteration map is a contour plot that shows how many iterations each point required achieve the convergence criteria or make the loop stop by maxing out the iterations. I designed it to be centered at the origin and with a half-width of 2. Then I split it up into a 1001-by-1001 grid of points within [-2,2] on the x-axis and [-2,2] on the y-axis. As a result, when you look at the grid in Figure 1, the upper leftmost element correlates with the point and the element to its right correlates with the point . Further, if you pick a random point in the grid, say the point with coordinates {5,6}, then its element represents the number of iterations corresponding to the point in the complex plane. You can see that all along the diagonal elements the number of iterations required is 100. This means that Newton’s method was either unsuccessful (in terms of meeting the convergence criteria) on these values or was successful in exactly 100 iterations.

A screenshot of a cell phone

Description automatically generated

Figure : A visual representation of the array variable “iterVec”. It shows at all the diagonal points that the Newton's method required 100 iterations in the aforementioned setup.

Further, you can see in Figure 2 a screenshot of the upper left of the grid I used to plot the root map (Figure 3). The coordinates of this grid correspond exactly to the complex plane as those in Figure 1 do. By examination of the diagonal elements we see that none of them converged to one of the four known roots. In other words, they did not converge. This negates the possibility that any of the points met the convergence criteria in exactly 100 iterations.

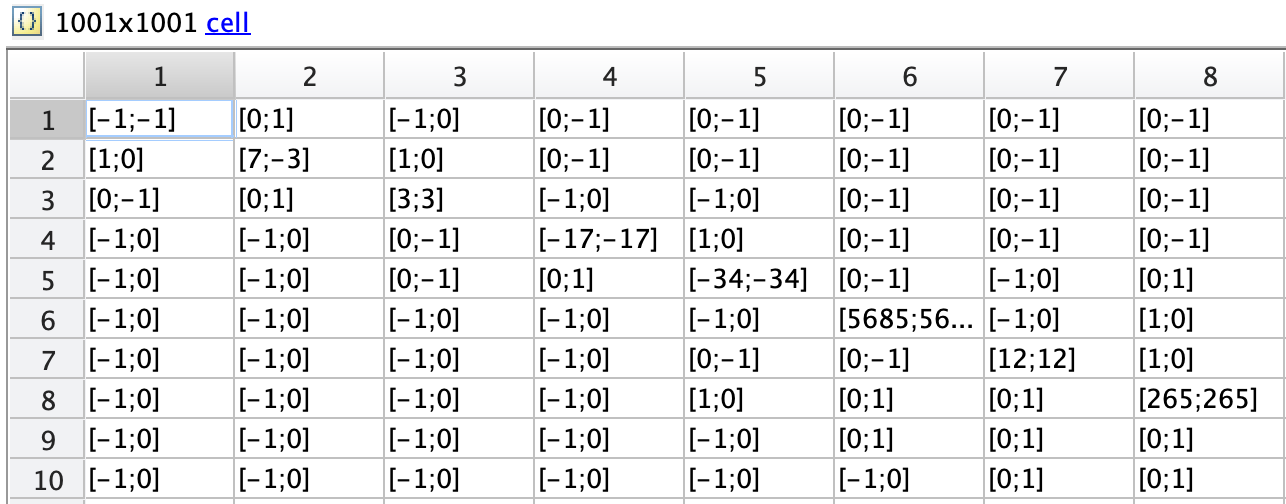


Figure : A visual representation of the array variable "roots". It shows that none of the diagonal elements converged, while all the nondiagonal elements did.

To more closely examine this issue, I plotted all values that failed to converge by Newton’s method in 100 iterations in black. In Figure 3, you can see how these problem elements manifested in visual terms. I zoomed in on the origin, and you can see two diagonal lines with black dots along them where the method did not converge. Of course, there is only discrete points in black in this case because I created the map with finite number of elements. Because I plotted it as a contour map (using contourf() function) the space around the black points filled in to match the color of the nearest most points (in this case blue). One could imagine that you are looking down the z-axis at a surface.

