MATH426 LAB3

Shane Cincotta

November 8, 2019

1 Abstract

The purpose of this lab is to design an algorithm to find as many points of intersection as possible between a helix defined parametrically for all t as:

$x(t) = \cos(2\pi t)$

$y(t) = \sin(2\pi t)$

$z(t) = t$

and the plane:

$Ax + By + Cz = 0$

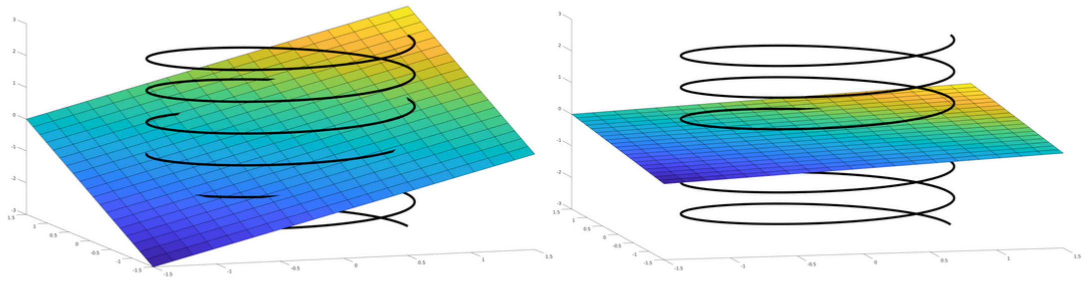
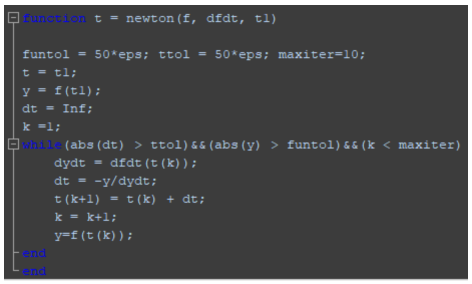


Figure 1: Here are two examples of what can happen: (A=B=1, C=-1) and (A=1/5, B=1/5, C=-1).

The function accepts 3 arguments, A, B and C and returns a list of values of t where the plane intersects the helix

2 Detailed Strategy

For the root finding algorithm, I decided to use a modified version of the Newton Method. To begin, I first implemented the Newton Method without any modifications. This was achieved using my implementation in the prelab.



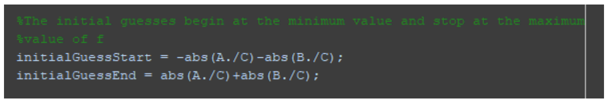
```
function t = newton(f, dfdt, t1)

funtol = 50*eps; ttol = 50*eps; maxiter=10;
t = t1;
y = f(t1);
dt = Inf;
k =1;
while(abs(dt) > ttol)&&(abs(y) > funtol)&&(k < maxiter)
    dydt = dfdt(t(k));
    dt = -y/dydt;
    t(k+1) = t(k) + dt;
    k = k+1;
    y=f(t(k));
end
end
```

Figure 2: My implementation of the Newton Method as used in HW7

One of the drawbacks of the Newton Method is that it requires an initial guess, thus providing a good guess of the root is crucial to the Newton Method. This is trivial if we can look at a graph and derive an educated guess, but a computer cannot view a graph, thus we must devise a way to pick good initial guesses. Another drawback of the Newton Method is the inability to find multiple roots.

The Newton Method stops after converging to the first root that is found. To circumvent this, I will pick multiple initial guesses and run the Newton Method at each of these guesses.

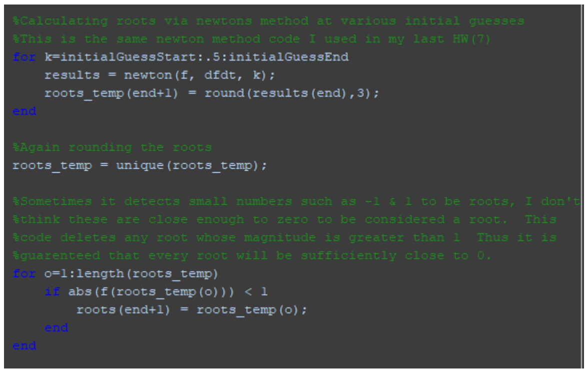
I then created bounds for the initial guess, which I will iterate through later. The start of the initial guess is based off the minimum value of the function. Since the function is made of trigonometric functions (which have maximum magnitudes of 1), we can derive the minimum and maximum possible values of the functions depending on the values of the coefficients A,B and C.



```
%The initial guesses begin at the minimum value and stop at the maximum
%value of f
initialGuessStart = -abs(A./C)-abs(B./C);
initialGuessEnd = abs(A./C)+abs(B./C);
```

