Zubin Mishra
UID: 604644805
CEE/MAE M20
May 9, 2017

**Homework 5**

# 1 The Split-and-Average Problem

## 1.1 Introduction

The goal in this problem is to develop a program that uses the split-and-average approach to subdivide and smooth a surface. This is accomplished by creating two functions, one to split points and one to average points. The beginning conditions and the end results are printed out on the same plot. It will be discussed how different weights affect the process and end results.

## 1.2 Model and Methods

The script first sets up the x, y, and weight arrays. It then plots these initial x and y arrays. The script then sets up the initial conditions for the loop, setting the convergence condition to false and the loop counter to 0. Next, the script goes through a while loop, using the splitting and averaging functions, checking the convergence condition at the end of each loop:

```
while (~converge && i < 15)
    %Split and average, set x and y
    xs = splitPts(x);
    ys = splitPts(y);
    xa = averagePts(xs, w);
    ya = averagePts(ys, w);
    x = xa;
    y = ya;

    %Determine convergence condition
    dx = xa - xs;
    dy = ya - ys;
    converge = (max(sqrt(dx.^2+dy.^2)) < 10^-3);

    %Increment count
    i = i+1;
end
```

The splitPts function uses a for loop with an edge case handled by an if statement:

```
for i = 1:length(x)
        xs(2*i-1) = x(i);
        if i < length(x)
            xs(2*i) = (x(i) + x(i+1))/2;
        else
            xs(2*i) = (x(i)+x(1))/2;
        end
    end
```

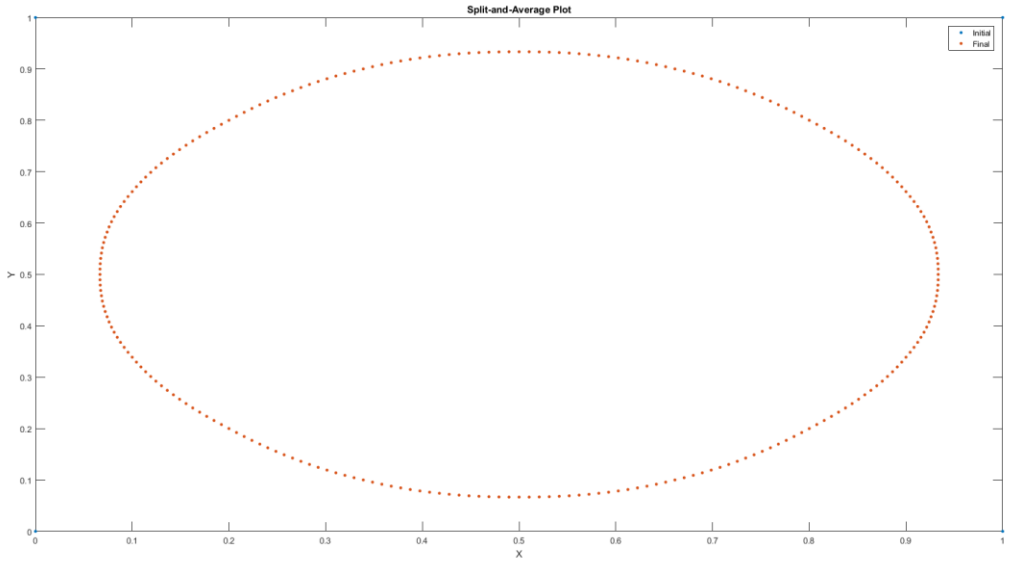
The averagePts function first normalizes the weight array and checks for errors. It then uses a for loop with edge cases handled outside of the loop:

```
xa(1) = norm_w(1)*xs(length(xs)) + norm_w(2)*xs(1) + norm_w(3)*xs(2);
    for i = 2:length(xs)-1
        xa(i) = norm_w(1)*xs(i-1) + norm_w(2)*xs(i) + norm_w(3)*xs(i+1);
    end
    xa(length(xa)) = norm_w(1)*xs(length(xs)-1) + norm_w(2)*xs(length(xs)) +
norm_w(3)*xs(1);
```