A close up of a logo

Description automatically generated

Figure : A close-up of the origin of the convergence of Newton's Method.

Next, I decided that if I could show that the diagonal elements caused the Jacobian matrix to be defective then I would understand why they didn’t converge. Unfortunately, the diagonal elements caused no issues for the Jacobian. For example, the following lines of code evaluate the value of the Jacobian for the starting point of . I tested many other diagonal elements that all yielded results of a similar form. Clearly this matrix is invertible.

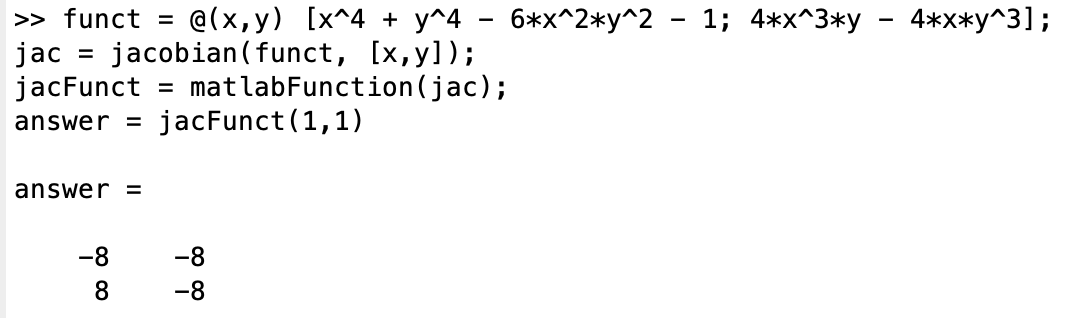


Figure : An example of the Jacobian matrix of one of the diagonal elements

Next I began to directly derive the potential conditions in which the Jacobian matrix would be non-invertible. The Jacobian was defined as such:

And I knew that 2-by-2 matrices with one zero element and 3 nonzero elements were defective. Setting each element equal to zero one by one you solve for only two equations:

Unfortunately, these equations do not have anything to do with the diagonal elements, so they were not useful. Furthermore, the above logic actually has a flaw. Given the Jacobian’s nearly symmetric behavior, it is impossible to have only one element be equal to zero. If the bottom left element is equal to zero for a non-trivial reason then the top right is also, and the matrix is simply a scalar multiple of the identity matrix.

After all these dead-ends I finally solved the problem with the simplest hypothesis of all: the diagonal elements simply require more than 100 iterations to converge. So, I increased the number of iterations in the Newton’s method from 100 to 500. In Figures 5 and 6 you can see the upper left corners of the iteration grid and the root grid once again, but with the new implementation of maximum 500 iterations. As you can see, all the diagonal elements have some iteration number less than 500 and have roots equal to one of the four known roots.

A screenshot of a cell phone

Description automatically generated

Figure : The "IterVec" array with the new implementation.

A close up of a telephone

Description automatically generated

Figure : The "roots" array with the new implementation.

Intuitively, it makes sense that the diagonal elements take a relatively long time to converge. They sit equidistant from two of the four roots of the problem, so we wouldn’t expect them to converge quickly and to an obvious root, like we would with the point , for example. In the following Figures 7 and 8 you can examine the visual representations of the number of iterations each point required for convergence and the roots that they solved to using Newton’s method.

A close up of a logo

Description automatically generated

Figure : A map of the number of number of iterations required to achieve convergence. In this case, only the origin point did not achieve convergence. Were the map appears black is actually due to the fact that convergence rate is changing rapidly over a short distance, thus causing the level sets of the contour map to bunch up and obscure the true colors. Note that in the colorbar on the right I only displayed the colors for iteration values between 0 and 50, since the overwhelming majority of points in this 1001-by-1001 grid converge within 50 iterations.

