Jeffrey Ding
UID: 104928991
CEE M20
May 5th, 2018

Homework 5

1. The Split-and-Average Problem

1.1 Introduction

The goal of this problem is to subdivide and smooth the surfaces given a set of x and y coordinates.

1.2 Model and Methods

The script begins by creating x, y, and w arrays such as this:

```
x = [0.5, 0, 0.5, 1];
y = [0, 0.5, 1, 0.5];
w = [1, 2, 1];
```

The script also sets a variable to track the difference in maxDisplacement. Then, the script copies the initial x and y coordinates into new variables for future plotting using the following code:

```
initialX = x;
initialY = y;
```

Then, the script runs a while loop that runs when maxDisplacement is larger than the designated tolerance like this:

```
while maxDisplacement > 10^-3
...
end
```

In the while loop, xs and ys arrays are created by calling the splitPts function. The splitPts function first determines the size of the input array using the following code:

```
N = length(x);
```

Then, the function creates a return vector, xs, twice the size of x. The original points are copied into the odd indexes of the new vector using the following for loop:

```
xIndex = 1;
for k = 1:2:N*2
   xs(k) = x(xIndex);
   xIndex = xIndex + 1;
end
```

The even indexes are then filled by calculating the averages of the neighboring elements using the following for loop:

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

Then, in the main script, xa and ya arrays are created and filled using the averagesPts function. The averagePts function first checks if the w vector is legal using the following if statements:

```
if length(w) ~= 3
   error("Vector w not of correct size");
elseif sum(w) == 0
   error("Sum of the three weights is equal to zero");
end
```

Then, the function determines the size of the input vector and creates a return vector. Then, the weighted averages are calculated using the following code:

```
w1 = w(1)/sum(w);
w2 = w(2)/sum(w);
w3 = w(3)/sum(w);
```

Next, the function fills in the vector using the following for loop:

```
for k = 1:N
   if k == 1
      xa(k) = w1*xs(N) + w2*xs(k) + w3*xs(k+1);
   elseif k == N
      xa(k) = w1*xs(k-1) + w2*xs(k) + w3*xs(1);
   else
      xa(k) = w1*xs(k-1) + w2*xs(k) + w3*xs(k+1);
```

```
    end
end
```

Then, the main script replaces the x and y vectors with the newly calculated xa and ya vectors. Then, the script calculates the difference between the average and split values and the maxDisplacement with the following code:

```
dx = xa - xs;
dy = ya - ys;
maxDisplacement = max(sqrt(dx.^2 + dy.^2));
```

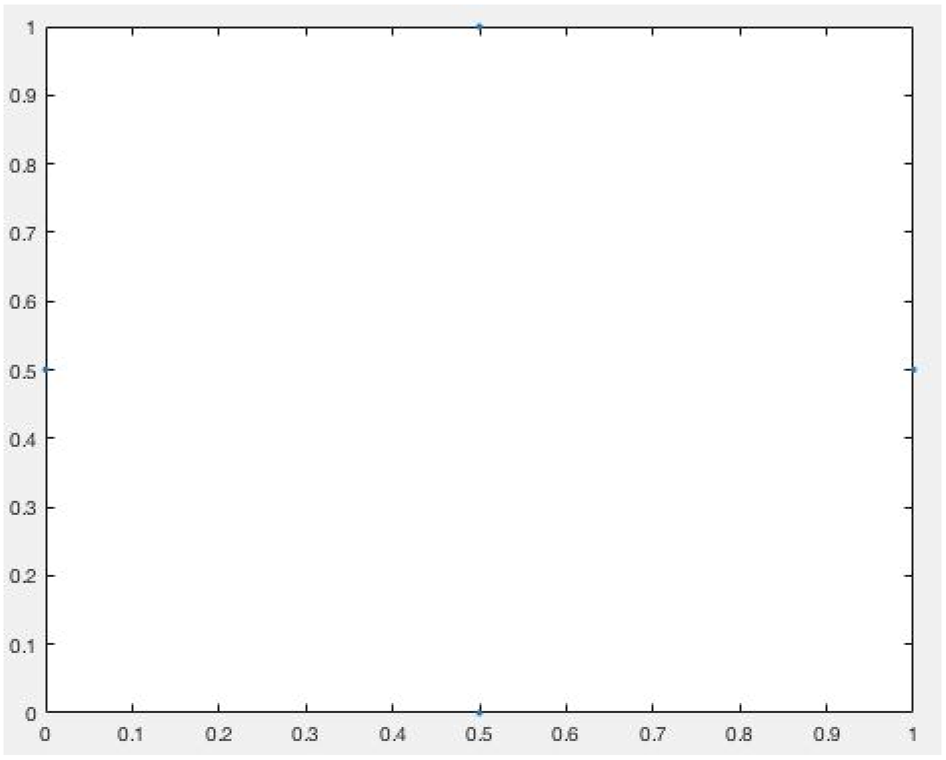
Repeat this process until the maxDisplacement is too small. Then, the script creates two plots, figure 1 being the initial plot and figure 2 being the finished plot.

