Motion Control in Computer-Aided Modeling

MAD 4401 Intro to Numerical Analysis

Group Name: Green

Group Members:

Davis Jackson, Sophia Teklitz, Matthew Taylor, Anna Yang, Yixuan Yang

Task	Person Completed	Date Completed
Activity 1	Davis, Yixuan, Anna	11/26/2023
Activity 2	Davis, Sophia, Anna, Yixuan	11/27/2023
Activity 3	Davis, Sophia, Anna, Yixuan	11/27/2023
Activity 4	Davis, Sophia, Anna, Yixuan	11/27/2023
Activity 5	Davis, Sophia	12/01/2023
Activity 6	Sophia	12/02/2023
Activity 7	Sophia	12/02/2023
Presentation	Everyone	12/02/2023
Written Report	Everyone	12/06/2023

Basic Scenario:

In the fields of computer-aided modeling for design and manufacturing, one of the most critical challenges is the precise control of motion along a given path. For example, when using a cutting tool to cut a component, or programming robotics hand movement, the ability to complete the trajectory precisely is essential. This project aims to provide a motion control framework, which is applicable across various fields, from industrial machinery manufacturing to animation and robotics.

The need for precision in motion control arises from the high standards of modern production and design. In manufacturing, the performance of machined parts is directly influenced by the ability to maintain the toolpath without making errors (*Ultra-Precision Motion Control for Machining*, n.d.). Similarly, in the animation industry, the realism and fluidity of movement depend on the ability to control the motion of characters and objects seamlessly (Limtrakul et al., 2010). Both scenarios demand a method of breaking down complex paths into segments that allow for uniform motion, a challenging requirement due to the complexity of path geometries and the laws of physics that relate to movement.

The main idea of this project is based on the principle of equipartitioning a path into equal arc lengths, which guarantees that a moving point will travel through each segment in an equivalent period. This requires techniques that are able to handle the non-linear nature of pathways. Traditional methods like linear interpolation are insufficient as they do not account for curvature and speed variation along the path; therefore, leading to uneven speeds and inconsistent results.

To overcome these limitations, the project employs Adaptive Quadrature. This numerical integration technique adjusts the step size for certain subintervals during the calculation to achieve an arc length that is more precise. This method is particularly effective when dealing with non-linear parametric curves, which are commonly used to define paths in computer-aided modeling. Moreover, root finding methods, Bisection Method and Newton's Method, are incorporated to locate equipartitioning points more precisely, hence resulting in a better approximation of integration.

Algorithm:

The rationale behind the algorithm for this project is to equally divide a motion path into segments with equal arc lengths, in addition to combining numerical integration and root-finding techniques. This process is very important and useful for applications requiring uniform motion where precision dictates the quality of the output.

The primary numerical approach applied is adaptive quadrature, a kind of numerical integration that adapts the size and number of intervals depending on the behavior of the function along the range. Unlike fixed-step methods, Adaptive Quadrature responds to variations among the function, providing additional computational capability to regions with greater curvature or fluctuation. Adaptive Quadrature is especially helpful when dealing with non-linear curvatures that have non-uniform speed.

First of all, Adaptive Quadrature begins by calculating the integral over the entire path, which is arc length in this case. The next step is to break this whole arc into a few segments, and then determine the arc length of each segment by taking integral over those segments. This process is repeated until the sum of the estimated segment length is within the error tolerance level. If the sum deviates noticeably from the actual estimate or has too large error, then the segment with the largest error is further subdivided.

When a parametric curve is expressed as an integral, its arc length usually cannot be solved in a closed form. Therefore, numerical integration is required. However, simply knowing the arc lengths is not enough for motion control. To keep motion constant, one must also be able to locate specific points along the path that correspond to equal increments of this arc length. This requires a robust root-finding algorithm.

Two root-finding methods were implemented in this project: the Bisection Method and Newton's Method. The Bisection Method is a straightforward and iterative approach that shrinks the range in which a root—in this case, the parameter value corresponding to the given arc length—lies. It is chosen for its convergence, provided that the function changes the sign over the interval. However, the Bisection Method is not very efficient since its rate is linearly convergent, which makes the solution not as precise.

On the other hand, Newton's Method presents an improved approach. It finds the location of each root by calculating the derivative of the function. Since it is quadratically convergent (unless g(r) = 0 in which case it's linearly convergent), the accuracy improves during each step since the number of correct digits roughly doubles in each step in Newton's Method whereas the number of correct digits increases by one in the Bisection Method. Moreover, Newton's Method typically reaches a solution with fewer iterations compared to the Bisection Method, resulting in a faster computation. Nevertheless, Newton's Method converges locally, which means if starting with a good initial guess and the existence of a derivative, the solution would be highly accurate.

Using the Bisection Method and Newton's Method, points along the curve for equipartitioning are accurately located, and the arc length of each segment is ensured to be the same, with the speed moving along the path for each segment being the same, resulting in the speed constant.

Program Structure:

For Question 1, the inputs for the adapquad function, which is in the supporting file adapquad.m, are created with are the variables f, a, b, and errtol. The arclength of f with the given x(t) and y(t) functions is the output, within the bounds of a and b and using the acceptable error of the adaptive quadrature given. The variables a and b must be between 0 and 1 as this is a constraint of the parametric equations used to create the arc.

For Question 2, the input variables are s, g, the bounds and the errtol. The variable s is the desired proportion of the arc length. The variable g is the formula used to solve for the $t^*(s)$ value to find which point on the arc corresponds to the desired s value. This is done by having the ratio of the indefinite integral of the arc length equation over the arclength and then subtracting the value s. The arclength variable in this function is the arc length found in the previous question.

The output of this function is ts, the value of $t^*(s)$ found using the bisection method. The bisection method comes from the supporting file bisect.m.

For Question 3, the main portion of interest is the for loop. In the case above, the arc is being partitioned into 4 equal parts using the arguments from Question 2, with the added matrix A to hold each value of $t^*(s)$. Afterward, the end points are added since the bisection method failed at those points. The final lines of code are evaluating the $t^*(s)$ points at x(t) and y(t). These will be used in graphing the plots.

This is how the two arcs are plotted. The plot function is used, with the x4 and y4 (or x20 and y20) as the x and y values respectively. The 'ro-' argument affects how the plot will appear. The xlim and ylim functions affect the margins of the graphs.

For Question 4, the variable gprime was added. This is the derivative of the function used to find $t^*(s)$. This was found using WolframAlpha for ease of calculations. Then the newton function was used as an alternative to the bisection method, from the supporting file newton.m. The arguments are the function, its derivative, an estimate for the point, and the number of steps.

