M20 Homework 4

Colin Skinner - 505795313

2022/07/22

1 The Split-and-Average Problem

1.1 Introduction

The goal of this problem is to split and average two arrays of points corresponding to the x and y coordinates of a graph, essentially smoothing them. To do so, a function called `splitPts` is used to insert a value between neighboring values equal to their average, and a function called `averagePts` is used to compute a weighted average of each value in the array with itself and the two values next to it, additionally taking in an array of weights. This is called on both arrays until the maximum displacement of a split-and-average iteration is less than 1×10^3. The code then plots both the original points and the split-and-averaged points on a single graph.

1.2 Model and Theory

The following equations are applicable for this problem when splitting the array:

$$x = [x_1, x_2, ...x_n] \tag{1}$$

$$x_s = [x_1, \frac{x_1 \times x_2}{2}, x_2, ..., x_{n-1}, \frac{x_{n-1} \times x_n}{2}, x_n] \tag{2}$$

Where

- x = Initial array of size n
- x_s = Split array of size $2n$

Then, in order to average these arrays, the following function is used:

$$x_a(k) = w_1 \times x_s(k-1) + w_2 \times x_s(k) + w_3 \times x_s(k+1) \tag{3}$$

Where

- x_a = Split array in Equation 2
- $x_a(k)$ = kth value of the split array
- w = Weights in an array of size 3

1.3 Methods and Pseudo-code

The main `MATLAB` script should carry out the following pseudo-code:

1. Initialize starting arrays for x and y coordinates
2. Plot initial points on a graph
3. Initialize `error` to be 1 (greater than the error stop condition for the while loop to follow)
4. Initialize `count` to be 0 for the number of iterations
5. While the error is greater than 1×10^3 and the iteration count is less than 50 (in the case that it does not converge by then)

    - Split points of x and y arrays
    - Average points of x and y arrays into new arrays xa and ya
    - Set error equal to the maximum of the Cartesian distance between each respective point before and after it had been averaged
    - Set x to xa and y to ya for the next iteration, and increment *count*

6. Plot new graph of points

The function `splitPts` should carry out the following pseudo-code:

1. Input: Array x
2. Initialize array xs with twice the size of x
3. Loop for every element in xs except the last
    - If respective index of xs is odd, it is part of the original array
    - If respective index of xs is even, it is the average of the two numbers it is between
4. The value at the end of xs is the average of its predecessor and the first index

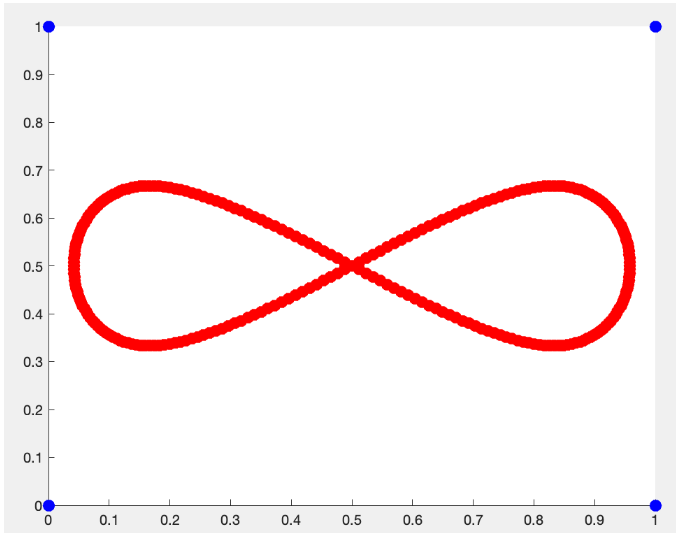
---

The function `averagePts` should carry out the following pseudo-code:

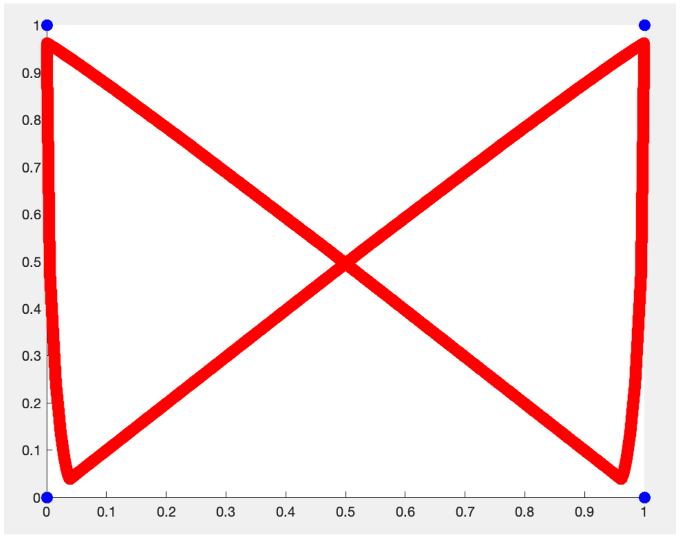
1. Inputs: Array xs, weights array w
2. If the sum of weights is 0, output error, else divide all values by the sum of w
3. Create array xa with the same length as xs
4. The first element in xa is equal to the sum of the last element, itself, and the second element of xs multiplied by the first, second, and third weights respectively
5. The last element in xa multiplies the weights with itself, the second-to-last element, and the first
6. Loop for every other value in xs besides first and last
    - That respective index of xa is equal to the sum of the previous value, current value, and next value multiplied by the three weights respectively

1.4 Calculations and Results

Script output with $x = [0, 0, 1, 1]$ and $y = [0, 1, 0, 1]$ and a weights array $w = [1, 2, 1]$:



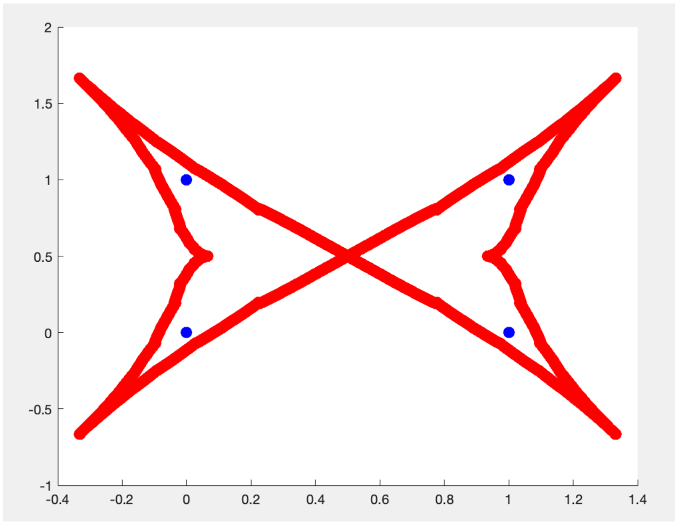
Modified output with $w = [1, 2, 100]$:



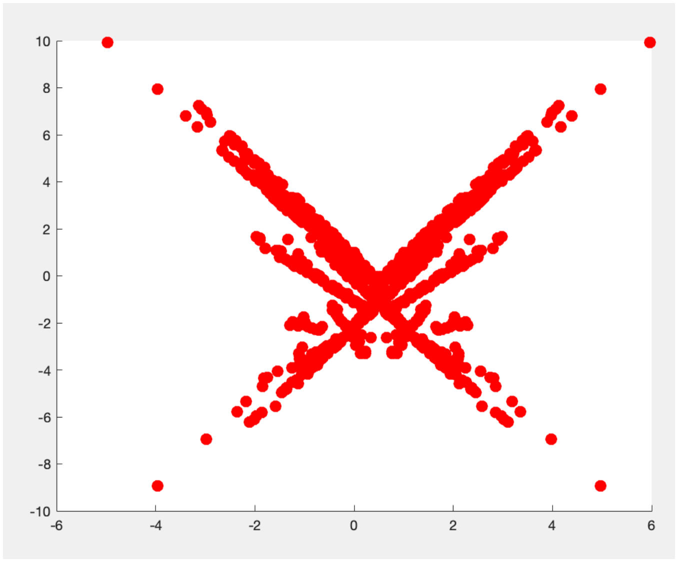
Modified output with $w = [1, -2, 1]$ (to induce error):

```
Which problem to test?
1
Error using averagePts
Sum of weights cannot be 0
```