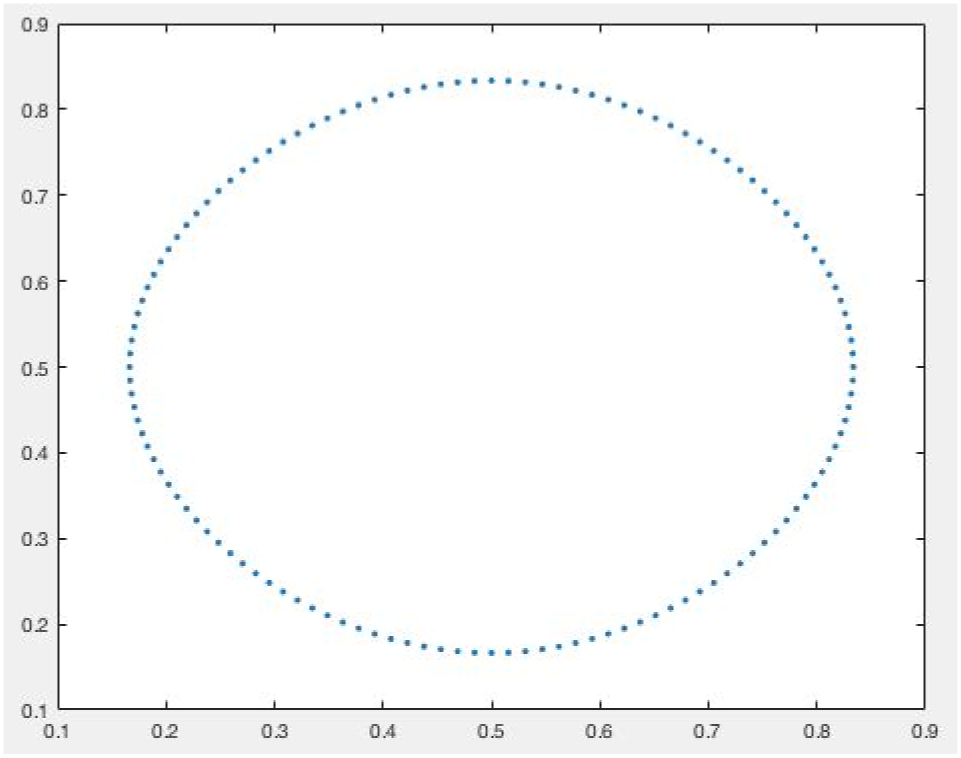
1.3 Calculations and Results

When the program is executed with the following inputs:

```
x = [0.5, 0, 0.5, 1];
y = [0, 0.5, 1, 0.5];
w = [1, 2, 1];
```

The following graphs are created:





The output shows are very smooth circle

1.4 Discussion

Given the ease of the inputs, the output figure 2 is a very good representation of what the program should do. It smooths the edges and approximates a circle based on the weightings given.

However, as we play with the weighting, the shape can greatly change and contort into interesting shapes. As the weighting becomes more and more uneven, which means the weighting doesn't follow the pattern [a, b, a], the shape contorts into elongated ellipses. Furthermore, if we make one or more weightings negative, it contorts the shapes into sharp shapes, albeit the approximation is still smooth.

From all of the weightings that I have tried, all weight values lead to a convergence. Then only weight values that do not lead to a convergence would be those where the sum of the weights add up to zero. However, this case is protected against in the script.

For most weight values, it takes roughly 10 iterations for my method to converge.

2. Runga-Kutta Radioactivity

2.1 Introduction

The goal of this problem is to approximate carbon-15 levels using three different types of approximations: explicit Euler, second-order, and Runga-Kutta methods.