The for loop is a modification of the one used for the bisection method in Question 3. To graph the function, the exact same code can be used from Question 3.

For Question 5, the path of the curves with 4 and 20 partitions is animated. The inputs include modifiers for the animation and the for loop to plot the points. Each point found before is added sequentially and drawn using the drawnow function. The pause function slows the animation down by adding a pause between each point added.

For Question 6, the only new inputs are the new parametric functions and the new arc length function. The remaining process to animate the function does not change from the process outlined above.

For Question 7, the only new addition was the modification to the s values in the for loop to find $t^*(s)$ to account for the new C(s) function. Here, s was squared. The variables can be modified to reflect other C(s) functions.

Discussion:

- **Why is your algorithm good/bad? What could be done to make it better?**

The algorithm used only includes the necessary code to answer each question, but it could have been organized better. Each part feels disjoint from each other despite the code building off of what was done in the previous question. The code could have been condensed to make the process of running it more streamlined and easier to understand. Some of the for loops could have been made into functions to prevent having to duplicate the code with minor changes.

Overall, the code runs quickly and well, which is a benefit. There are no times when the code slows down, other than running the animation which takes a couple of seconds, or runs into an error.

The main way that it can be improved is to implement Newton's Method more often than when it was required for Question 4. Bisection Method was used more often since it did not have the condition of choosing a point near where the $t^*(s)$ value is, choosing accuracy over speed. If run time was an issue, the code should be adapted to use Newton's Method to help improve run time. Another way to improve the runtime would be to use Simpson's Rule in Adaptive quadrature as recommended by Question 3. This could be recommended for arc lengths with larger bracketing intervals but in our implementation it is only called a few times with small bracketing intervals.

- **Use cases in real-world problems**

When integrating adaptive quadrature and root-finding methods to achieve motion control, it will open up various potential uses in industries. For instance, this method can be applied to advanced manufacturing in guiding numerical control machines to create complex parts with high precision. In making subparts of a medical device or aerospace machinery, it is important that the cutting tool follows a designated path with constant speed so that the critical parts are precise (*Ultra-Precision Motion Control for Machining*, n.d.). In the field of robotics, this algorithm can be employed to program robotic arms to make movements. For example, in robotic-involved assembly lines, the smoothness of motion translates directly to the efficiency and quality of the assembly process (Liu, Liu, 2021). In computer graphics and animation, the same principles ensure the movement of characters looks more realistic by making sure the animators follow the paths with a uniform velocity (Limtrakul et al., 2010). In addition, such an algorithm can be applied to autonomous vehicles. Figuring out the path required to move in a curved shape at a constant speed can be used to improve comfort and ensure the safety of passengers (Syavasya, Muddana, 2022). In general, all the above applications benefit from the ability to break down complex pathways into simpler and uniform segments and control the motion.

- **What did not work? Where were the problems?**

The part with the most difficulty was trying to get Newton's and Bisection Methods to work. For the Bisection Method, the way the function was written needed to be modified. After ensuring that all the carets had periods before them, the Bisection Method worked without any errors.

For Newton's Method, it was harder to get it to work. The part of the code that was most difficult was finding a way to compute the derivative of the function used to find the $t^*(s)$ value. The original attempt used the diff function, but there were complications when the derivative was plugged into the Newton function. Even after trying to modify the way the derivative was written, running the Newton function with the derivative written like that continued giving error codes. To make things easier, the derivative was taken using WolframAlpha. When it was written out in this way, the Newton function worked and it was able to be used to find the $t^*(s)$ value.

- **Speed vs. accuracy**

In this algorithm, one of the most important considerations is to find a balance between speed and accuracy of the motion. Adaptive Quadrature was chosen for its dynamic precision. In some usage scenarios and applications, small errors may lead to major quality issues or functional deviations. Therefore, although adaptive quadrature may be slower than fixed-point methods, it provides the high accuracy that applications require. Moreover, Adaptive Quadrature has the ability to adjust according to the complexity and variance of the path. It avoids unnecessary calculations in order to optimize speed and accuracy.

The integration of both the Bisection Method and Newton's Method for root-finding was another wise choice in this algorithm. Although slower, the Bisection method guarantees convergence, meaning that it ensures the root-finding is successful. Newton's Method, on the other hand, offers faster convergence but has the risk of being non-convergent when the initial guess is poorly chosen or if the function's behavior is turbulent. That being the case, the inclusion of both methods allowed us to maintain accuracy through the Bisection Method while Newton's Method improves the speed when applicable.

Predominantly, it is important to make sure that the accuracy of the path is not compromised for speed since the applications demand a high level of precision for practical use.

Reference

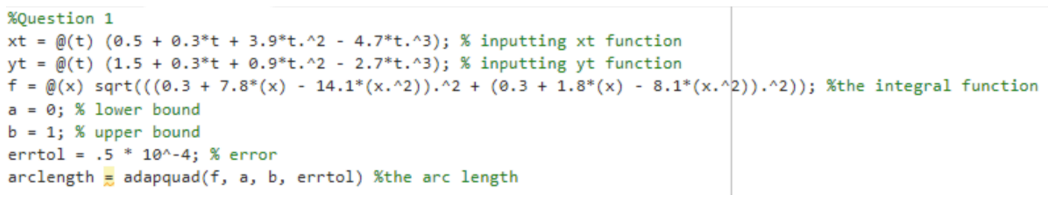
Limtrakul, S., Hantanong, W., Kanongchaiyos, P., and Nishita, T. (2010), *Reviews on Physically Based Controllable Fluid Animation*, Engineering Journal [online]. Available from: https://engj.org/index.php/ej/article/view/99#:~:text=In%20computer%20graphics%20animation%2C%20animation,support%20more%20required%20control%20model [Accessed 5th Dec 2023]

Liu, S., Liu, P. (2021), *A Review of Motion Planning Algorithms for Robotic Arm Systems*, Springer Link [online]. Available from: https://link.springer.com/chapter/10.1007/978-981-16-4803-8_7#:~:text=,as%20welding%2C%20installation%2C%20and%20inspection [Accessed 5th Dec 2023]