Figure 3: Code to calculate the initial guess start and end

I then iterated through the initial guesses, using the Newton Method to calculate the root at each initial guess. The step between each initial guess is 0.5, this was done to assure that a root was not skipped over.



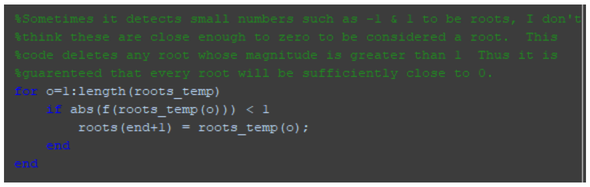
```
%Calculating roots via newtons method at various initial guesses
%This is the same newton method code I used in my last HW(7)
for k=initialGuessStart:.5:initialGuessEnd
    results = newton(f, dfdt, k);
    roots_temp(end+1) = round(results(end),3);
end

%Again rounding the roots
roots_temp = unique(roots_temp);

%Sometimes it detects small numbers such as -1 & 1 to be roots, I don't
%think these are close enough to zero to be considered a root.  This
%code deletes any root whose magnitude is greater than 1  Thus it is
%guarenteed that every root will be sufficiently close to 0.
for o=1:length(roots_temp)
    if abs(f(roots_temp(o))) < 1
        roots(end+1) = roots_temp(o);
    end
end
```

Figure 4: Iterating through each initial guess

3 Results

Throughout this lab I encountered a few challenges. The first challenge was encountered while trying to guarantee that no duplicate roots were included in the roots. I initially planned on using the unique method to remove duplicates, but after using this method I found that there were still some duplicate roots. I investigated using the Matlab workspace by observing the values stored in roots. I observed that some values were indeed duplicates, yet they were not removed by the unique function. This led me to believe that the resulting roots were calculated using more significant figures than Matlab is able to display. That is, the root values are different, but the difference is so small that Matlab can't show the difference in the workspace screen. To circumvent this, I rounded the roots and then called the unique method. After doing this, there were no duplicate roots.

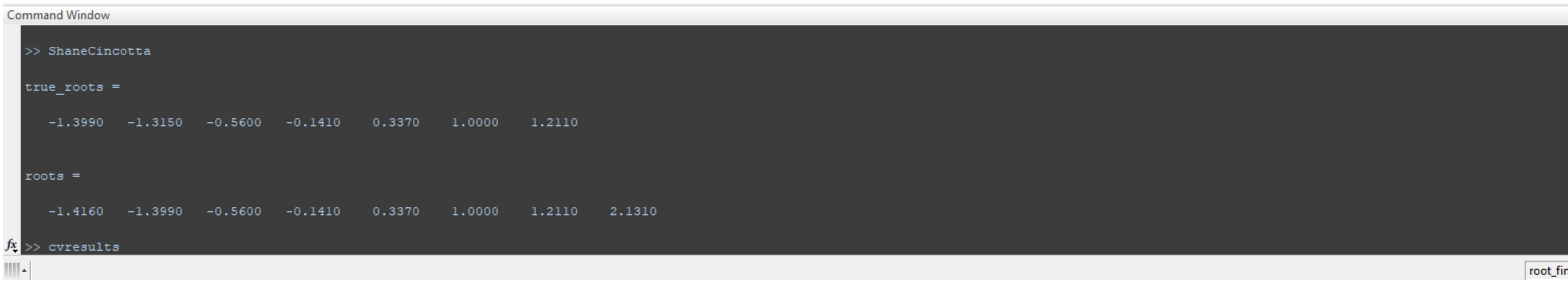
Another challenge I encountered was small (yet non zero) numbers being included in the roots. For example, sometimes my implementation would count a small number, say 1, to be a root. While this may be close to 0 and thus close to being a root, in my opinion, it is not close enough to 0 to be considered a root. To delete these "pseudo roots" I checked the value of the function at each root, and if the magnitude of the output value was less than 1 (my criteria for being close enough to be considered a root), I included that root in my root list. This could be circumvented further by picking better tolerance values within the Newton Method.



```
%Sometimes it detects small numbers such as -1 & 1 to be roots, I don't
%think these are close enough to zero to be considered a root.  This
%code deletes any root whose magnitude is greater than 1  Thus it is
%guarenteed that every root will be sufficiently close to 0.
for o=1:length(roots_temp)
    if abs(f(roots_temp(o))) < 1
        roots(end+1) = roots_temp(o);
    end
end
```