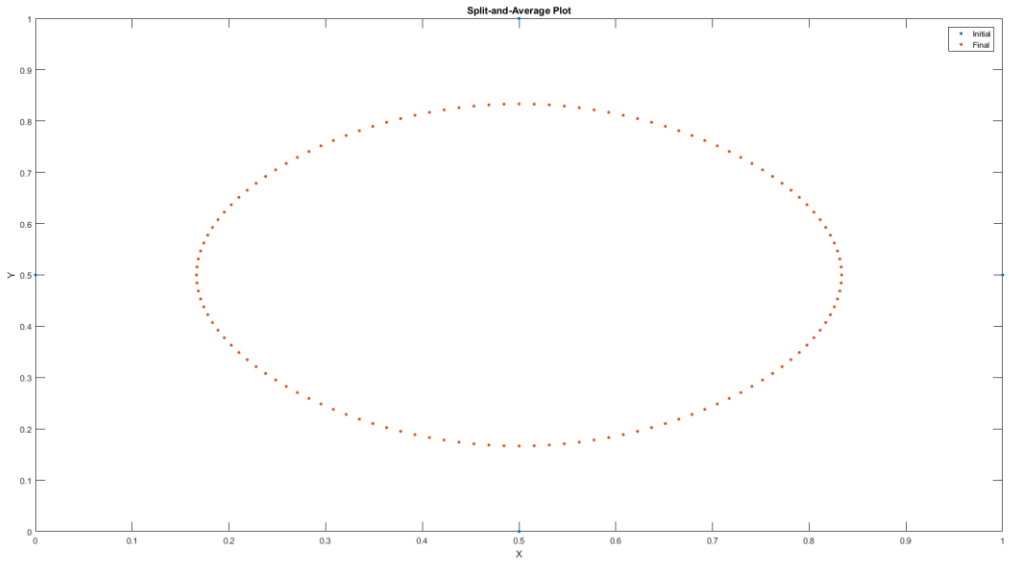
The script concludes with plotting the final arrays.

## 1.3 Results and Calculations

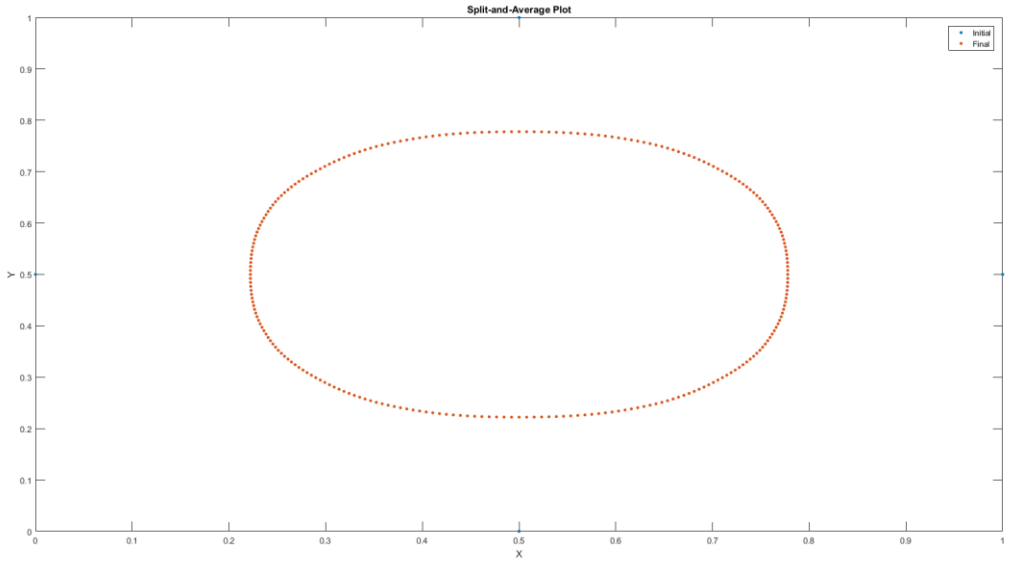
When x = [ 0 0 1 1 ]
y = [ 0 1 1 0 ]
w = [ 1 1 1 ]
The following plot is generated:



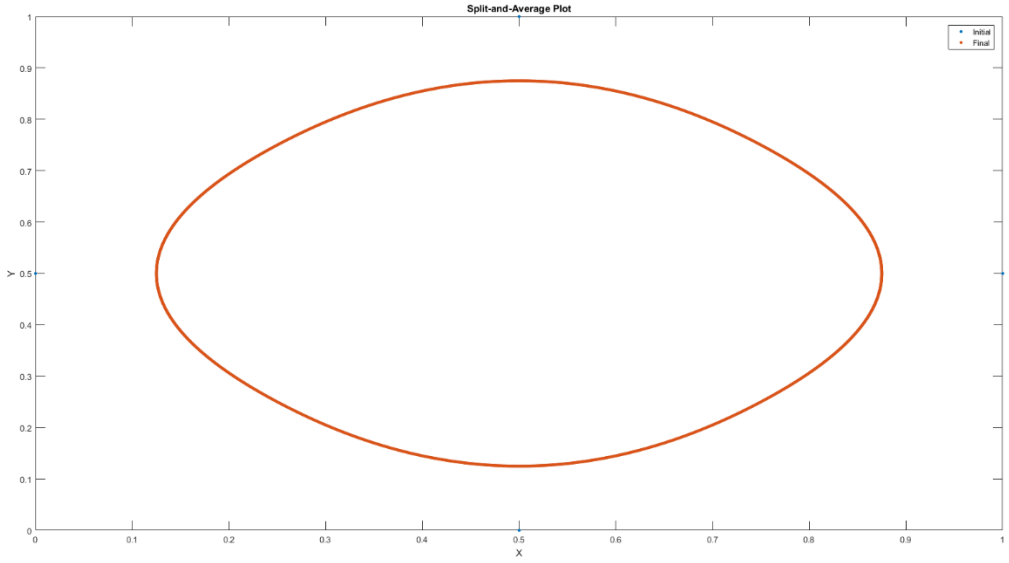
When x = [ 0.5 0 0.5 1 ]
y = [ 0 0.5 1 0.5 ]
w = [ 1 2 1 ]



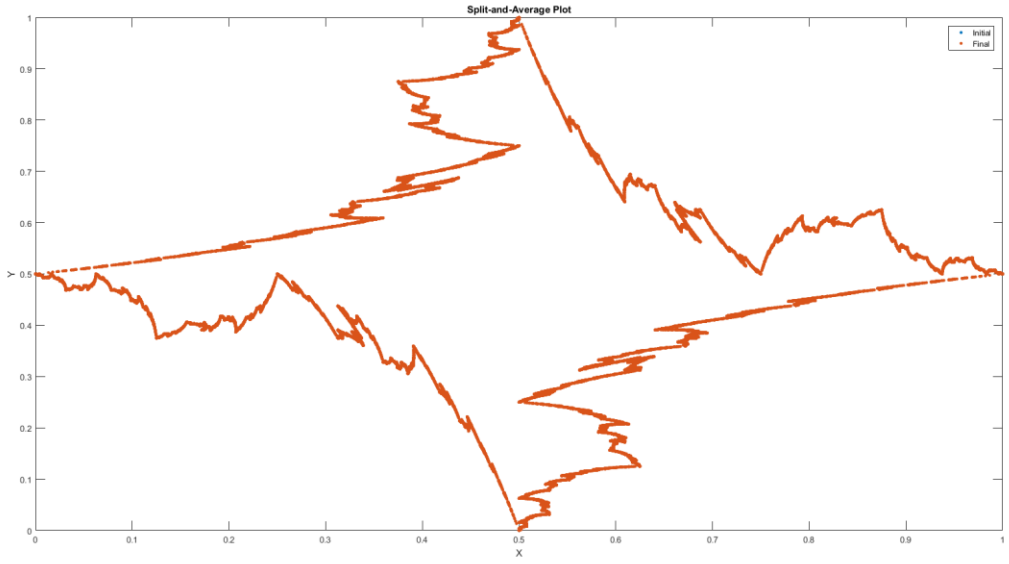

When w = [ 2 1 2 ]



