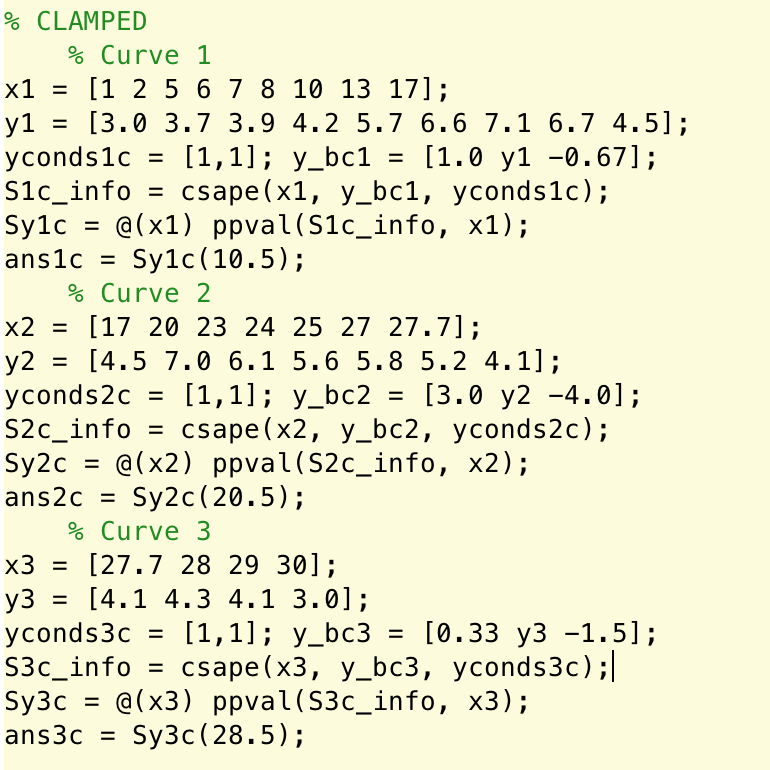
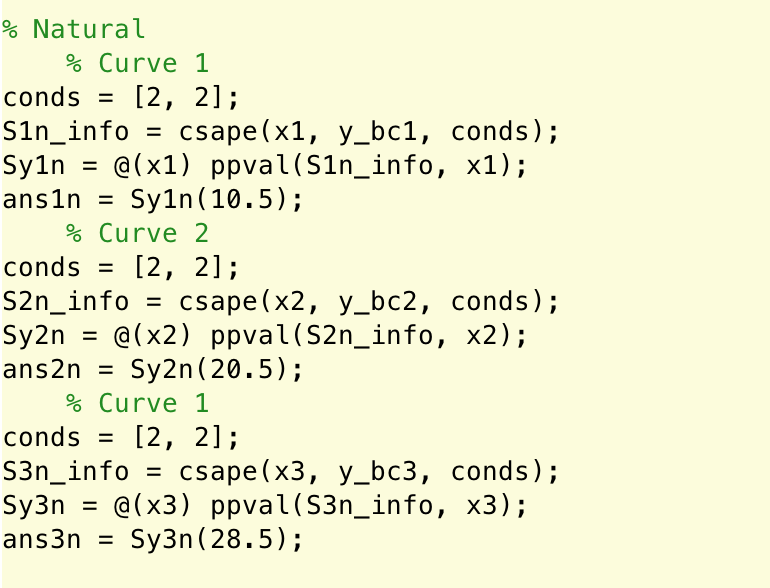
Final Project

1. Top Profile of the Noble Beast
   1. Cubic Spline Interpolation for natural and clamped

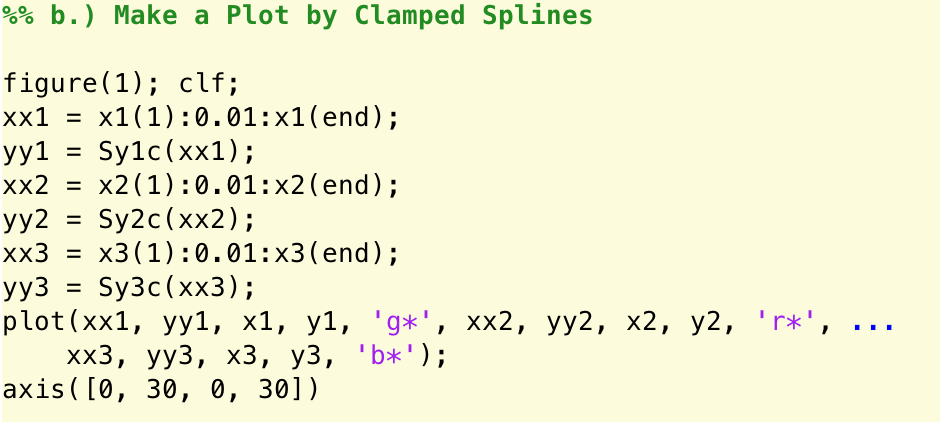


The clamped and natural condition cubic splines are essentially different only by