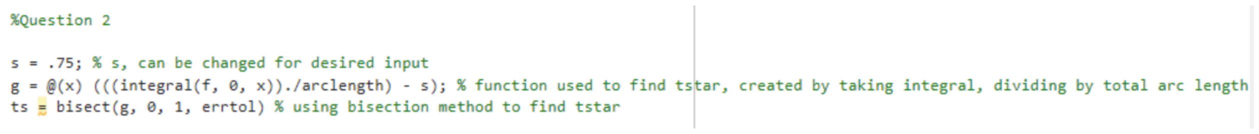
Syavasya, C., Muddana, A. (2022), *Optimization of autonomous vehicle speed control mechanisms using hybrid DDPG-SHAP-DRL-stochastic algorithm* [online]. Available from: https://www.sciencedirect.com/science/article/pii/S096599782200148X#:~:text=The%20schematic%20diagram%20for%20the,speed%20control%20of%20autonomous%20vehicle [Accessed 5th Dec 2023]

Ultra-Precision Motion Control for Machining (n.d.), Polaris Motion [online]. Available from: https://pmdi.com/applications/advanced-motion-control-specialty-applications/ultra-precision-motion-control/#:~:text=Key%20Polaris%20Motion%20technologies%20used,Limited%20jerk%20trajectory%20generation [Accessed 5th Dec 2023]

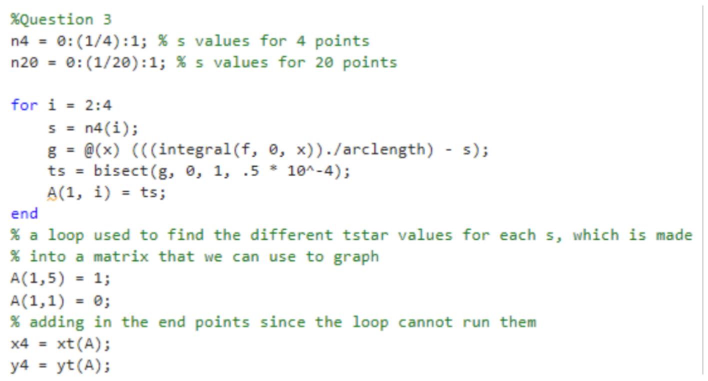
Code for Questions 1 - 3:



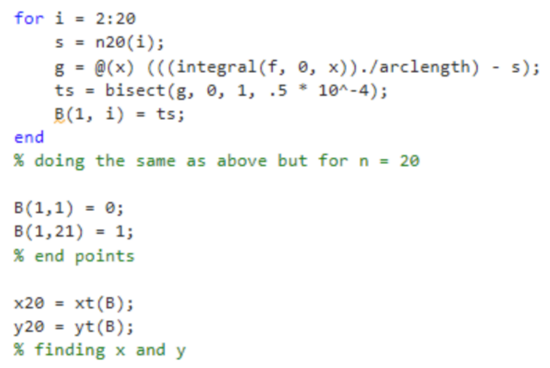
```
%Question 1
xt = @(t) (0.5 + 0.3*t + 3.9*t.^2 - 4.7*t.^3); % inputting xt function
yt = @(t) (1.5 + 0.3*t + 0.9*t.^2 - 2.7*t.^3); % inputting yt function
f = @(x) sqrt(((0.3 + 7.8*(x) - 14.1*(x.^2)).^2 + (0.3 + 1.8*(x) - 8.1*(x.^2)).^2)); %the integral function
a = 0; % lower bound
b = 1; % upper bound
errtol = .5 * 10^-4; % error
arclength = adapquad(f, a, b, errtol) %the arc length
```



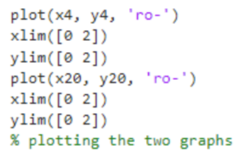
```
%Question 2

s = .75; % s, can be changed for desired input
g = @(x) (((integral(f, 0, x))./arclength) - s); % function used to find tstar, created by taking integral, dividing by total arc length
ts = bisect(g, 0, 1, errtol) % using bisection method to find tstar
```



```
%Question 3
n4 = 0:(1/4):1; % s values for 4 points
n20 = 0:(1/20):1; % s values for 20 points

for i = 2:4
    s = n4(i);
    g = @(x) (((integral(f, 0, x))./arclength) - s);
    ts = bisect(g, 0, 1, .5 * 10^-4);
    A(1, i) = ts;
end
% a loop used to find the different tstar values for each s, which is made
% into a matrix that we can use to graph
A(1,5) = 1;
A(1,1) = 0;
% adding in the end points since the loop cannot run them
x4 = xt(A);
y4 = yt(A);
```

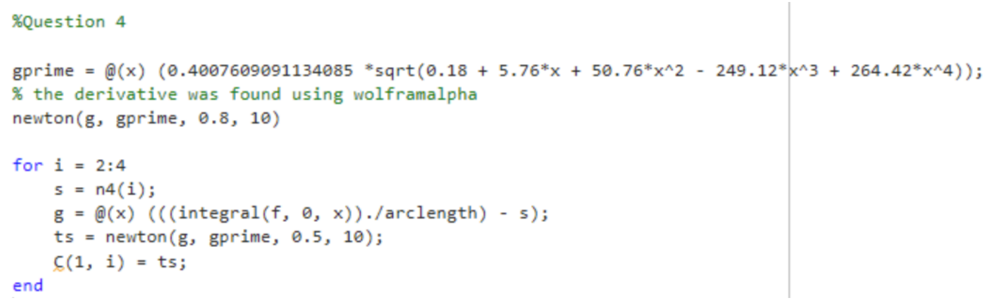


```
for i = 2:20
    s = n20(i);
    g = @(x) (((integral(f, 0, x))./arclength) - s);
    ts = bisect(g, 0, 1, .5 * 10^-4);
    B(1, i) = ts;
end
% doing the same as above but for n = 20

B(1,1) = 0;
B(1,21) = 1;
% end points

x20 = xt(B);
y20 = yt(B);
% finding x and y
```

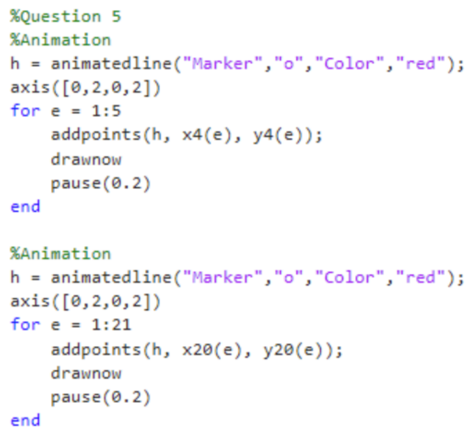


```
plot(x4, y4, 'ro-')
xlim([0 2])
ylim([0 2])
plot(x20, y20, 'ro-')
xlim([0 2])
ylim([0 2])
% plotting the two graphs
```