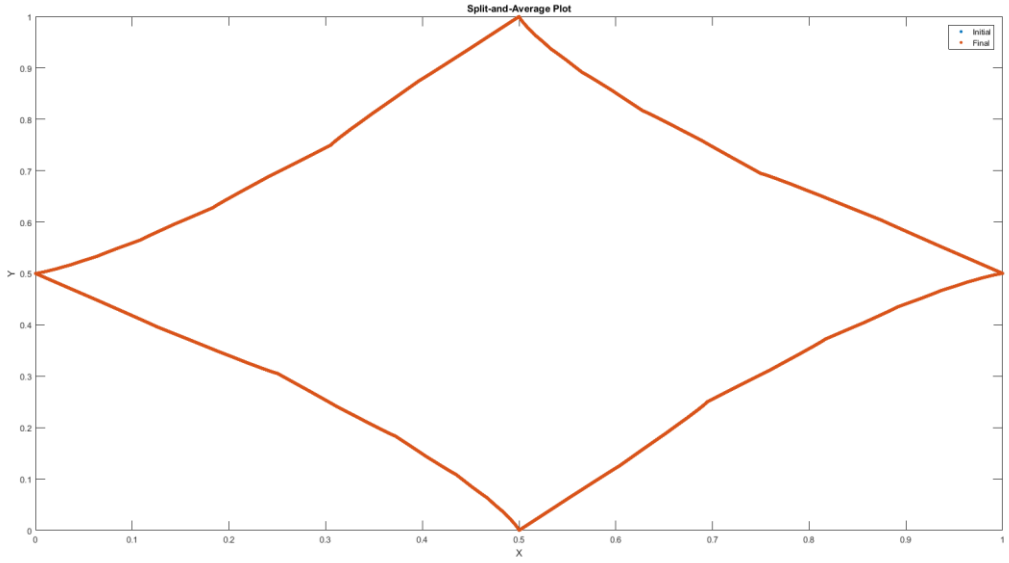
When w = [ 0 1 1 ]



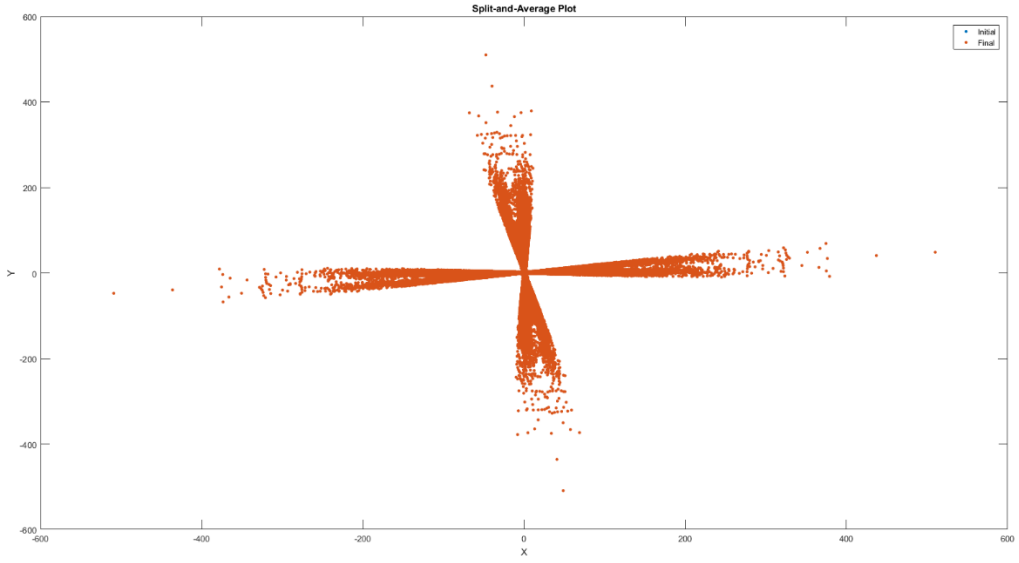

When w = [ 1 -2 3 ] (15 iterations)