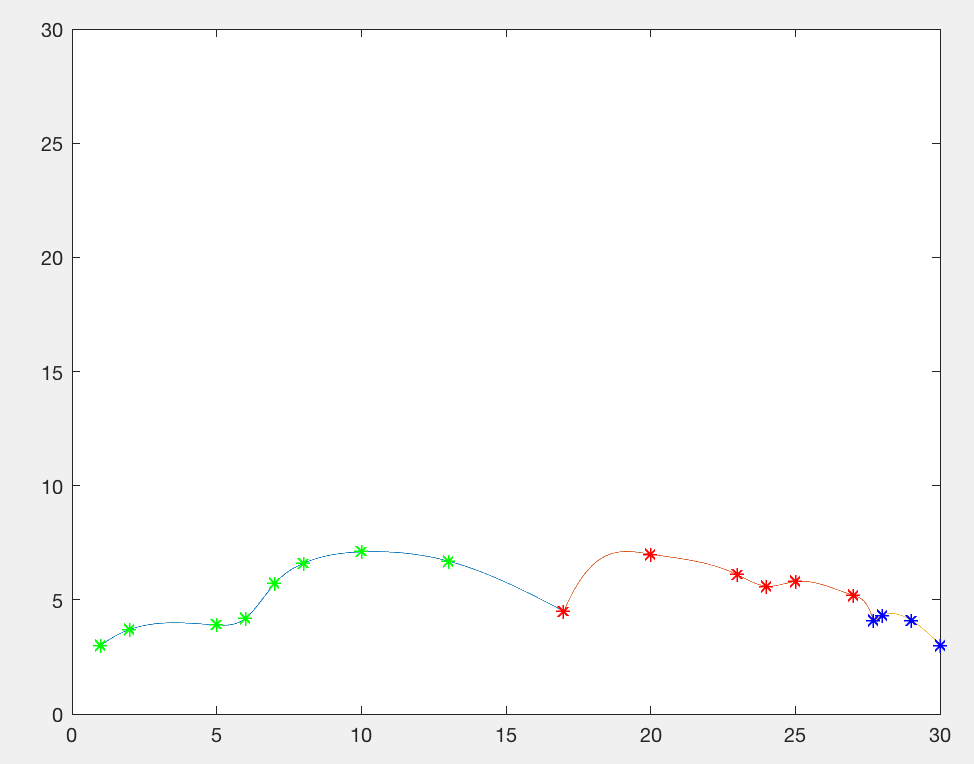
The conditions used for the boundaries. The solutions were very similar though, verifying that the interpolations were done correctly and similarly as they should be.

|  |  |  |
| --- | --- | --- |
| X | Clamped | Natural |
| 10.5 | 7.110678197163165e+00 | 7.086459002427792e+00 |
| 20.5 | 6.906883901298629e+00 | 7.319374995844806e+00 |
| 28.5 | 4.378341584158416e+00 | 4.347187500000000e+00 |

* 1. Make a plot using the clamped cubic splines



The clamped splines were calculated from beginning to end of the x, y, and z intervals. The plots contain the three curves for the noble beast with the interpolated data points highlighted as larger points. The color represents each of the curves 1, 2, and 3.

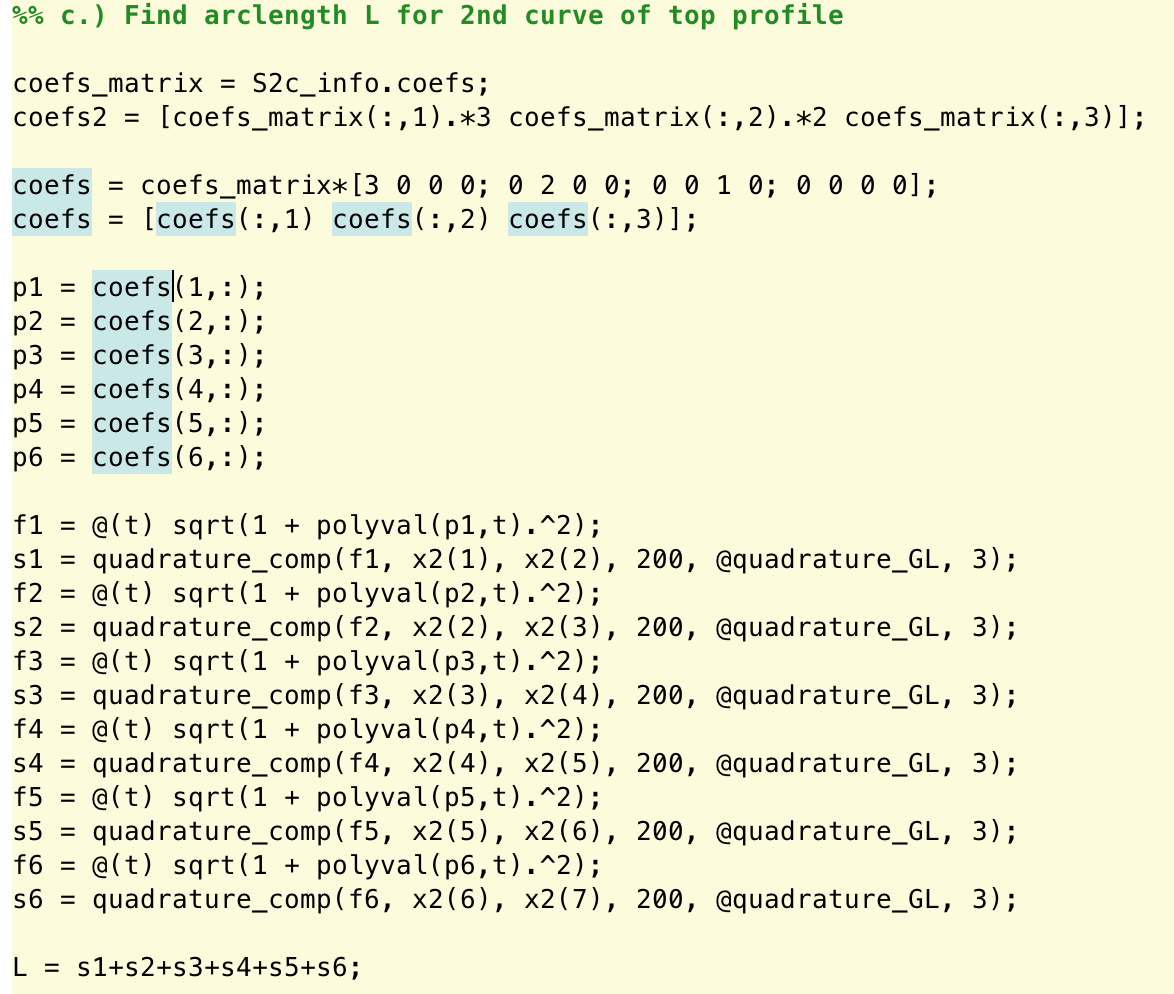


* 1. Find length L for the 2nd curve on the top of the noble beast

Using a matrix of coefficients from each piece of the cubic splines, I applied a matrix multiplication to get the derivatives of their polynomials and stored the coefficients.