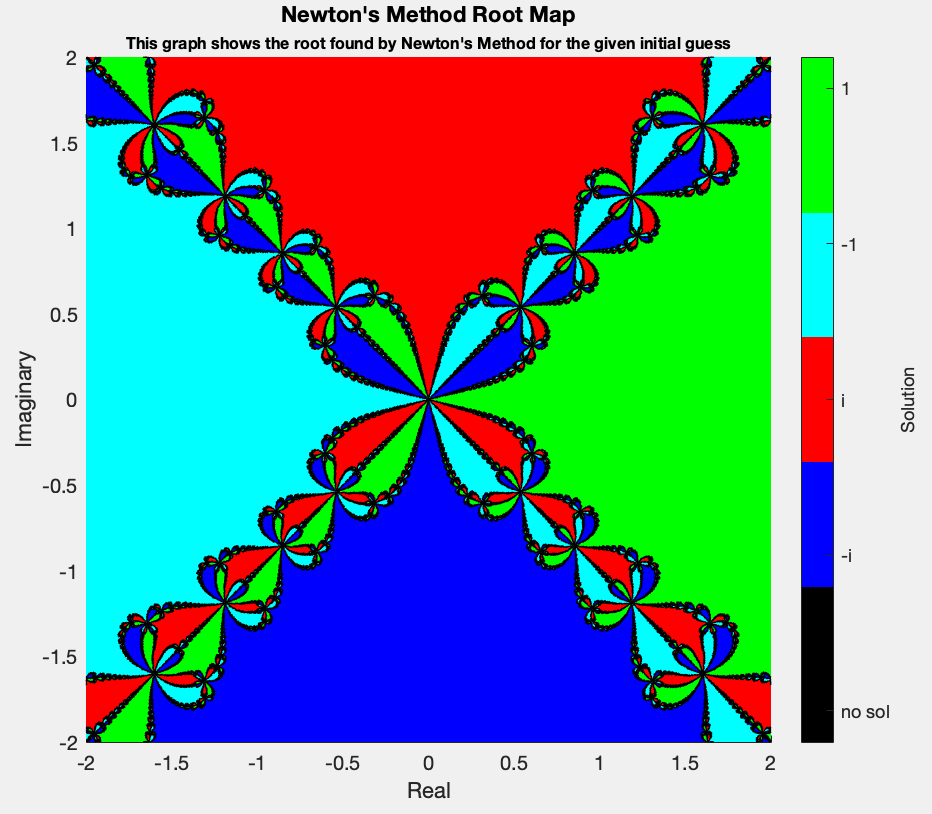


Figure : Note that the only point in this map that has no solution is the origin point, so there are no black regions (no solution regions) on the map at this scale. The areas where the root changes rapidly may appear black from this scale, but it is only due to the fact that there are so many contour lines clustered together in those areas that it appears so. Figure 10 show in more detail that when you zoom in on this map there are no black points except at the origin. Separate from the main script below I ran a loop to verify that none of the points in the 1001x1001 have no solution except the origin point.

Last but not least, in Figures 9 and 10, I have zoomed in on the elements around the origins of Figures 7 and 8, respectively. In Figure 9 you can see how the rate of convergence changes quickly around the origin and around the diagonal elements and creates some interesting patterns. In Figure 10 you can see that in fact all elements converge (i.e. the color black is nowhere to be found except at the origin).

A close up of a map

Description automatically generatedA picture containing text, screenshot

Description automatically generated

Figure 9: Left: A zoom in of the origin of Figure 7. Right: A zoom in on the upper right of the Left image, centered on the diagonal. Note that I changed the axis limits of the colorbar in this figure to illustrate points with a much higher iteration number than most other elements.

A close up of text on a white background

Description automatically generated

Figure : A close up of Figure 8 at the origin.

2019/11/14 20:58: I just submitted this project a few minutes ago, but then started playing around with the iteration map at different scales and discovered some very interesting fractal patterns, so I’m including them here as well. There seems to be two different fractal patterns. One that exists at a scale of large than order 10^0. And Another that exists at scale of less than order 10^0. I’m showing these maps in order of decreasing scale. They are all the same exact square map but with a different width. On the last few you can observe how the iterations increases rapidly as you approach the origin.