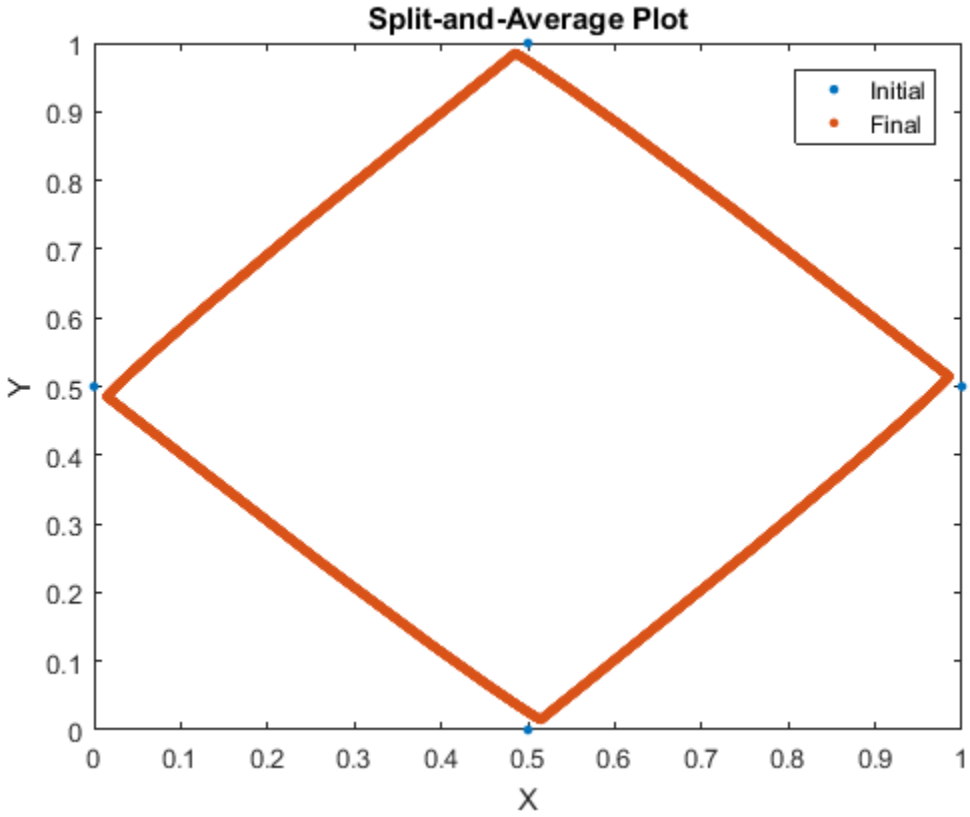
When w = [ 1 -2 10 ]




When w = [ -1 5 -6 ] (15 iterations)



When w = [ 1 1 99 ]




1.4 Discussion

When all the weights are positive, the split-and-average approach will typically converge in around 10 or less iterations. The same happens when all the weights are negative, as when the weights get normalized, it becomes as if they were all positive to begin with. In these cases, the final converged shape typically looks like a rounded, smaller version of the original shape (w = [ 2 1 2 ]). However, if the majority of the weight is on one point (w = [ 1 1 99 ]), then the final converged shape looks very close to the original.

When some of the weights are negative, convergence is not always reached. In the cases where convergence is reached, it typically takes fewer than 15 iterations. For example w = [ 1 -2 10] took 12 iterations. In these cases, the shape tends to be more angular. When convergence is not reached, there tends to be a fractal shape after 15 iterations. In the case of w = [ -1 5 -6 ], no shape is readily distinguishable.

## 2 Numerical Differentiation

### 2.1 Introduction

The goal in this problem is to develop a program that approximates the derivative of a function using the forward, backward, and central difference approximations and calculates and plots the errors for different increments. This is accomplished by creating three functions, one for each approximation, plus one additional function for the exact derivative. It will be discussed how the error scales with the increment.