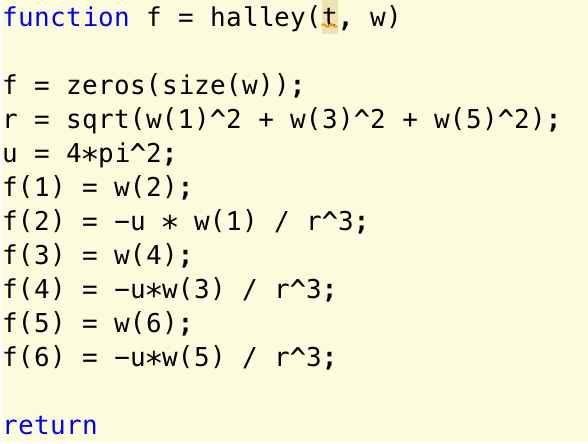
I used these coefficients with the arc length formula for non-parametric curves and used the quadrature\_comp function to evaluate the integrals.

I got L = 5.780410276667653e+03 which is very apparently wrong, but could not find the error.

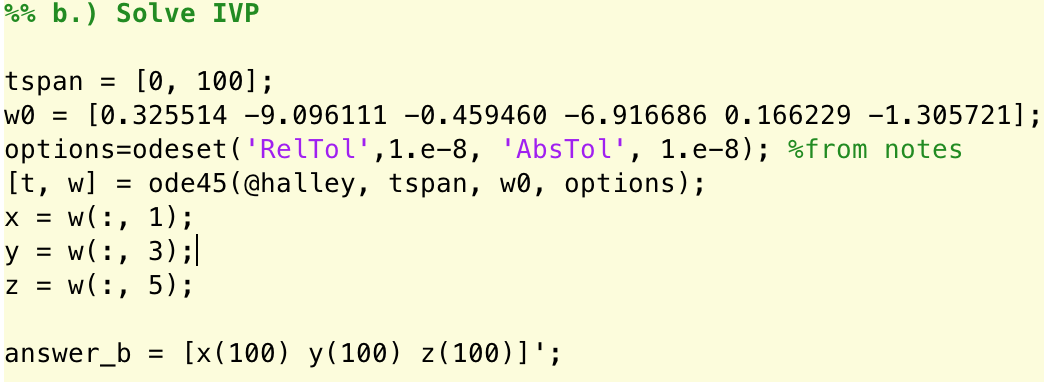


1. **Halley Comet**
   1. **MATLAB function for Halley ODE**

I reduced the 2nd order IVP ODE into that of a 1st order ODE. This was done to setup the problem so that it could be solved using numerical methods that we had learned in class.

****

* 1. **Solving the IVP**

****

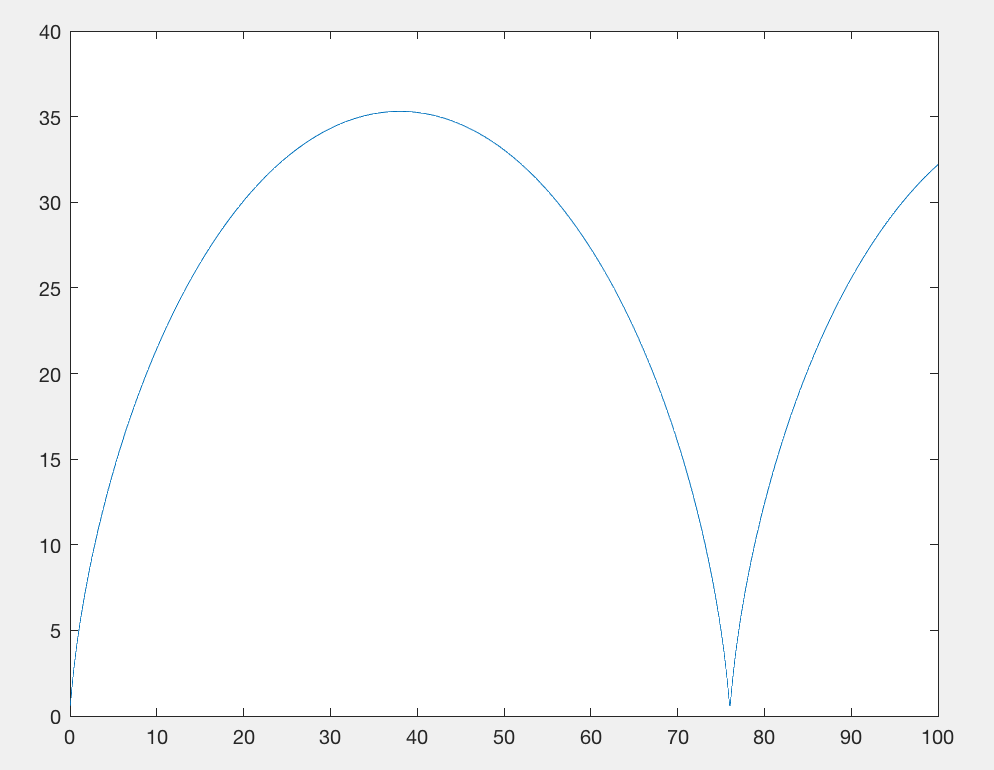
Using substitutions for the derivatives and variables as w, I used the ode built-in MATLAB function to find the values for t and w. This gave the solutions for our equations. I then found the 100th element of these vectors x, y, and z.

|  |  |
| --- | --- |
| X(100) | -7.569302605946165e-01 |
| Y(100) | -7.138455932059070e-01 |
| Z(100) | -8.528220727919801e-02 |