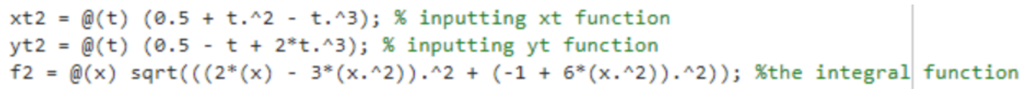
Code for Questions 4 - 5:



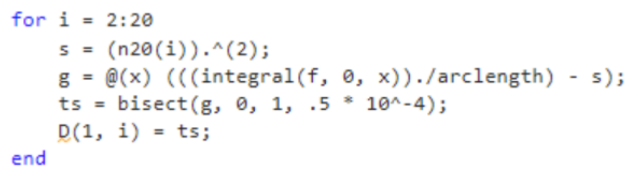
```
%Question 4

gprime = @(x) (0.4007609091134085 *sqrt(0.18 + 5.76*x + 50.76*x^2 - 249.12*x^3 + 264.42*x^4));
% the derivative was found using wolframalpha
newton(g, gprime, 0.8, 10)

for i = 2:4
    s = n4(i);
    g = @(x) (((integral(f, 0, x))./arclength) - s);
    ts = newton(g, gprime, 0.5, 10);
    C(1, i) = ts;
end
```



```
%Question 5
%Animation
h = animatedline("Marker","o","Color","red");
axis([0,2,0,2])
for e = 1:5
    addpoints(h, x4(e), y4(e));
    drawnow
    pause(0.2)
end

%Animation
h = animatedline("Marker","o","Color","red");
axis([0,2,0,2])
for e = 1:21
    addpoints(h, x20(e), y20(e));
    drawnow
    pause(0.2)
end
```

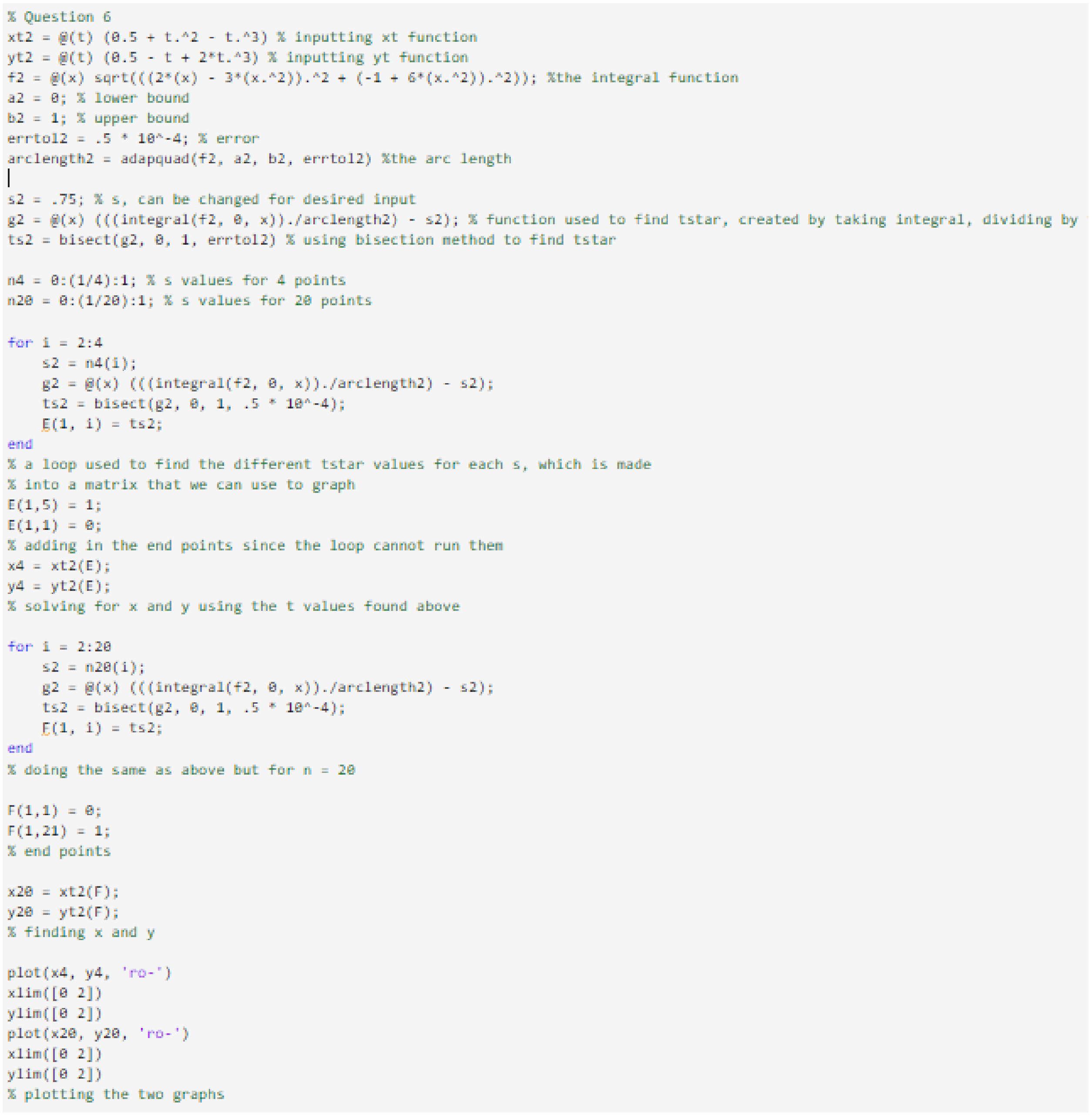


```
xt2 = @(t) (0.5 + t.^2 - t.^3); % inputting xt function
yt2 = @(t) (0.5 - t + 2*t.^3); % inputting yt function
f2 = @(x) sqrt(((2*(x) - 3*(x.^2)).^2 + (-1 + 6*(x.^2)).^2)); %the integral function
```