Modified output with $w = [1, -7, 1]$:



Modified output with $w = [-1, 10, -5]$ (loop set to stop after 8 iterations):



1.5 Discussion and Conclusion

The initial conditions of a square converged to an infinity symbol for all weights tested. Typical number of iterations was 6 when the weight was [1,h,1] for h values from 0 to 30 (except when h was 2 and the iterations was 5). The iterations only decreased as the middle value h increased, and at 100, the iterations was only 4. With the first two weights at 1 and 2, the typical number of iterations rose but plateaued at 10 after the third weight was 7 or higher (even when it was 100 with [1,2,100] weights). Finally there were no positive values of weights tested that did not converge to have an error smaller than 1×10^3.

However, when negative weights are involved, the graphs look more complicated, no longer always smooth and often jagged. With [1,h,1] for h values from -3 to -6, the graph will never converge according to the error stop condition, and for h = -2, the weights array would sum to 0, producing an error in the `averagePts` function. If any of the two outer values are set to a negative weight besides -1, a graph does not form without the iteration counter breaking the loop, suggesting that the outer two values need to be positive in order for a smooth curve to form. In the 8-iteration output for weights [-1,10,-5], the graph is very jagged and does not appear to be converging on one shape.

2 Runge-Kutta Radioactivity

2.1 Introduction

The goal of this problem is to evaluate the accuracy of several Runge-Kutta discretized approximations for the half-life of Carbon-15–a mere 2.45 seconds. The values changed in the calculations are the degree of the equation (first, second, and fourth degrees) and the width of the time step (1, 0.1, 0.01 seconds). The code will utilize a function called `advanceRK` which will advance a step in the discretized equations until a final time is reached. Then for each time step, the script will display the average error of all three approximations and plot their values along with the results of the actual equation.

2.2 Model and Theory

The differential equation to model the half-life of Carbon-15, and its solution, are modeled by the following equations:

$$\frac{dy}{dt} = -\frac{ln(2)}{t_{1/2}}y \tag{4}$$

$$y(t) = y_0\ exp\left(-\frac{ln(2)}{t_{1/2}}t\right) \tag{5}$$

Where:

- $\frac{dy}{dt}$ = Change in Carbon-15 over time
- $t_{1/2}$ = Half life of Carbon-15 (in seconds), which was set to 2.45 in the script
- y_0 = Initial amount of Carbon-15
- t = Time elapsed

To integrate these functions, four orders of Runge-Kutta equations are used.

$$\begin{array}{c|rl}
\textsf{RK1} & c_1 & = \Delta t f\left(t_k, y_k\right) \\
 & y_{k+1} & = y_k + c_1 \\
\hline
 & c_1 & = \Delta t f\left(t_k, y_k\right) \\
\textsf{RK2} & c_2 & = \Delta t f\left(t_k + \frac{1}{2}\Delta t, y_k + \frac{1}{2}c_1\right) \\
 & y_{k+1} & = y_k + c_2 \\
\hline
 & c_1 & = \Delta t f\left(t_k, y_k\right) \\
 & c_2 & = \Delta t f\left(t_k + \frac{1}{2}\Delta t, y_k + \frac{1}{2}c_1\right) \\
\textsf{RK4} & c_3 & = \Delta t f\left(t_k + \frac{1}{2}\Delta t, y_k + \frac{1}{2}c_2\right) \\
 & c_4 & = \Delta t f\left(t_k + \Delta t, y_k + c_3\right) \\
 & y_{k+1} & = y_k + \frac{1}{6}c_1 + \frac{1}{3}c_2 + \frac{1}{3}c_3 + \frac{1}{6}c_4
\end{array}$$

