Jared Rivera

804603106

CEE/MAE M20

February 14, 2017

Homework 5

1 The Split-and-Average Problem

1.1 Introduction

The goal of this problem is to take a set of points and find the shape they converge to by splitting the points to their midpoint and finding the average to reassign the points to. This will be done using two functions. The first function will split the arrays that hold the x and y coordinates and find their midpoints, assuming the function wraps around on itself like a circle rather than a vector. Next, the second function will find the weighted average of the split points based on a weight vector and reassign the x and y arrays to these average values. This process will be iterated until the reassignment made by the functions is negligible, and the final shape will be plotted in a separate figure from the original.

1.2 Model and Methods

The first step for this problem is to actually write the function that performs the splitting of the x and y arrays. The first step in the function is to calculate the length of the function being inputted and use that as the upper bound of iteration for assigning the original array values to every other space in the new split array. Next, the midpoints are found between each of the original points, and a special case is used to define the midpoint past the final original point by assuming the array is a circle. Both of these methods are shown below

```
for k=2:2:N-1
xs(k)=(xs(k+1)+xs(k-1))/2;
end

xs(N)=(xs(1)+xs(N-1))/2;
```

Now that this function is written and ready to be called, the averaging function has to be written. This is done simply by following the algorithm outlined in the problem statement, with the addition of a special case for each of the endpoints, again under the assumption that the array is a circle. These are

```
xa(1)=w1*(xs(M))+w2*(xs(1))+w3*(xs(2));
xa(M)=w1*(xs(M-1))+w2*(xs(M))+w3*(xs(1));
```

Where the left neighbor of the first entry is the last endpoint, and vice versa. With the two main functions written, the actual script is ready. The x and y coordinate arrays are initialized as are zero arrays that will hold the split and averaged arrays. The last preset value is the node displacement, which is set to 100 to ensure that the loop is entered later on.

Before the loop is entered, the original plot is made. The loop is set to run so long as the node displacement is greater than 0.001, hence the initialization at 100. Inside the loop the split function is called for each x and y array, and then the new split array is passed into the averaging array, as shown on the following page.

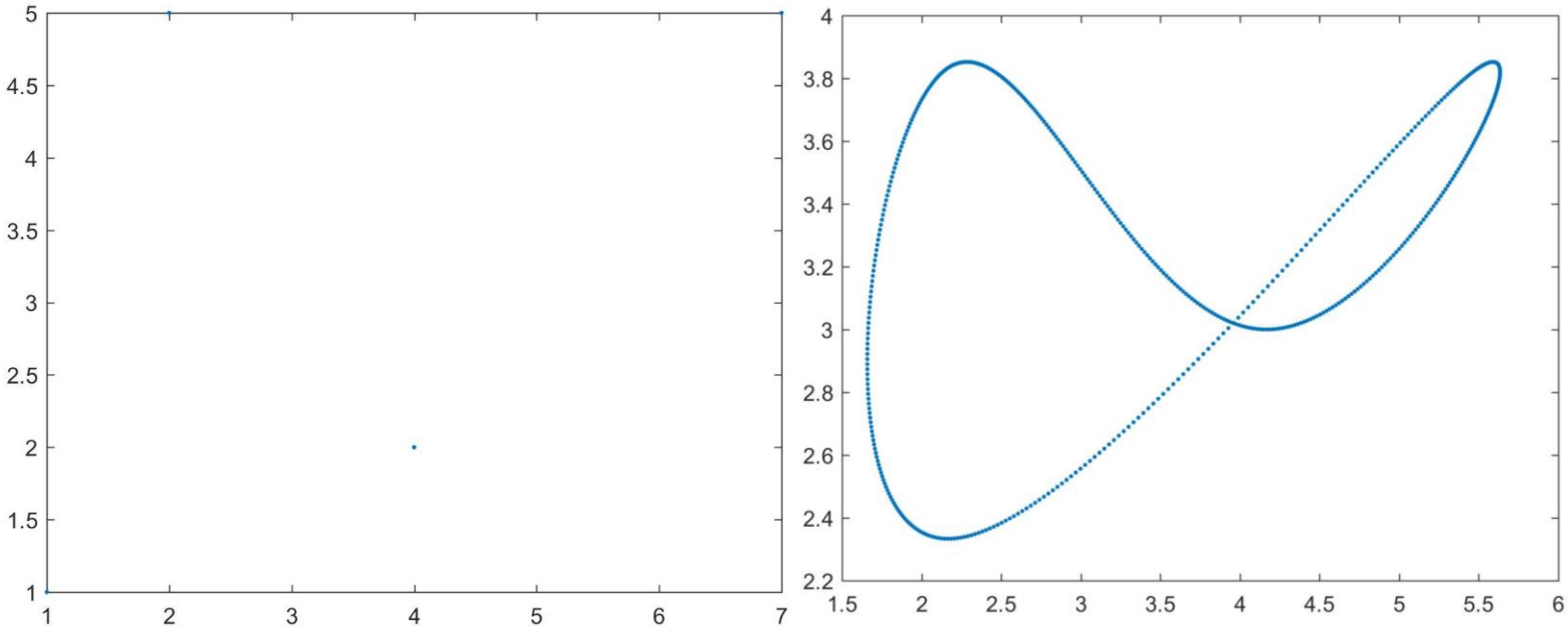
```
xs=splitPts(x);
ys=splitPts(y);

xa=averagePts(xs,w);
ya=averagePts(ys,w);
```