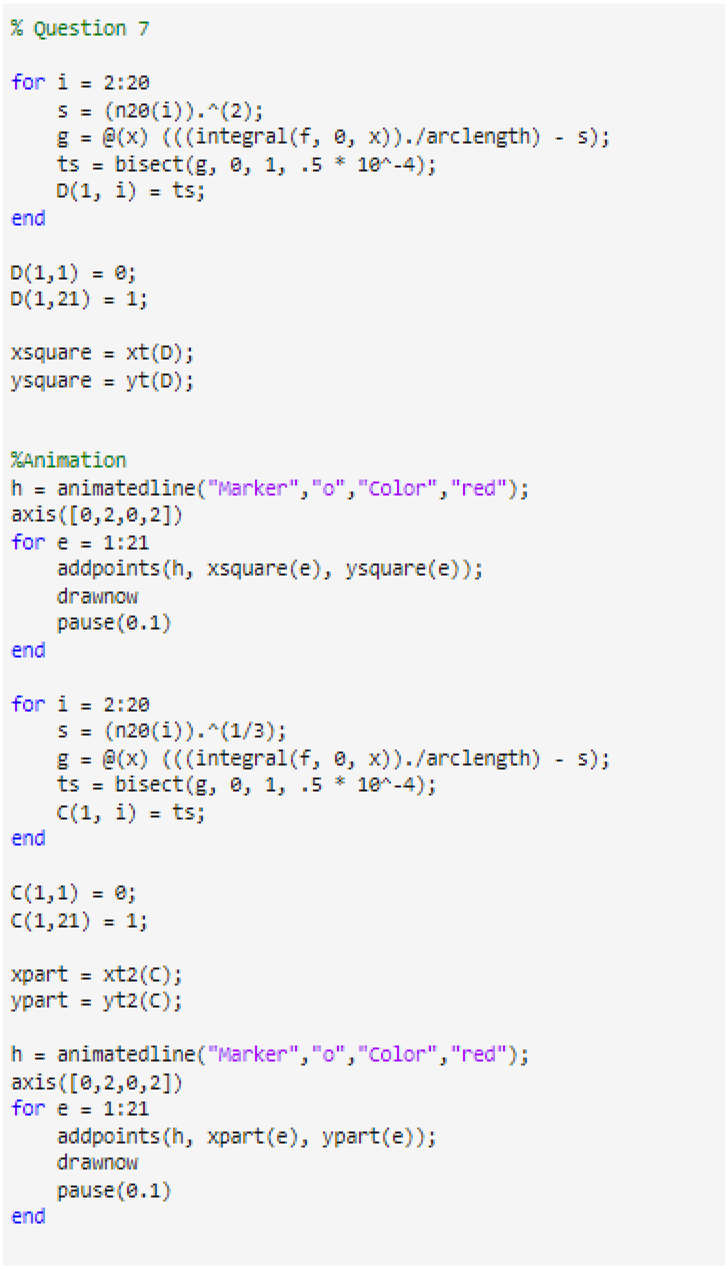
```
for i = 2:20
    s = (n20(i)).^(2);
    g = @(x) (((integral(f, 0, x))./arclength) - s);
    ts = bisect(g, 0, 1, .5 * 10^-4);
    D(1, i) = ts;
end
```

Code for Question 6:



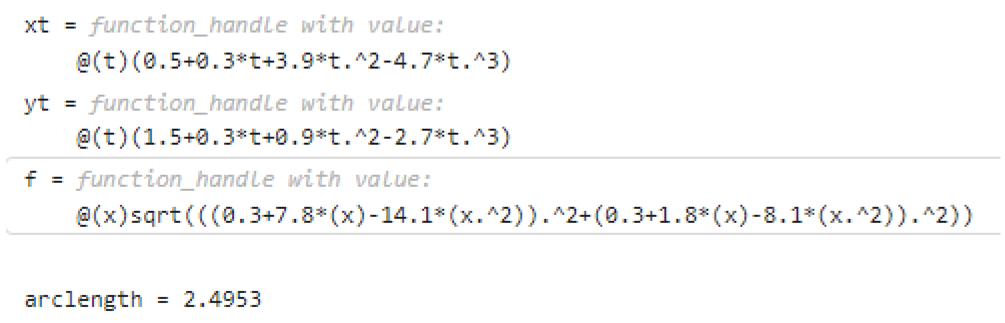
```
% Question 6
xt2 = @(t) (0.5 + t.^2 - t.^3) % inputting xt function
yt2 = @(t) (0.5 - t + 2*t.^3) % inputting yt function
f2 = @(x) sqrt(((2*(x) - 3*(x.^2)).^2 + (-1 + 6*(x.^2)).^2)); %the integral function
a2 = 0; % lower bound
b2 = 1; % upper bound
errtol2 = .5 * 10^-4; % error
arclength2 = adapquad(f2, a2, b2, errtol2) %the arc length

s2 = .75; % s, can be changed for desired input
g2 = @(x) (((integral(f2, 0, x))./arclength2) - s2); % function used to find tstar, created by taking integral, dividing by
ts2 = bisect(g2, 0, 1, errtol2) % using bisection method to find tstar

n4 = 0:(1/4):1; % s values for 4 points
n20 = 0:(1/20):1; % s values for 20 points

for i = 2:4
    s2 = n4(i);
    g2 = @(x) (((integral(f2, 0, x))./arclength2) - s2);
    ts2 = bisect(g2, 0, 1, .5 * 10^-4);
    E(1, i) = ts2;
end
% a loop used to find the different tstar values for each s, which is made
% into a matrix that we can use to graph
E(1,5) = 1;
E(1,1) = 0;
% adding in the end points since the loop cannot run them
x4 = xt2(E);
y4 = yt2(E);
% solving for x and y using the t values found above

for i = 2:20
    s2 = n20(i);
    g2 = @(x) (((integral(f2, 0, x))./arclength2) - s2);
    ts2 = bisect(g2, 0, 1, .5 * 10^-4);
    F(1, i) = ts2;
end
% doing the same as above but for n = 20

F(1,1) = 0;
F(1,21) = 1;
% end points

x20 = xt2(F);
y20 = yt2(F);
% finding x and y

plot(x4, y4, 'ro-')
xlim([0 2])
ylim([0 2])
plot(x20, y20, 'ro-')
xlim([0 2])
ylim([0 2])
% plotting the two graphs
```

Code for Question 7:



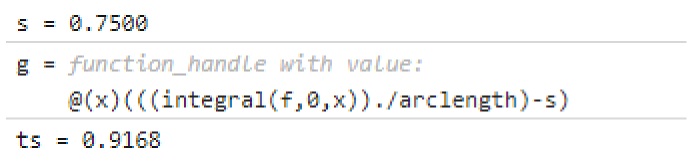
```
% Question 7

for i = 2:20
    s = (n20(i)).^(2);
    g = @(x) (((integral(f, 0, x))./arclength) - s);
    ts = bisect(g, 0, 1, .5 * 10^-4);
    D(1, i) = ts;
end

D(1,1) = 0;
D(1,21) = 1;

xsquare = xt(D);
ysquare = yt(D);


%Animation
h = animatedline("Marker","o","Color","red");
axis([0,2,0,2])
for e = 1:21
    addpoints(h, xsquare(e), ysquare(e));
    drawnow
    pause(0.1)
end

for i = 2:20
    s = (n20(i)).^(1/3);
    g = @(x) (((integral(f, 0, x))./arclength) - s);
    ts = bisect(g, 0, 1, .5 * 10^-4);
    C(1, i) = ts;
end

C(1,1) = 0;
C(1,21) = 1;

xpart = xt2(C);
ypart = yt2(C);

h = animatedline("Marker","o","Color","red");
axis([0,2,0,2])
for e = 1:21
    addpoints(h, xpart(e), ypart(e));
    drawnow
    pause(0.1)
end
```