Where

- Δt = Time step
- RK1-4 = Order of Runge-Kutta equation
- c_1-c_4 = Constants of iteration for respective orders 1-4
- y_k = Current value of solution equation
- y_{k+1} = Next value of solution equation

2.3 Methods and Pseudo-code

The `MATLAB` script should carry out the following pseudo-code:

1. Set initial condition y_0 to be 1
2. Initialize dt array with 3 values: 1, 0.1, 0.01
3. Display line with column headers dt, $RK1$, $RK2$, and $RK4$.
4. Loop that iterates over all values in the dt array (3 times)
   (a) Create time array from $0 \leq t \leq 15$ seconds iterating by the loops's respective value of dt
   (b) Compute exact solution of the half-life equation on all points of the time array and store in an array
   (c) Preallocate 3 arrays $y1$, $y2$, and $y4$, for RK1-4 with the same size as time array and set initial condition to y_0
   (d) For loop that iterates over all values in the time array (minus the last)
      - Call *advanceRK* function for all three methods, and set their result to next values of $y_{1,2,4}$ arrays
   (e) Display the outer for loop's value of dt and the average errors between each Runge-Kutta Method approximation value and the actual value
   (f) Plot the exact function and all three estimates on one plot

---

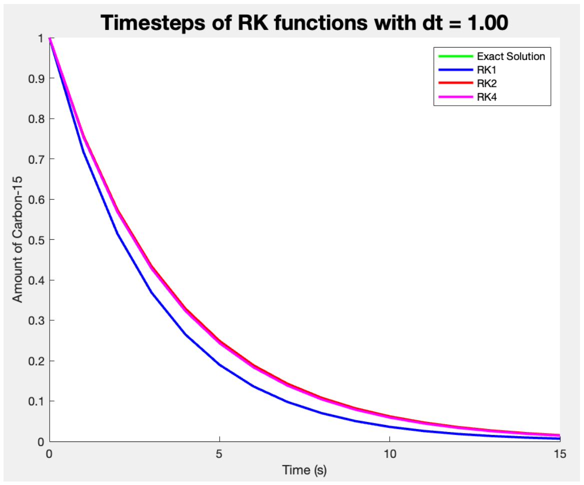
The function `advanceRK` should carry out the following pseudo-code:

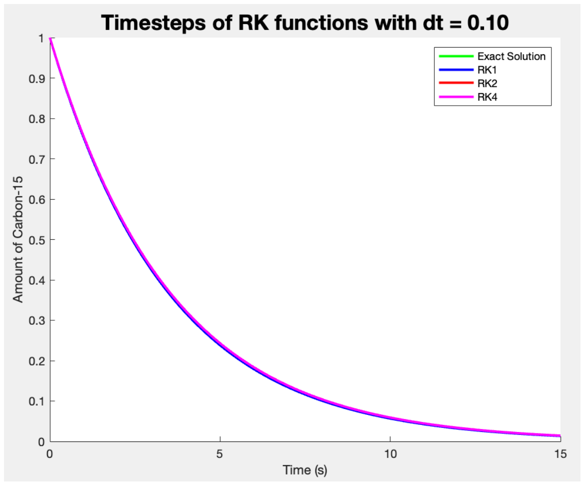
1. Inputs: y = initial amount (of Carbon-15), dt = time step, *method* = order of RK approximation
2. In the case of 1:
   - Calculate the first constant of iteration c_1 and add it to y
3. In the case of 2:
   - Calculate the first constant of iteration c_1
   - Calculate the second constant of iteration by considering c_1, and add it to y
4. In the case of 4:
   - Calculate the first constant of iteration c_1
   - Calculate the second constant of iteration by considering c_1
   - Calculate the third constant of iteration by considering c_2
   - Calculate the fourth constant of iteration by considering c_3
   - Add $c_1/6$, $c_2/3$, $c_3/3$, and $c_4/6$ to y
5. Otherwise display error if method does not match one of the three