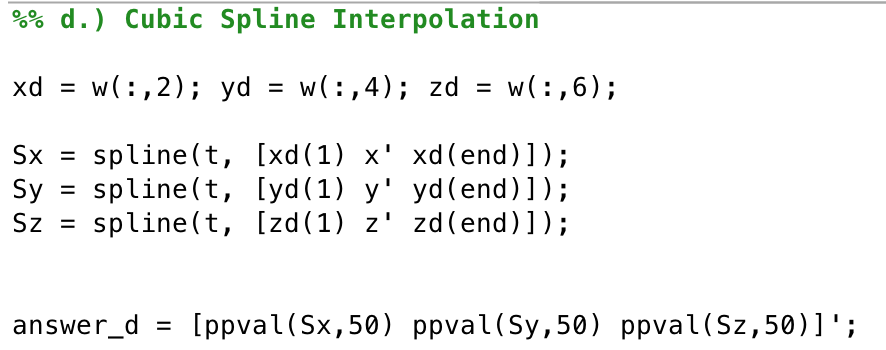
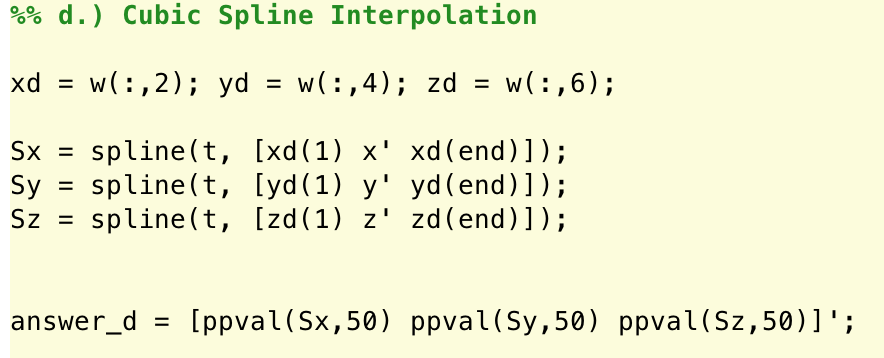
This method is good because we used the built-in functions that are created to be accurate. The solutions follow the structure of the in-class power point examples.

* 1. **Plot and find the closest point to the sun**

Since I had found the values for x, y, and z, I was able to create a vector r using these values and plot them from t = [0, 100].



* 1. **Use Cubic Splines with clamped conditions**

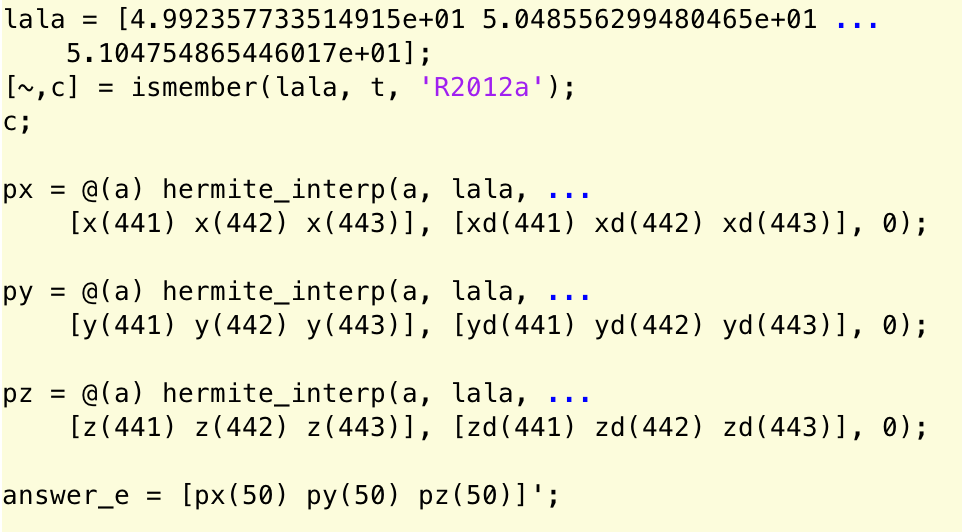
****

This problem was solved using the x,y, z derivatives stored in w to use for endpoints in the cubic spline interpolation. The parameters for the spline built-in function uses clamped conditions, which were evaluated at the specified points.

|  |  |  |  |
| --- | --- | --- | --- |
| t | X | Y | Z |
| 50 | -1.651338528551362e+01 | 2.715794088541071e+01 | -9.083508021479046e+00 |

This formatting was found through “help spline” which specified the parameter arguments that would specify a clamped condition cubic spline interpolation