Next, node displacement is calculated and x and y are reassigned to the split and averaged values before the next iteration. Once the loop is exited, the second and final figure is called using the method shown below to allow the original figure to stay open as well.

```
figure();
plot(x,y, '.');
```

1.3 Calculations and Results

When the program is run, the following two figures are displayed

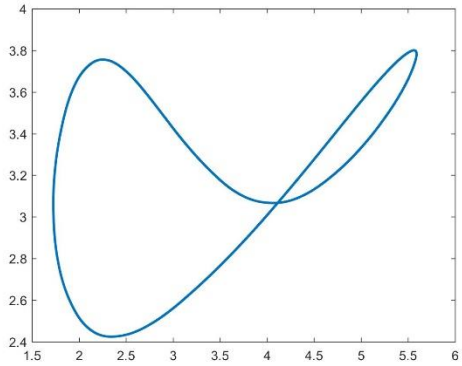


The figure on the left is the plot of the points (1,1), (2,5), (4,2), (7,5). With the weight vector w=[1,2,1], the figure on the right is the result of the splitting and averaging.

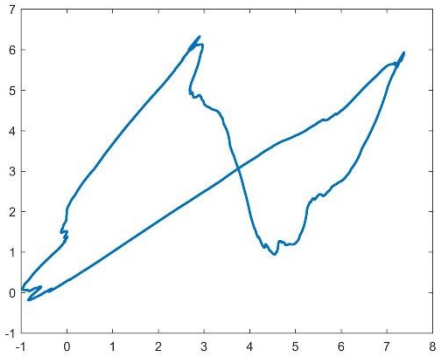
1.4 Discussion

The weight vector has much to do with the convergent shape of the split and averaged points, though initial positions so weigh in heavily when the weight vector isn't changed by much. Take, for example, the initial points and weight vector shown above. By changing the vector from [1,2,1] to [1,2,2], the returned shape looks almost identical, but seems to have a higher distribution of points toward the neck of the figure 8 rather than on the ends which gives a smoother look to the curve, as is shown in the bottom of this section. Both of these take 5 iterations to complete and again give a similar shape, and that seemed to be the case in general. Those values that converged did it all in under 7 iterations, and typically took more than 4. In another case, where the weight vector is [-1,2,-2], the shape will never converge. This may be

due to the fact that rather than have the points draw to each other, the negative values push the points out toward infinity in each direction. Finally, in the case of the weight vector being [-1,2,1], the returned shape is similar to the original, but seems deformed and warped. This only appears to happen in the presence of negative values within the weight vector and each time that is the case a rough shape is presented. This may be due to the rejection of leftward points and the new averaged points are attempting to go toward the right, causing disruptions in the otherwise smooth shape. This example is also displayed below.



W=[1,2,2]



W=[-1,2,1]

2 Numerical Differentiation

2.1 Introduction

The goal of this problem is to perform numerical differentiation on a function using three different methods. These methods are forward differentiation, backward differentiation, and central differentiation. The values derived from the three methods will then be compared to the values found from the true derivative (found by hand) and the error from each approximation will be considered. The error values will then be plotted on the same figure for comparison.

2.2 Model and Methods

The goal in this program is to write a function that uses three methods of numerical differentiation on a function and compares the error in relation to the actual derivative of each. Unlike the previous problem, this problem has many smaller functions in use. The first function to be written is the function function (weird to say, I know), that holds the function to be differentiated. In this case, the function is

```
f=exp((-x.^2)/5).*cos(5*x);
```

Next is the true derivative of the function,

```
df=(-2*x/5).*exp((-x.^2)/5).*cos(5*x)-5*exp((-x.^2)/5).*sin(5*x);
```

Here begins the use of numerical differentiation. The three methods in use are forward, backward, and central approximations. For the sake of saving space, the methods shown will be in relation to the forward method, since they are nearly identical. The forward method is shown below

```
dfx=(fx(x+h)-(fx(x)))/h;
```