### 2.2 Model and Methods

The script begins by setting up the arrays for x and h using linspace and logspace respectively. The error arrays for each approximation method are initialized as well. The exact derivative array is then created using a function. The script then iterates through a for loop for each increment value, calculating average error for each:

```
for i = 1:50;
    %Performs difference approximations
    dfx_f = dfx_forward(x, h(i));
    dfx_b = dfx_backward(x, h(i));
    dfx_c = dfx_central(x, h(i));

    %Calculates the error for a h value
    err_f(i) = mean(abs(dfx_f-dfx_e));
    err_b(i) = mean(abs(dfx_b-dfx_e));
    err_c(i) = mean(abs(dfx_c-dfx_e));
end
```

The script then plots the increments against the errors with log-log scaling.

The forward difference approximation function follows the following formula:

$$f'(x_i) \approx \frac{f(x_i + h) - f(x_i)}{h}$$

The backward difference approximation function follows the following formula:

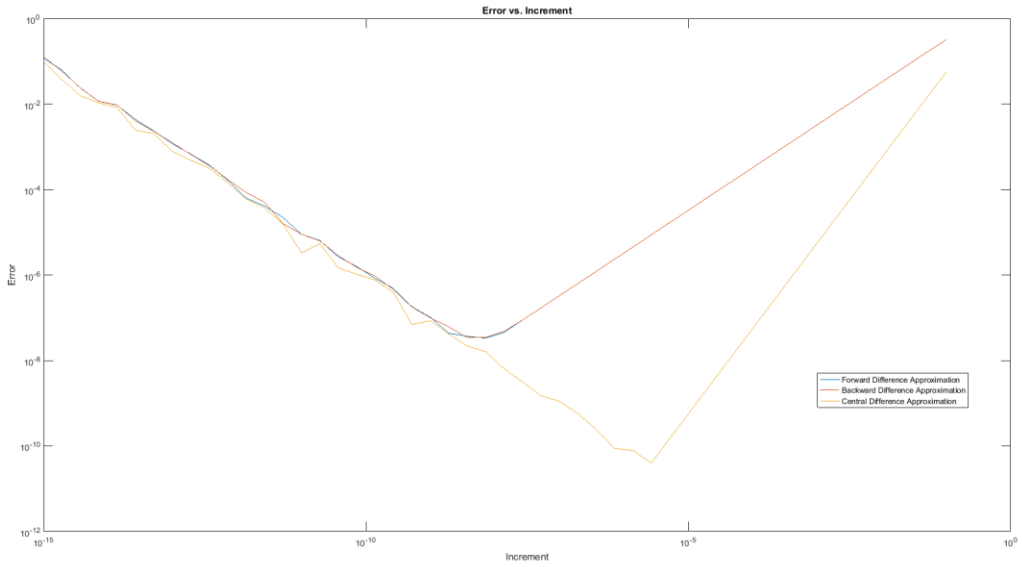
$$f'(x_i) \approx \frac{f(x_i) - f(x_i - h)}{h}$$

The central difference approximation function follows the following formula:

$$f'(x_i) \approx \frac{f(x_i + h) - f(x_i - h)}{2h}$$

## 2.3 Results and Calculations

After running the script, the following plot is generated:




## 2.4 Discussion

The plot generated can be used to verify the scaling of the error by looking at the slopes of the lines generated in the area of the graph where there are clean lines (increment > $10^{-5}$). If the error and the increment follow some relation of the form

$$Error = ah^k$$

Then in log-log plot will yield the equation

$$E = k \cdot H + b$$

The slope is equal to the power.

We can see that for forward and backward differencing, the slope is approximately 1, indicating that the error scales linearly with the increment. We can see that for central differencing, the slope is steeper, approximately 2, indicating that the error scales quadratically with the increment.

When the increment gets very small, the error starts increasing, and the line plot becomes jagged. This is likely due to rounding errors resulting from MATLAB's finite precision, which get worse as the increment gets smaller, because more information gets lost when performing the approximations.