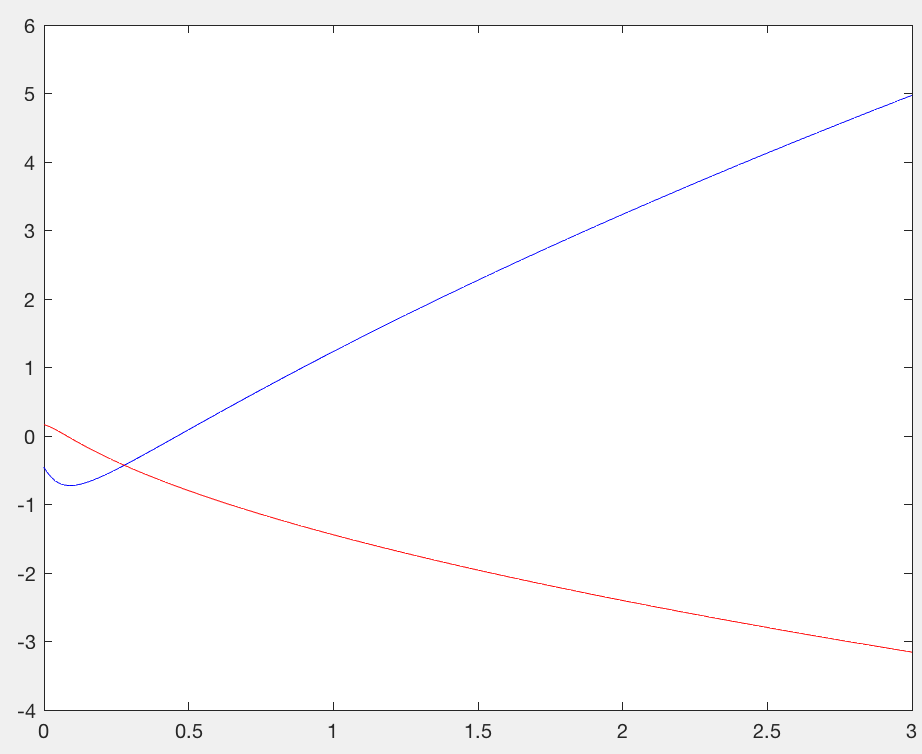
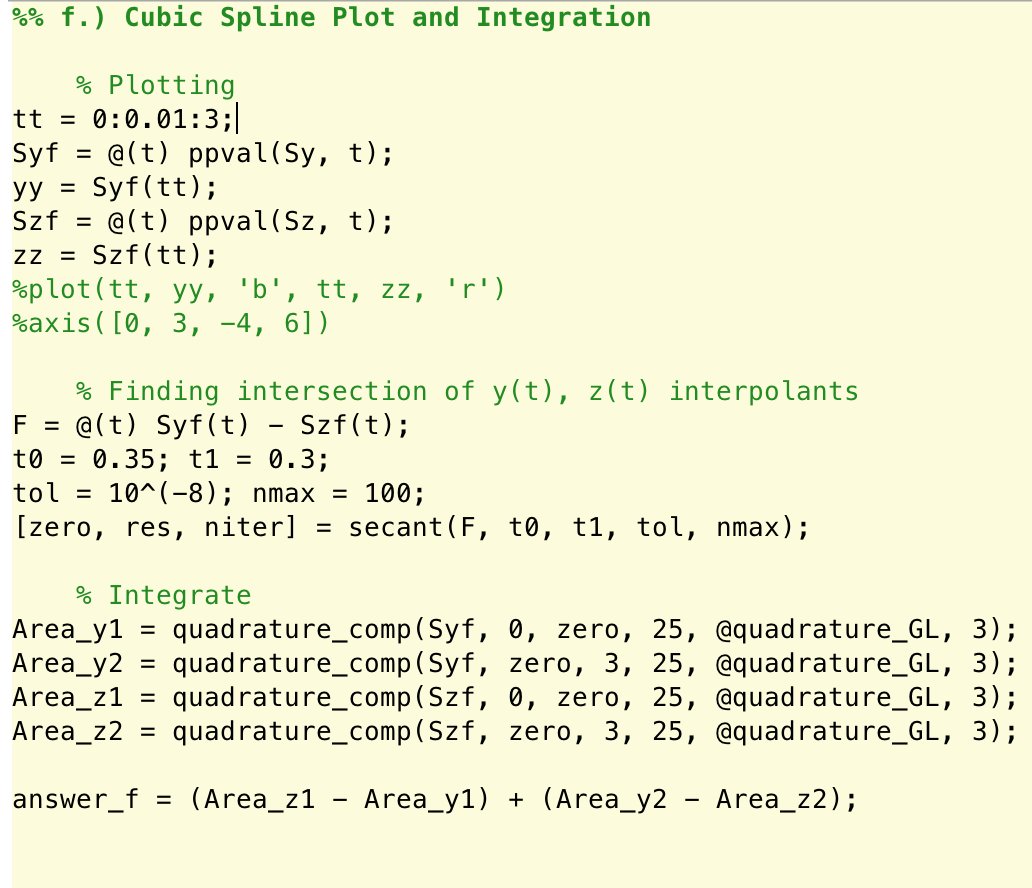
* 1. **Cubic Hermite Interpolation**

****

|  |  |  |  |
| --- | --- | --- | --- |
| T | X | Y | z |
| 50 | -1.651338528379989e+01 | 2.715794088433153e+01 | -9.083508020830360e+00 |

I found the indexes for the t vector that had values crossing over 50. Using values, their indexes, and the corresponding x, y, and z values, I was able to create a cubic Hermite Interpolation. These answers are very close to that of the cubic spline interpolation, which verify that both of the interpolants are well constructed.

* 1. **Plot the Splines with Clamped Conditions**

****

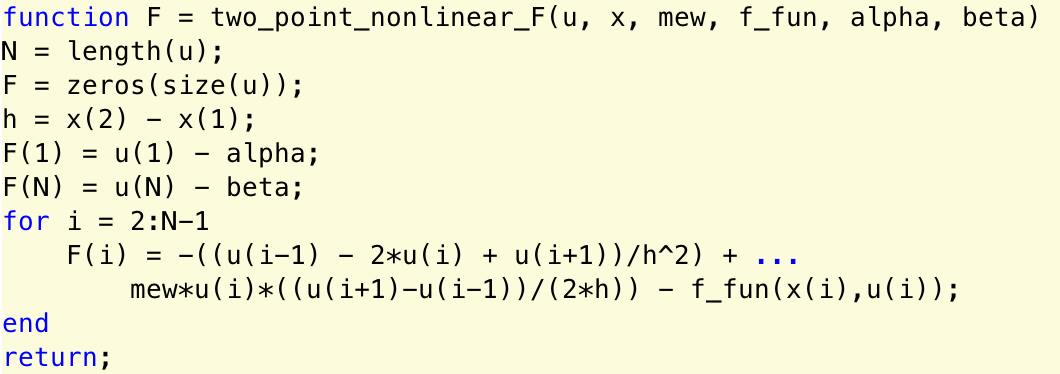
This solution was found by first using the plot to estimate a general location for the intersection between the two interpolants. This enabled me to use the secant method for finding the zero from their subtractions. I used the secant method because it did not require a derivative like Newton’s method, but only easy to approximate estimations for the initial and first iteration.

This zero was used to split the integrations of the y and z functions which were integrated to then find the total area between the curves.