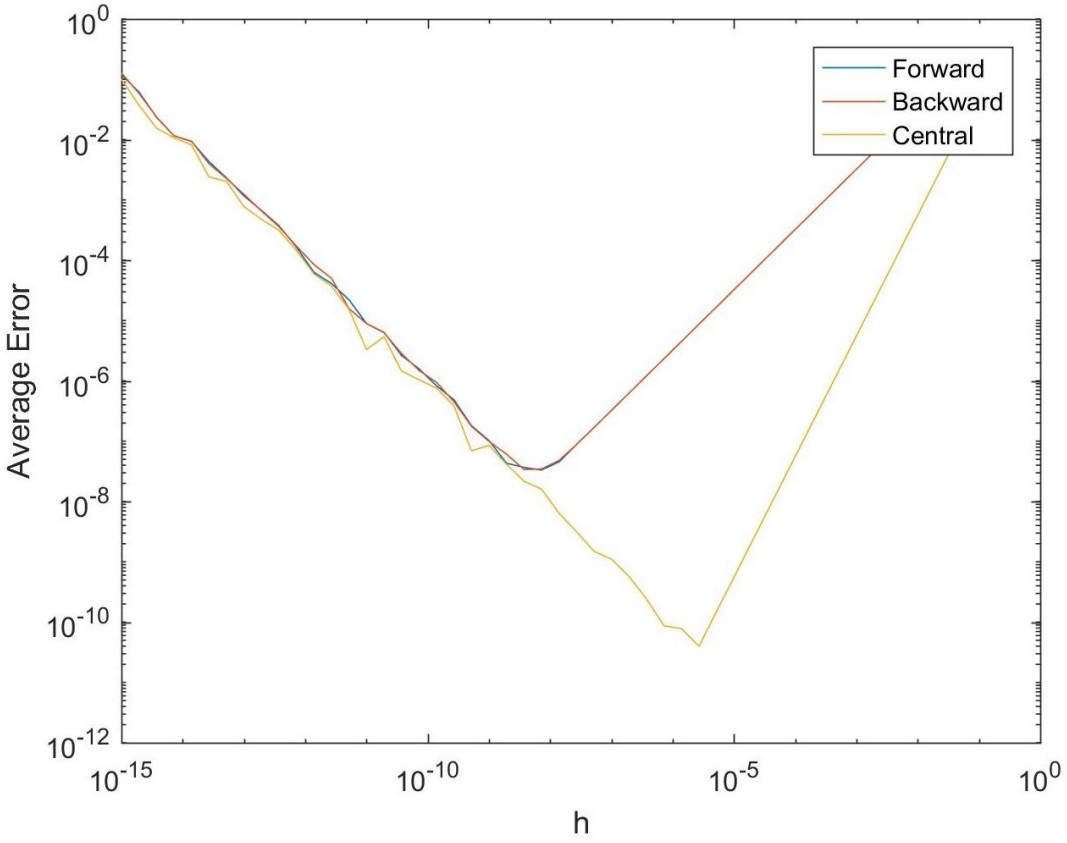
Where h is a real number that will be iterated logarithmically between 10^{-15} and 10^{-1}. This will be done by creating an array of evenly spaced values using the logspace function in MATLAB. This is the first part of the main script following the initialization of the x-range across which the plot function will be analyzed. Next, error arrays are preset using the zeros function with a length equal to the length of the array of h values. The values of the function using the forward method are then found and populated into an array, and the average error is calculated to be the mean absolute error value, the process shown below

```
for k=1:harraylength
approxdforward=dfx_forward(x,harray(k));

errorforward(k)=mean(abs(approxdforward-exactd));
end
```

Again, a similar approach is taken for each method. Finally, each average error value across the h-domain is plotted on the same figure, which is a logarithmic graph created using loglog.

2.3 Calculations and Results

When the program is run, the following figure is displayed



Where the yellow line is the average error from the central difference method, and the blue and red overlapping lines are the average error from the forward and backward difference methods, respectively.

2.4 Discussion

One can see from inspection that until h is in the neighborhood of $10^{-8.5}$ the values are extremely similar, and they also begin to converge once again near 10^{-6} where the error from the central difference method begins to rise. This initial similarity is likely due to the fact that a perturbation near the order of 10^{-10} is extremely small and will likely not cause much error. That trend diverges earlier for the forward and backward difference methods due to their error being on the order of O(h). They will feel the impact of perturbation much more quickly than the central method, which has an error on the order of O(h^2).

By looking at the graph, one can tell which order of error these methods lie on. For the forward and backward methods, they each are order O(h) which can be seen in the slope. The slope of the lines once they begin to increase is close to 1, which on the logarithmic scale means the error and h values are linearly related, implying a first order error. For the central method, the order is O(h^2) which can also be seen in the slope. By similar reasoning this slope near 2 implies a quadratic relationship between error and h, thus implying second order error.

The order of error is important in telling which method is more reliable. In the case of error values being less than zero, which is appropriate here, as the order increases, the value goes to zero faster. This means that higher order errors actually are closer in value to the exact, making them more reliable, In this case, the central method is the most reliable due to its order being higher than that of the forward or backward methods. This should be expected, however, due to the fact that the central method in a way takes the best of each method and roles it into one.