2.2 Model and Methods

The script begins by calculating the real values at each timestep using the following lines of code:

```
t1 = linspace(0, 15, 16);
t2 = linspace(0, 15, 151);
t3 = linspace(0, 15, 1501);
real1 = exp(-(log(2)/2.45)*t1);
real2 = exp(-(log(2)/2.45)*t2);
real3 = exp(-(log(2)/2.45)*t3);
```

t1 corresponds to timesteps of 1 second, t2 corresponds to timesteps of 0.1 seconds, and t3 corresponds to timesteps of 0.01 seconds.

Then, the script creates an array dt that stores the different time steps. Afterwards, the script creates 9 different arrays full of ones to store the approximations:

```
% dt = 1.00
rk11 = ones(1, 16);
rk12 = ones(1, 16);
rk13 = ones(1, 16);

% dt = 0.10
rk21 = ones(1, 151);
rk22 = ones(1, 151);
rk23 = ones(1, 151);

% dt = 0.01
rk31 = ones(1, 1501);
rk32 = ones(1, 1501);
rk33 = ones(1, 1501);
```

Then, the script calculates all of the arrays using the following for loops to stimulate different timesteps:

```
% calculate for dt = 1.00
for t = 2:16
```

```
    rk11(t) = advanceRK(rk11(t-1), dt(1), 1);
    rk12(t) = advanceRK(rk12(t-1), dt(1), 2);
    rk13(t) = advanceRK(rk13(t-1), dt(1), 4);
end

% calculate for dt = 0.10
for t = 2:151
    rk21(t) = advanceRK(rk21(t-1), dt(2), 1);
    rk22(t) = advanceRK(rk22(t-1), dt(2), 2);
    rk23(t) = advanceRK(rk23(t-1), dt(2), 4);
end

% calculate for dt = 0.01
for t = 2:1501
    rk31(t) = advanceRK(rk31(t-1), dt(3), 1);
    rk32(t) = advanceRK(rk32(t-1), dt(3), 2);
    rk33(t) = advanceRK(rk33(t-1), dt(3), 4);
end
```

Each time, the advanceRK function is called. The function runs by calculating the differential equation of carbon-15 and then applying it to c1, c2, c3, and c4 like this:

```
dydt = ((-log(2))/2.45)*y;
c1 = dt*dydt;
dydt = ((-log(2))/2.45)*(y+0.5*c1);
c2 = dt*dydt;
dydt = ((-log(2))/2.45)*(y+0.5*c2);
c3 = dt*dydt;
dydt = ((-log(2))/2.45)*(y+c3);
c4 = dt*dydt;
```

The first, c1, is specifically for explicit Euler. The second, c2, is specifically for second order approximation. When all four are used, the approximation in question is Runga-Kutta.

After approximations, the function checks the method variable and determines which method to run. Here is the following switch statement:

```
switch method
    case 1
        y = y + c1;
    case 2
        y = y + c2;
    case 4
        y = y + (1/6)*c1 + (1/3)*c2 + (1/3)*c3 + (1/6)*c4;
    otherwise
        error("Invalid method number");
end
```

Then, the script calculates the absolute error between the approximations and the real values using the following code:

```
error11 = abs(rk11 - real1);
error12 = abs(rk12 - real1);
error13 = abs(rk13 - real1);
error21 = abs(rk21 - real2);
error22 = abs(rk22 - real2);
error23 = abs(rk23 - real2);
error31 = abs(rk31 - real3);
error32 = abs(rk32 - real3);
error33 = abs(rk33 - real3);
```

Finally, the script prints out the average error and plots each graph at different timesteps:

```
fprintf('  dt        RK1          RK2          RK4\n')
fprintf('1.00:   %.2e     %.2e      %.2e\n', mean(error11), mean(error12), mean(error13));
fprintf('0.10:   %.2e     %.2e      %.2e\n', mean(error21), mean(error22), mean(error23));
fprintf('0.01:   %.2e     %.2e      %.2e\n', mean(error31), mean(error32), mean(error33));

% plots dt = 1.00
figure(1);
plot(t1, rk11, t1, rk12, t1, rk13, t1, real1);

% plots dt = 0.10
figure(2);
plot(t2, rk21, t2, rk22, t2, rk23, t2, real2);

% plots dt = 0.01
figure(3);
plot(t3, rk31, t3, rk32, t3, rk33, t3, real3);
```