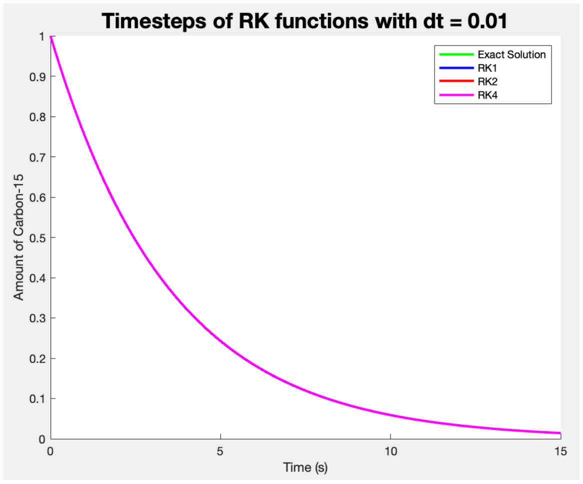
2.4 Calculations and Results

Script output for the default value $y_0 = 1$

```
Which problem to test?
2
   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
```

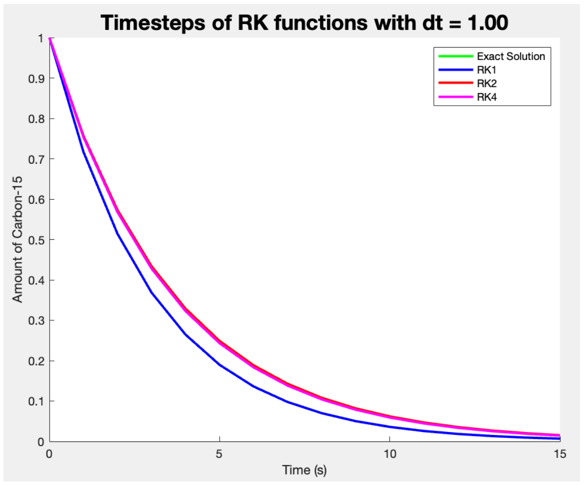




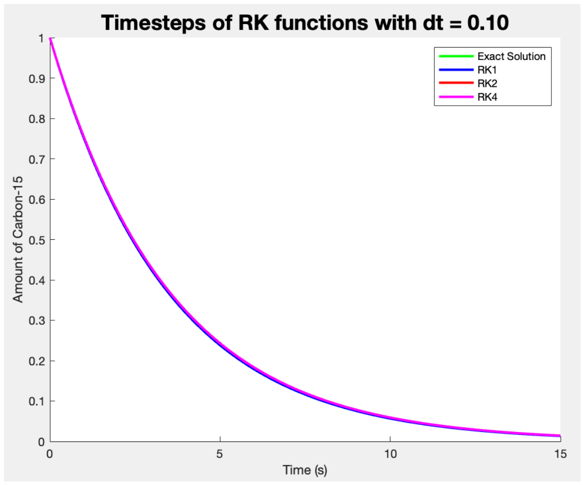


Modified output with $y_0 = 1$ but adding a time step of 5 seconds:

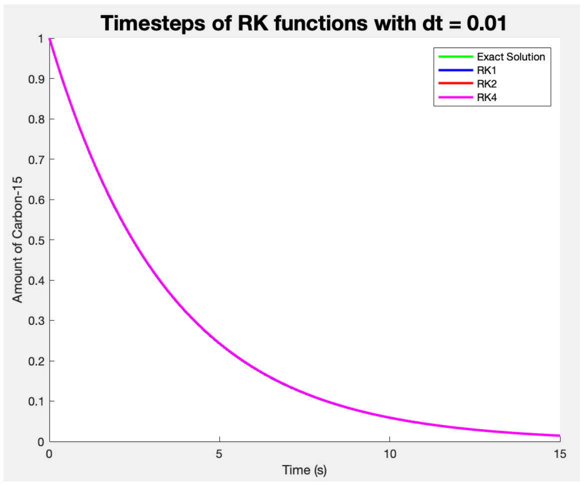
```
Which problem to test?
2
   dt        RK1       RK2       RK4
1.00:   3.11e-02 3.42e-03 1.38e-05
0.10:   3.09e-03 2.95e-05 1.18e-09
0.01:   3.08e-04 2.91e-07 1.16e-13
5.00:   2.14e-01 2.03e-01 1.64e-02
```