Figure 5: Iterating through each initial guess

After these challenges were overcome, I was ready to test my results. I tested my results against the roots that were provided by using the built in Matlab function, fzero. I tested them using two sets of coefficient values, $1, 1, -1$ and $\frac{1}{5}, \frac{1}{5}, -1$. I first began by testing on the first set of coefficient values $(1, 1, -1)$.

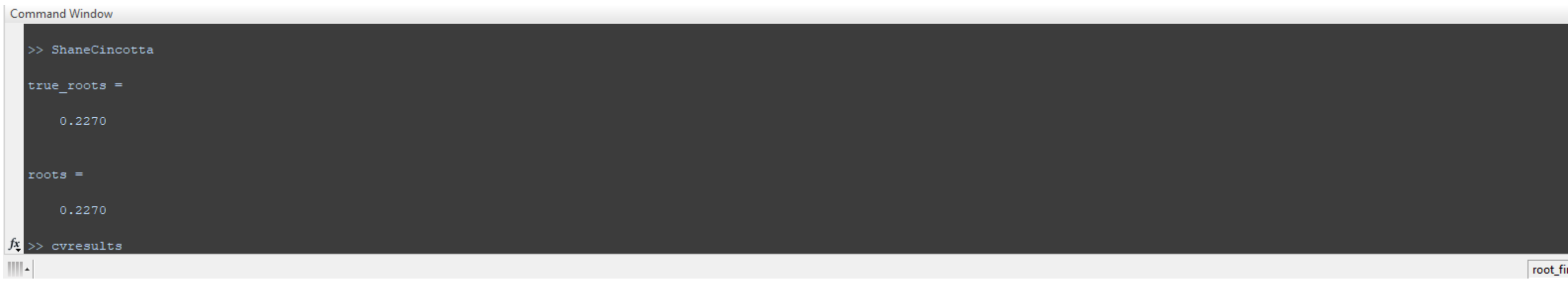


```
Command Window
>> ShaneCincotta

true_roots =

   -1.3990   -1.3150   -0.5600   -0.1410    0.3370    1.0000    1.2110


roots =

   -1.4160   -1.3990   -0.5600   -0.1410    0.3370    1.0000    1.2110    2.1310

>> cvresults
```

Figure 6: Testing with the first set of coefficients

The true_roots values were the roots calculated from using the fzero function. The roots values were the roots calculated using my method. As you can see, my algorithim is able to detect a root that fzero was not (2.1310). I manually computed the value of the function at this supposed root and it is indeed a root. The value of the function is -0.7, which meets my criteria for a root ($\leq |1|$). Despite this initial success, my method was not able to detect the root at -1.3150, this might be due to my step between the initial guess start and my initial guess end being to large and skipping over this root. After trying different step increments, I still was not able to find the root at -1.3150 (this is indeed a root, I calculated it manually).

I then tested my method on the second set of coefficient values ($\frac{1}{5}, \frac{1}{5}, -1$).



```
Command Window
>> ShaneCincotta

true_roots =

    0.2270


roots =

    0.2270

>> cvresults
```

Figure 7: Testing with the second set of coefficients

As you can observe, there was only one root of this function and my method was able to find it (as was fzero).

4 Conclusion

To recapitulate, I was able to create my own root finding method by making modifications to the Newton Method. While my method was not able to find all of the roots as compared to a method like fzero, it was able to find most of the roots and in some cases find roots that fzero could not find. The most prominent limitations of this method appear to be the step between each initial guess (can lead to skipping over of roots) as well as the tolerance of what constitutes a root.