2.3 Calculations and Results

When the program is executed the following output is printed to the screen:

```
  dt        RK1          RK2          RK4
```

1.00:	3.11e-02	3.42e-03	1.38e-05
0.10:	3.09e-03	2.95e-05	1.18e-09
0.01:	3.08e-04	2.91e-07	1.16e-13

Figure 1

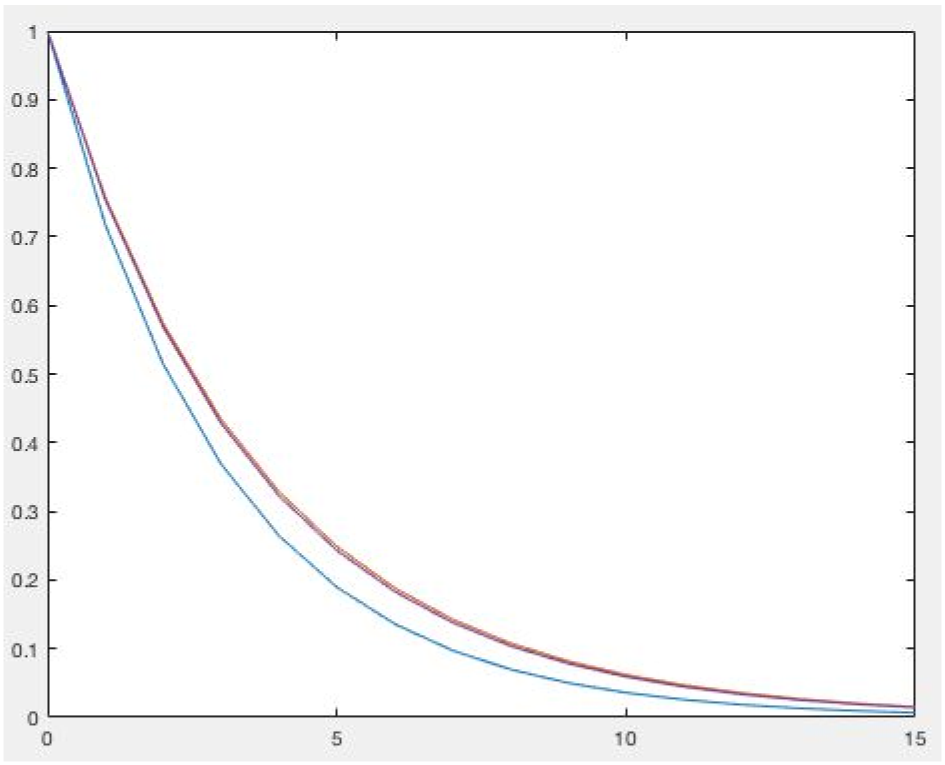


Figure 2

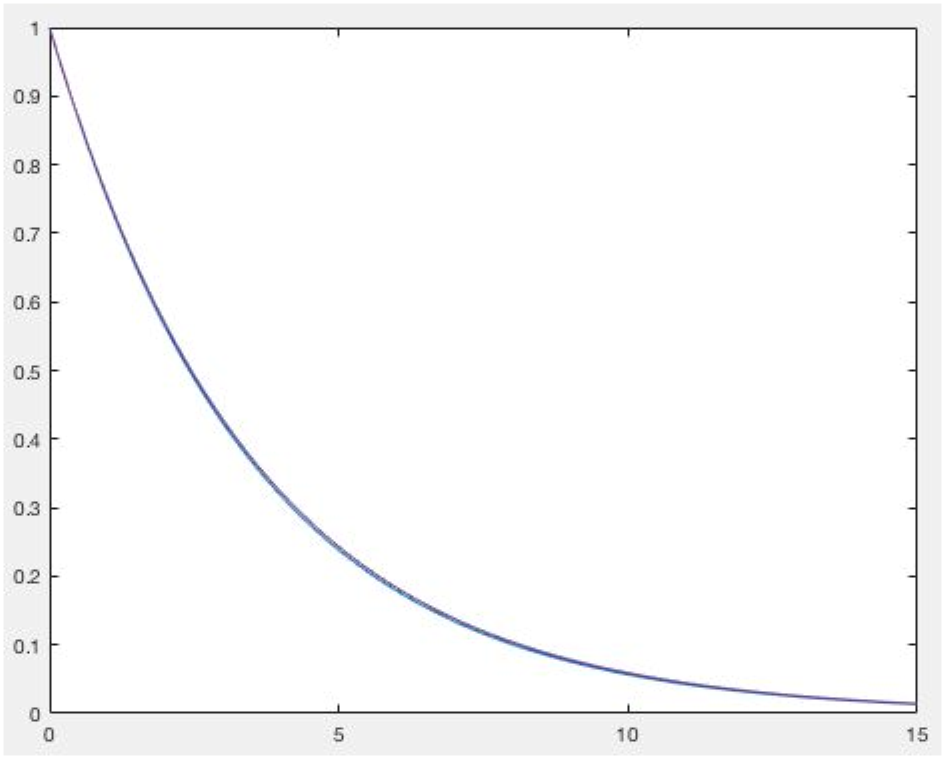
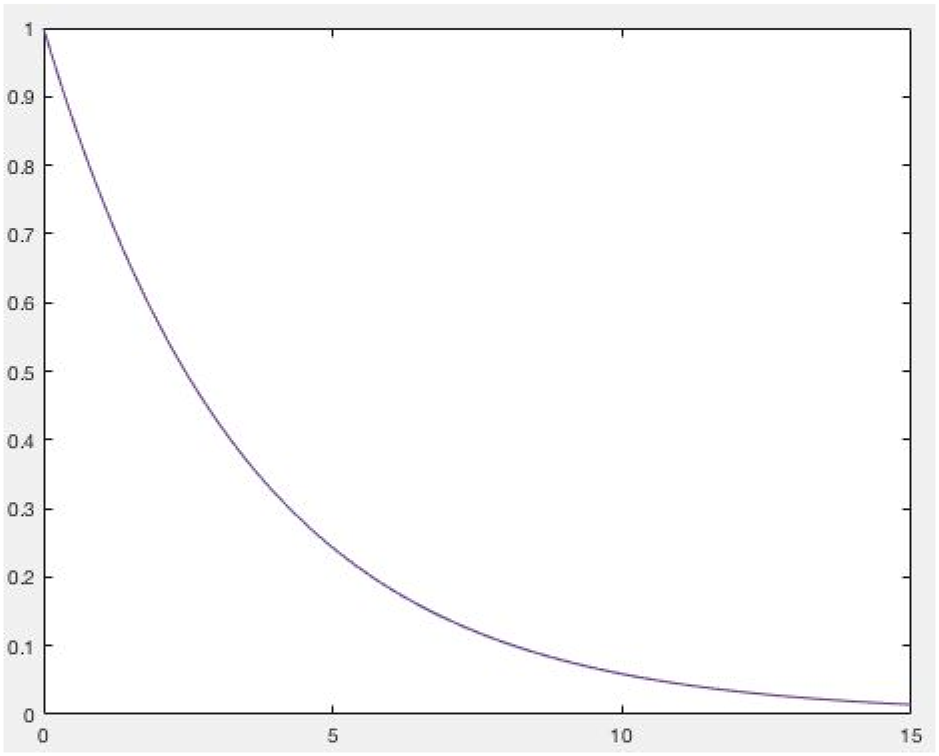


Figure 3



The outputs look really closely approximated, especially as t approaches 15.

2.4 Discussion

The outputs look approximated, especially as time approaches 15 seconds. Also, as the timesteps becoming smaller and smaller, the approximation also becomes more and more accurate. This isn’t a surprise, as the error margin becomes smaller as well as timesteps are decreased. Also, as we add more orders of approximation, the margin of error and the approximation becomes smaller and more accurate, respectively.

Using my most accurate numerical method, approximately 1.44% of carbon-15 remains after 15 seconds.

By examining the scaling of the average errors, we can see that the three different methods became more and more precise as we added orders of magnitude, as previously stated. In fact, everytime we added a order of magnitude, the margin of error almost always decreased by at least an entire magnitude.

When we take a very large timestep such as 5, the approximation becomes more inaccurate. However, the same principles still apply, in that the more orders of approximation we use, the more precise the results are, albeit that are still inaccurate. This occurs because 1st order is a very simple way to approximate change. However, RK4 still calculates different timesteps to better approximates the change.