Area = 1.225676158003338e+01

1. BVP for state Burgers
   1. Derive Finite Difference

For this problem, I used the centered finite difference formula to approximate the first derivative of y, which was translated into u for the function as was exampled by in class power point slide. I did this because the approximation for the second derivative was also done using centered difference.

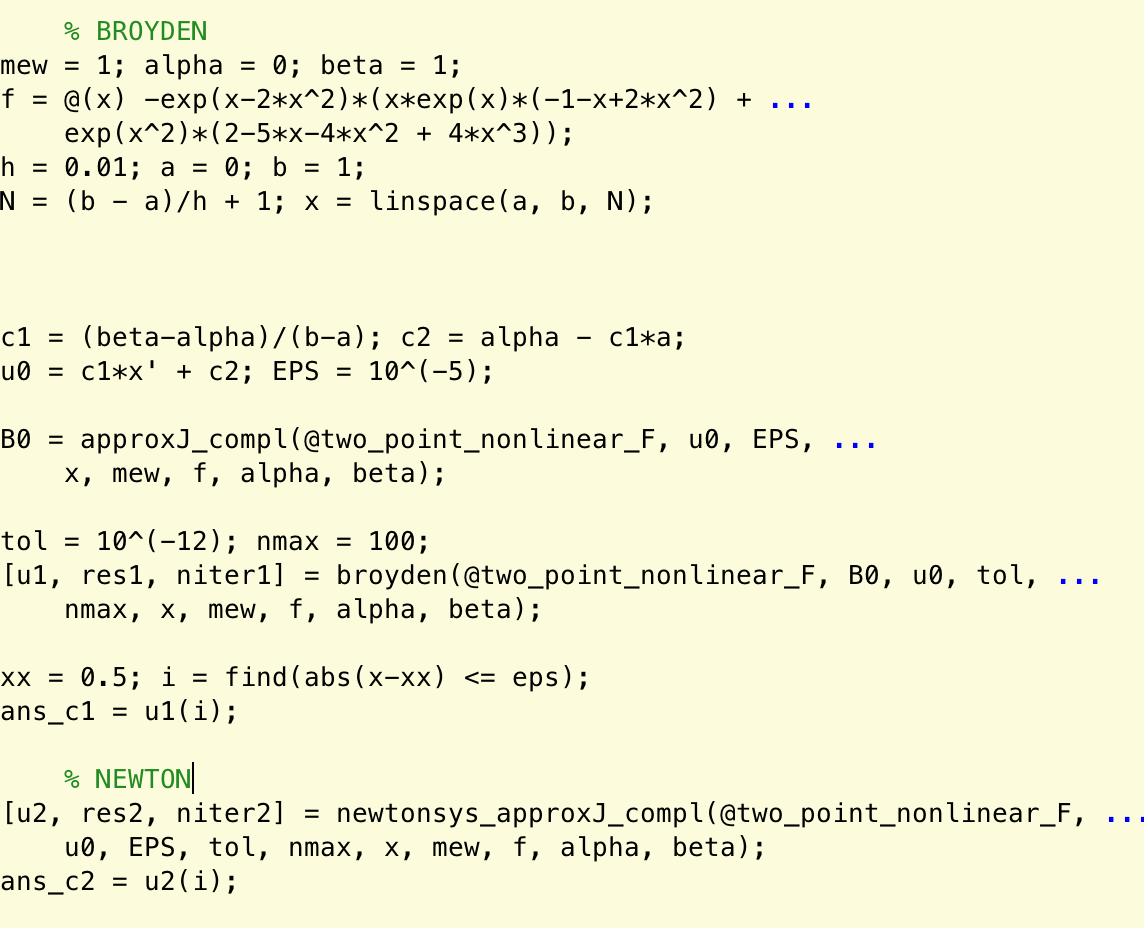


* 1. Jacobian
  2. **Use Broyden and Newton’s method**

I used the Broyden and Newton methods to solve the BVP for the Burger state equations. I used the approxJ function from earlier assignments to approximate the B0 estimate to be used in Broyden’s method. This is good because although we can use any estimation for B0, a matrix close to the Jacobian helps with accuracy and convergence.

For Newton’s method, I used the newton\_sys\_approxJ\_compl function from class and homework because it approximates the Jacobian to be used automatically. Furthermore, this was the same method for approximating the Jacobian as I used for Broyden’s method, so I chose it instead of using the finite-difference method for Newton systems.

Both answers were very close with help the idea that they are accurate.



|  |  |  |
| --- | --- | --- |
| Method | Number of Iter | y(0.5) |
| Broyden | 7 | 6.420249594878591e-01 |
| Newton | 5 | 6.420249594878589e-01 |

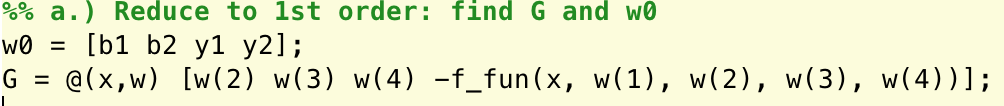
* 1. **Interpolation**

I used Lagrange polynomial interpolation since there was no derivative values available for cubic splines. I used the Newton’s method data since there was not much of a difference from Broyden’s, and chose 4 points- 2 after and 2 before the index where 1/pi would be in the x domain. Considering that the plot of the data generates a fairly straight line, the approximation from the interpolation seems to be a good fit for where the answer would lie.