A close up of text on a white background

Description automatically generatedA close up of a logo

Description automatically generated

A close up of a map

Description automatically generatedA close up of a logo

Description automatically generated

A close up of a map

Description automatically generatedA close up of a logo

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**SCRIPT**

%{

John Krone

MATH 458 Fall 2019

Project 2 Modified

jkrone@usc.edu

%}

clear; clc;

%% solve the equation z^4 - 1 in complex coordinates using Newton's method

% first set up the problem as a vector-valued function

syms x y;

funct = @(x,y) [x^4 + y^4 - 6\*x^2\*y^2 - 1; 4\*x^3\*y - 4\*x\*y^3];

% find the jacobian and inverse jacobian

jac = jacobian(funct, [x,y]);

jacFunct = matlabFunction(jac);

% find results for a grid of guesses

n = 1000;

d = 2.0;

tol = 0.01;

xVec = (-1:2/n:1)\*d;

yVec = (-1:2/n:1)\*d;

[x,y] = meshgrid(xVec,yVec);

iterVec = zeros(size(x));

roots = cell(size(x));

for k = 1:length(x(:))

guess = [x(k);y(k)];

[roots{k},iterVec(k)] = newtonMod(funct,jacFunct,guess,tol);

end

% Plot iteration results

a1 = figure;

hold on

levels = 0:5:100;

contourf(xVec,yVec,iterVec,levels)

c1 = colorbar;

c1.Label.String = 'Iterations required for convergence';

c1.Limits = [1,50];

colormap(hsv);

axis square

xlabel('Real')

ylabel('Imaginary')

title('Convergence rate countour map')

hold off

% Plot root results. This requires some modification to roots var first

% Here I will give all the roots a value so they plot on a countour map

% value 1 is root 1, 2 is root -1, 3 is root i, 4 is root -i, 5 is no sol

rootPlotVals = zeros(size(roots));

for index = 1:length(rootPlotVals(:))

if roots{index} == [1;0]

rootPlotVals(index) = 1;

elseif roots{index} == [-1;0]

rootPlotVals(index) = 2;

elseif roots{index} == [0;1]

rootPlotVals(index) = 3;

elseif roots{index} == [0;-1]

rootPlotVals(index) = 4;

else

rootPlotVals(index) = 5;

end

end

b1 = figure;

hold on

contourf(xVec,yVec,rootPlotVals)

c2 = colorbar('Direction','reverse');

c2.Ticks = linspace(1,5,5);

c2.TickLabels = {'1','-1','i','-i','no sol'};

c2.Label.String = 'Solution';

c2.Limits = [0.8,5.2];

map = [0 1 0; 0 1 1; 1 0 0; 0 0 1; 0 0 0];

c3 = colormap(map);

axis square

xlabel('Real')

ylabel('Imaginary')

title({'Newton''s Method Root Map', '\fontsize{8}This graph shows the’... ‘root found by Newton''s Method for the given initial guess'})

hold off

**FUNCTION FILE**

function [root,iter] = newtonMod(funct,jacFunct,guess,tol)

%NEWTONMOD [root,ind] = newton(funct,jacFunct,guess,tol).

% Newton's method for some function "funct" and Jacobian

% "jacFunct" at some initial point "guess". Returns the root and the

% number of iterations i for convergence.

iter = 0;

updVec = guess;

% I added the last while condition to avoid an infinite loop

while norm(updVec-[1;0]) > tol && norm(updVec-[-1;0]) > tol &&...

norm(updVec-[0;1]) > tol && norm(updVec-[0;-1]) > tol ...

&& iter < 500

iter = iter+1;

updVec = -(jacFunct(updVec(1),updVec(2))) \ funct(updVec(1),updVec(2)) + updVec;

end

root = round(updVec);

end