Results for Question 1:



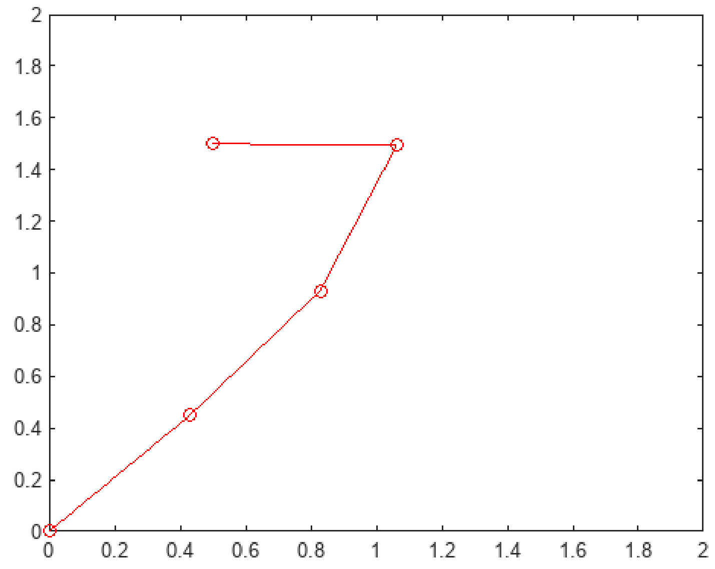
```
xt = function_handle with value:
    @(t)(0.5+0.3*t+3.9*t.^2-4.7*t.^3)
yt = function_handle with value:
    @(t)(1.5+0.3*t+0.9*t.^2-2.7*t.^3)
f = function_handle with value:
    @(x)sqrt(((0.3+7.8*(x)-14.1*(x.^2)).^2+(0.3+1.8*(x)-8.1*(x.^2)).^2))

arclength = 2.4953
```

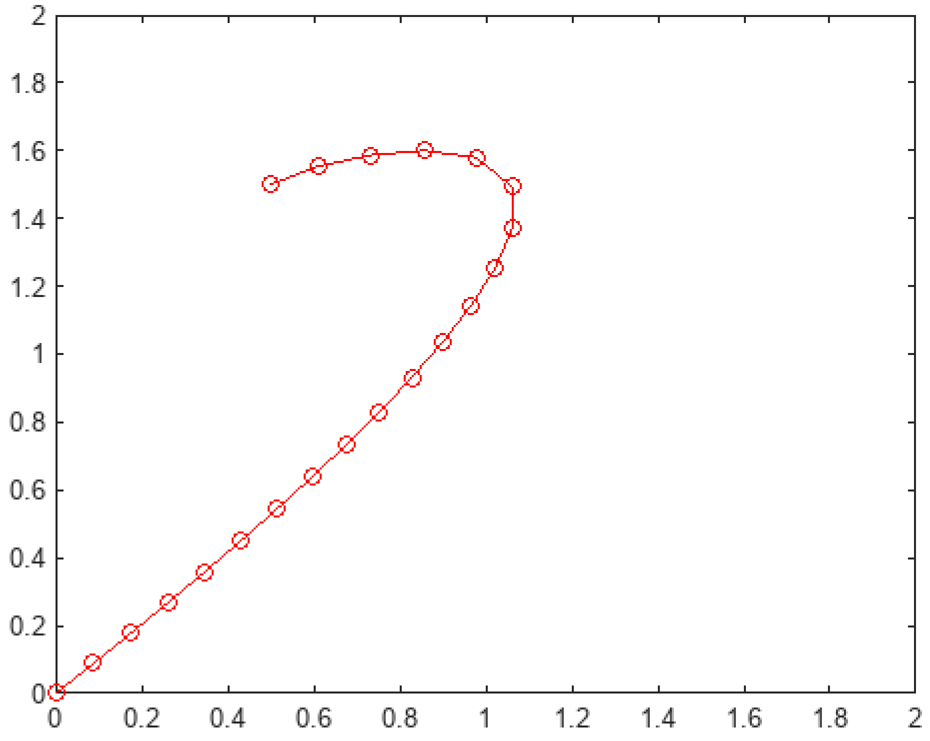
Results for Question 2:



```
s = 0.7500
g = function_handle with value:
    @(x)(((integral(f,0,x))./arclength)-s)
ts = 0.9168
```

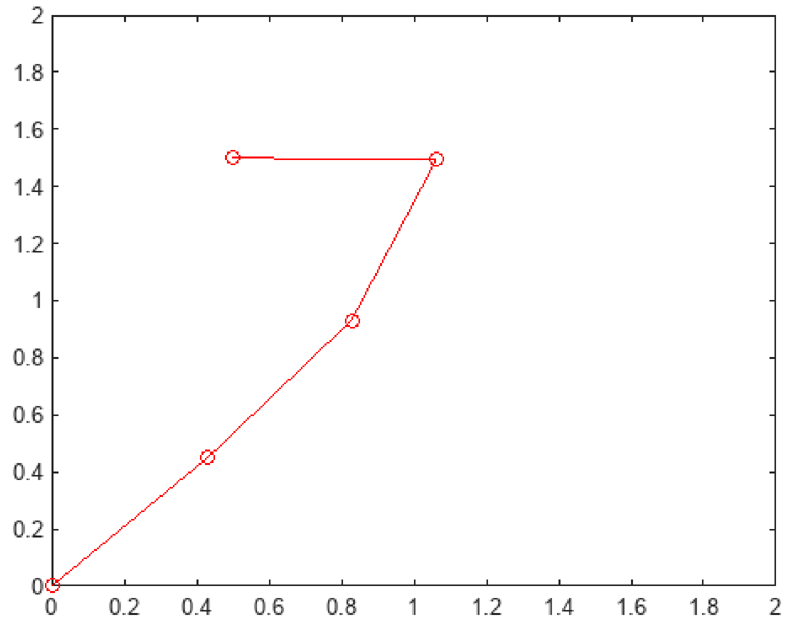
Results for Question 3:



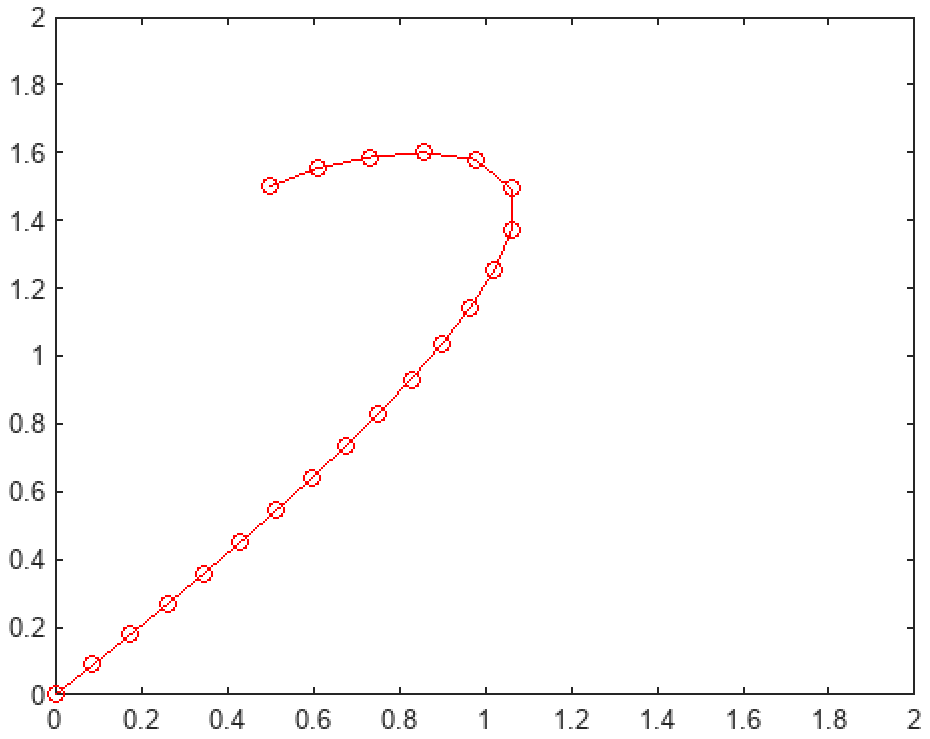


Results for Question 4:

```
ans = 0.9850
```

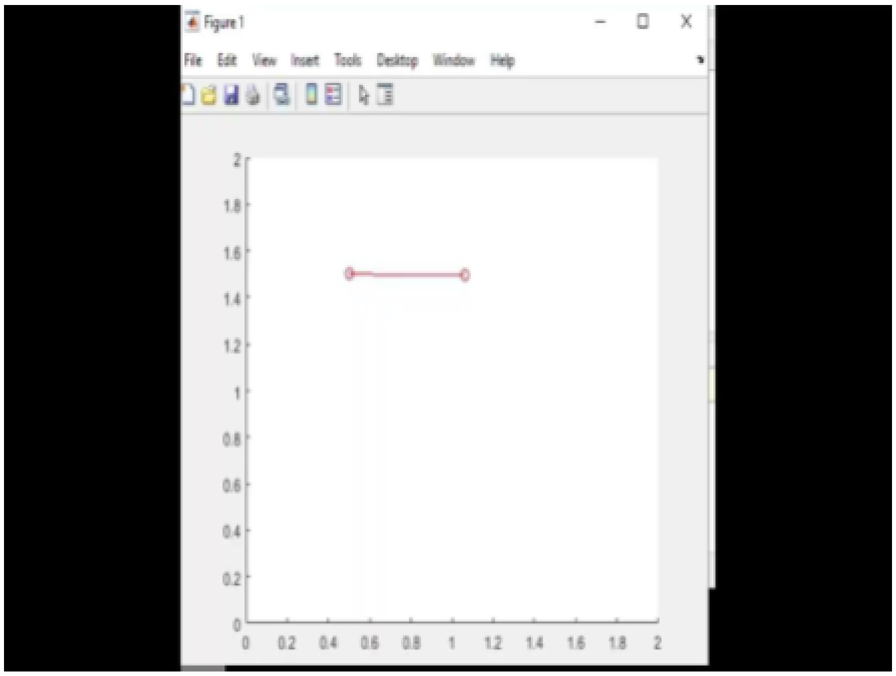


2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2

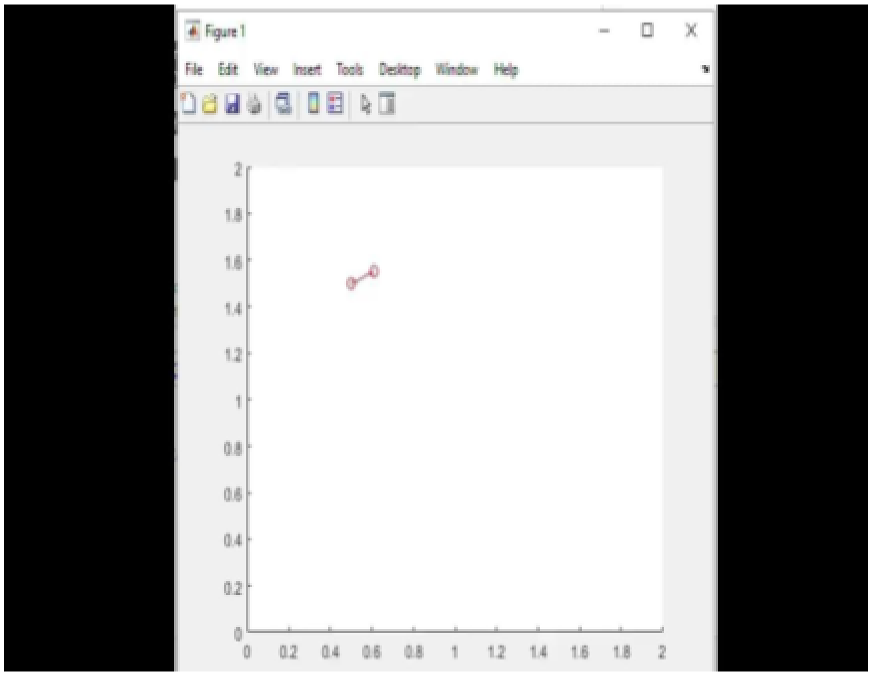


2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2

Bisection Method with n = 4.