Answer = 3.954523242095773e-01

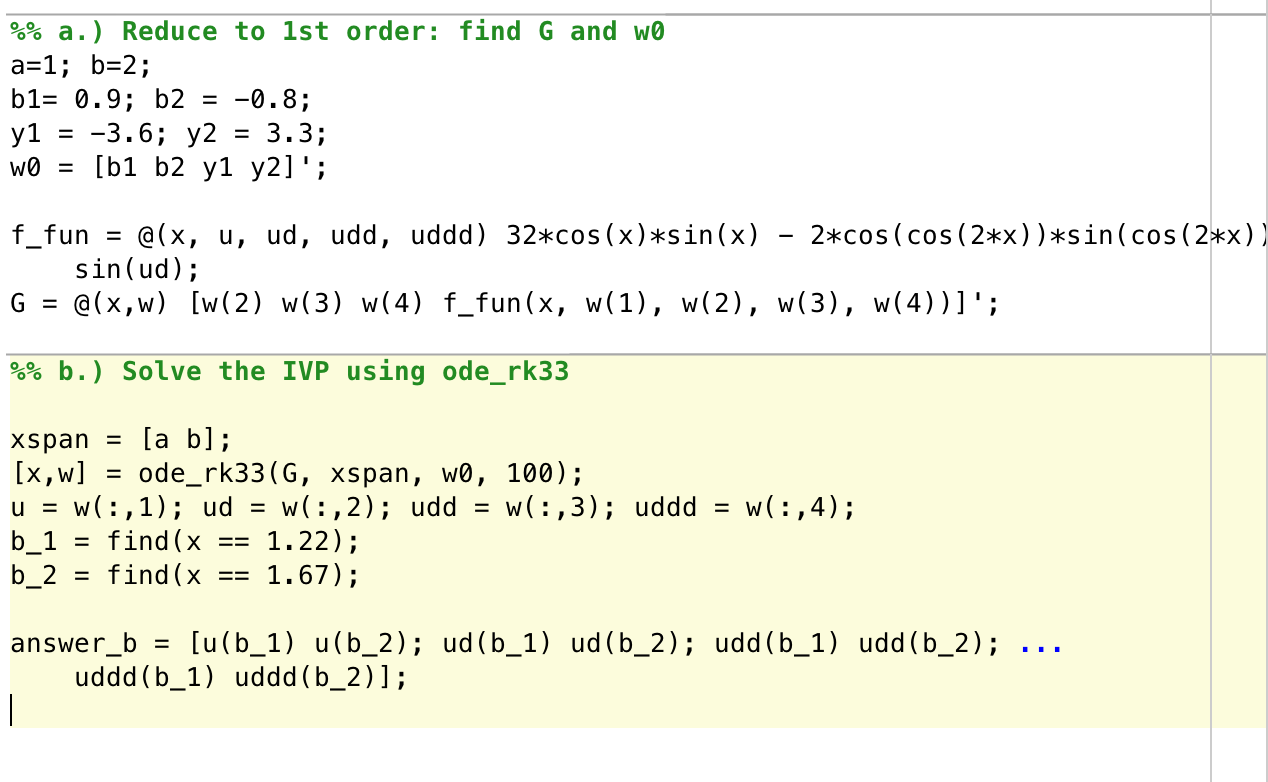
1. **BVP for Shooting Method**
   1. **Reduce the IVP to 4th differential equation**

Following the classroom power point notes, I reduced the order to 1st order ODE. The initial w points were found using the initial values given by the problem statement since each element in w is representative of each derivative for u

****

* 1. **Solve the IVP**

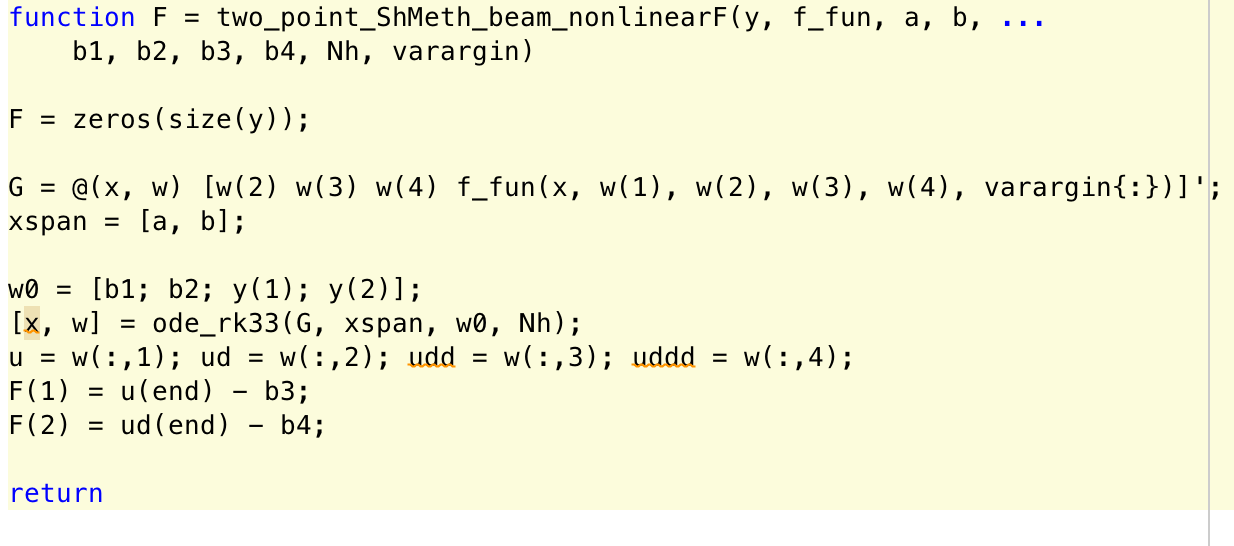
I followed the power point example from class and used your ode\_rk33 with the given parameters to solve the ODE. The indexes were easy to find by just looking at the x vector, or inputting into MATLAB using the find function.