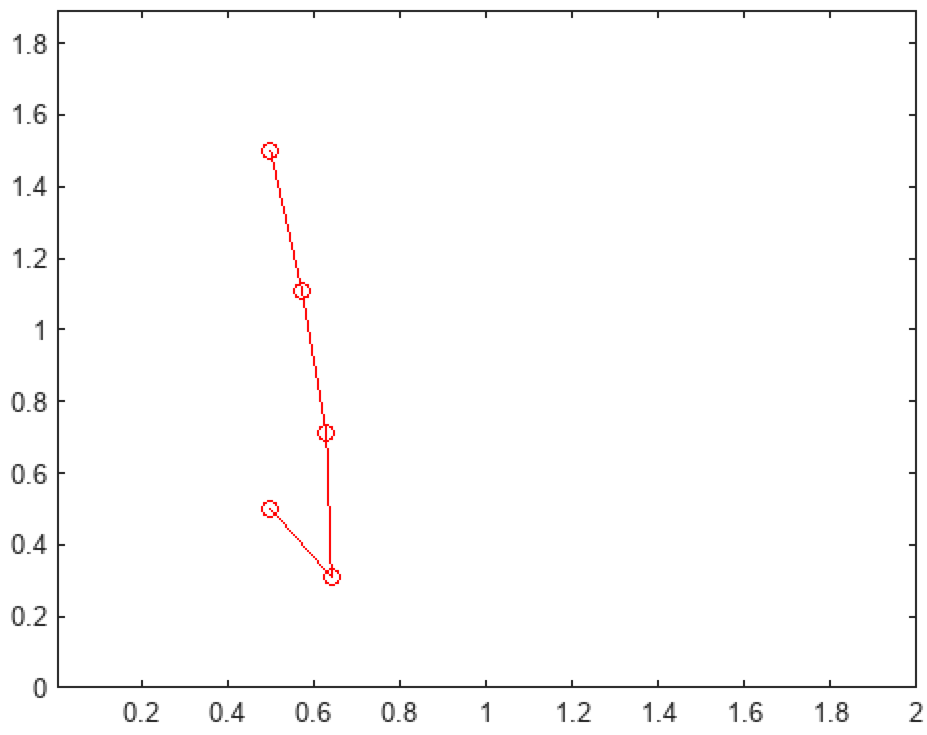


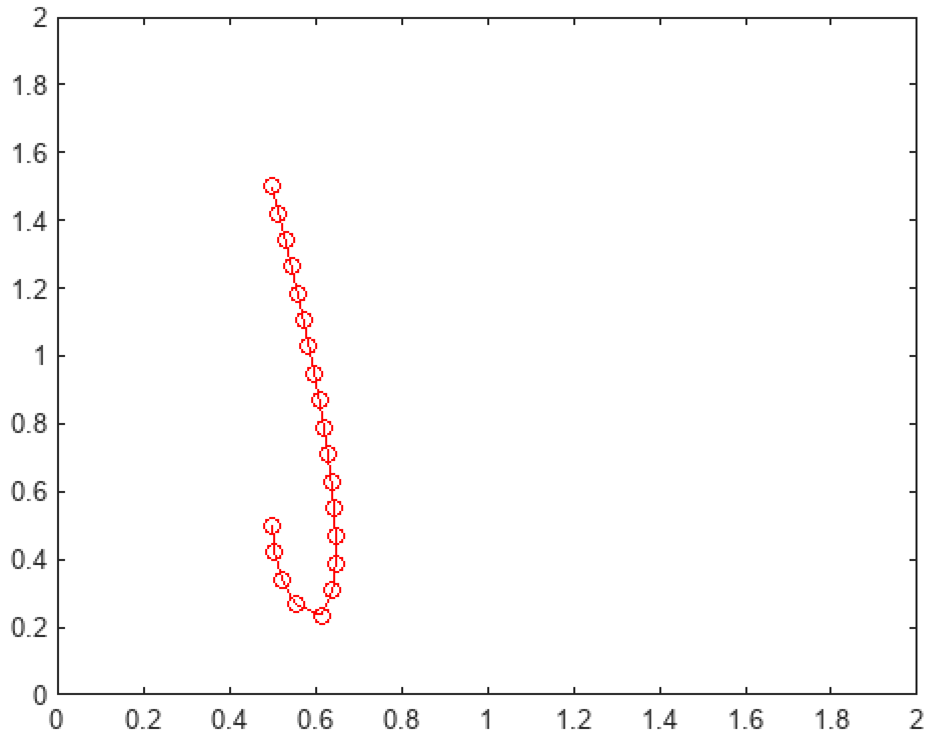
Bisection Method with n = 20.



Results for Question 6:

```
arclength2 = 1.6035
ts2 = 0.9122
```





Original curve with n = 20 and $C(s) = s^2$.

New curve with $C(s) = s^{1/3}$

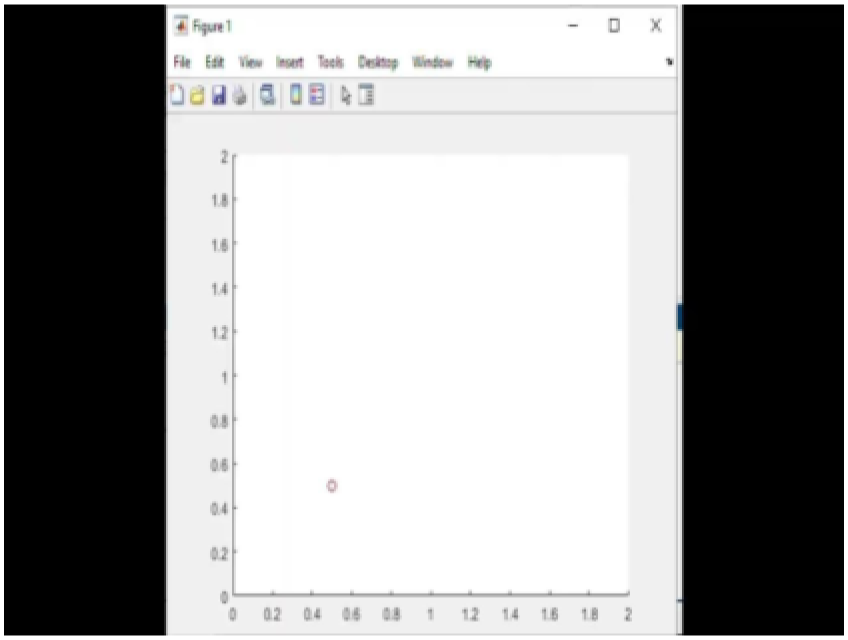


Figure 1
File Edit View Insert Tools Desktop Window Help
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2