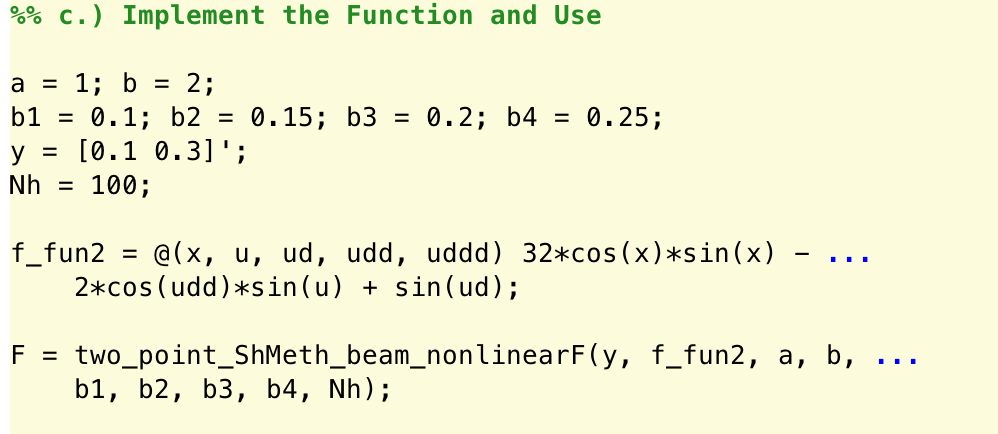
****

|  |  |  |
| --- | --- | --- |
| X | 1.22 | 1.67 |
| U | 6.440921406144874e-01 | -1.777932745371686e-01 |
| Ud | -1.487828928175715e+00 | -1.909705199499413e+00 |
| Udd | -2.550572045042943e+0 | 8.068214904006586e-01 |
| Uddd | 6.084218750423219e+00 | 7.810933527782076e+00 |

* 1. **Implement the function and use**

I implemented the two\_point function following the power point example but altering the parameters to account for the increase in w indexes and derivative orders. Also, I had the function output a vector of 2 elements.





F = [5.822034909416183e-01

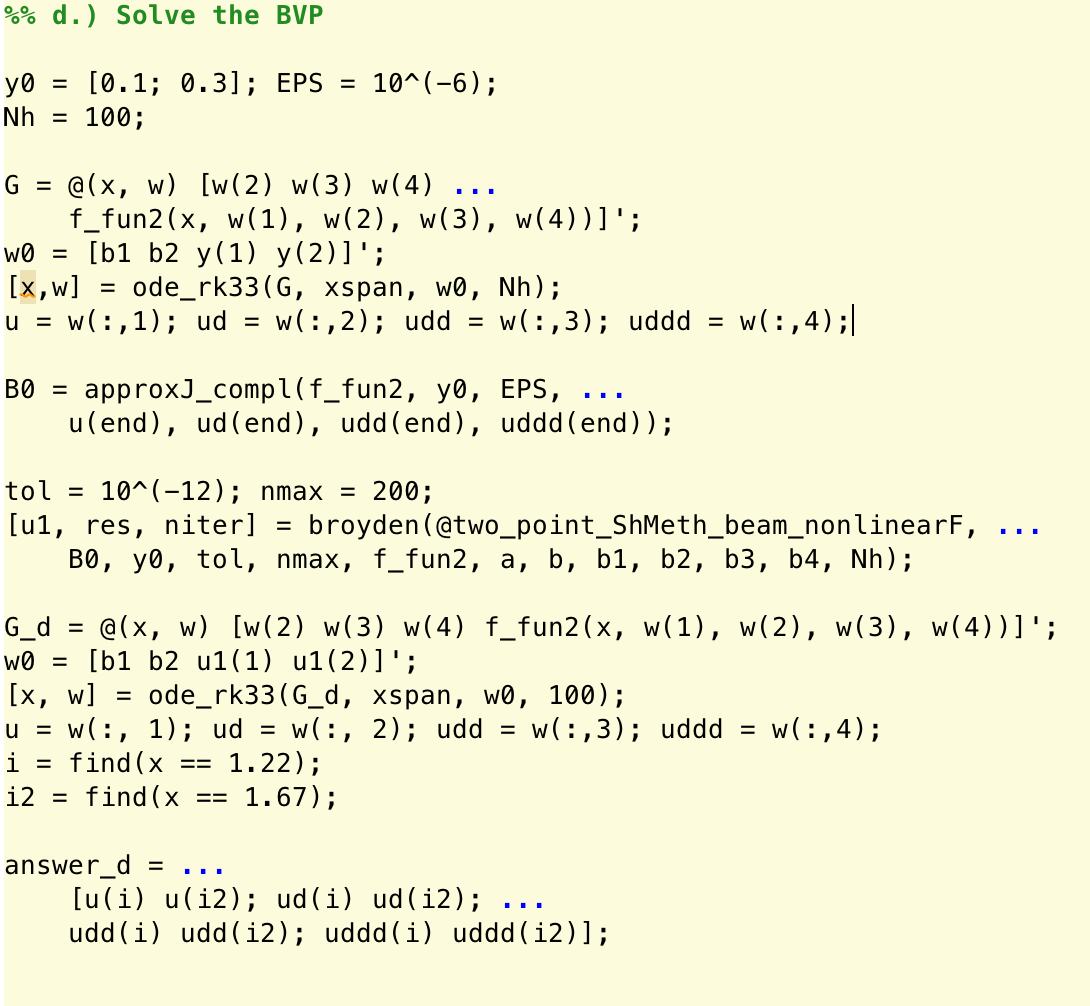
1.671596297303153e+00]

This MATLAB function of mine reproduces F(y) given above. (from the problem)

* 1. **Solve the BVP with Steps given**

I solved this question by first finding the u values through the same process as from previous questions but with the new function. This was used to calculate B0, in which I used the endpoints since that followed the convention of the last problem. This B0 was used for Broyden’s method.

This was used to find the solutions mentioned in the Step 2 of the problem.

****

|  |  |  |
| --- | --- | --- |
| X | 1.22 | 1.67 |
| U | 1.274036472358573e-01 | 1.502027507891033e-01 |
| U’ | 9.049244754113399e-02 | 6.652200360305605e-02 |
| U’’ | -3.391864740953957e-01 | 3.566205631239514e-01 |
| U’’’ | 2.145477798682272e-01 | 1.847413278734568e+00 |