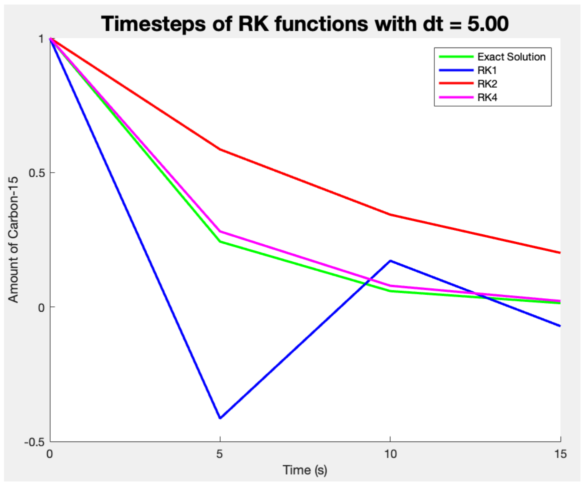
Timesteps of RK functions with dt = 1.00
Exact Solution
RK1
RK2
RK4
Amount of Carbon-15
Time (s)



Timesteps of RK functions with dt = 0.10
Exact Solution
RK1
RK2
RK4
Amount of Carbon-15
Time (s)



Timesteps of RK functions with dt = 0.01
Exact Solution
RK1
RK2
RK4
Amount of Carbon-15
Time (s)

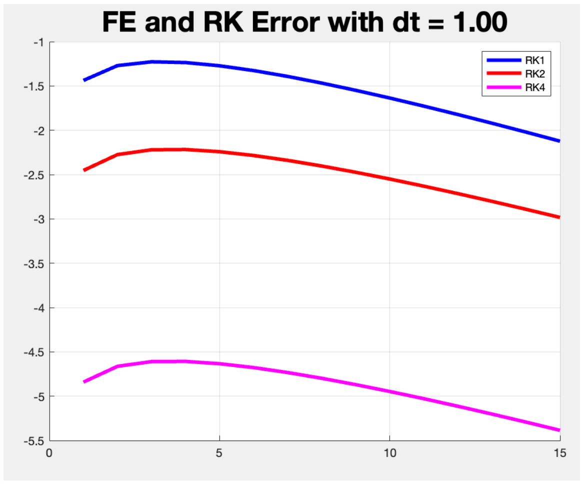


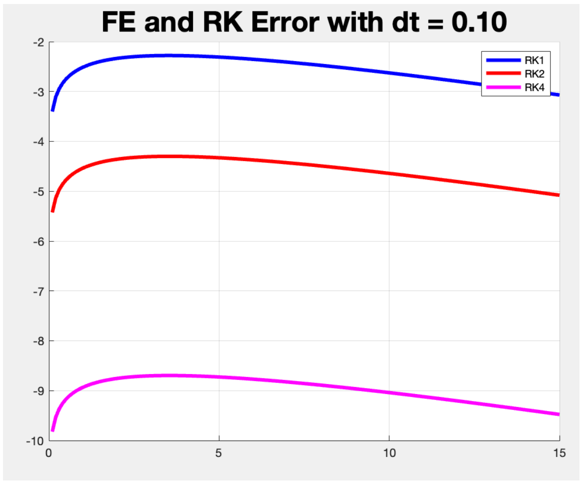
Timesteps of RK functions with dt = 5.00
Exact Solution
RK1
RK2
RK4
Amount of Carbon-15
Time (s)

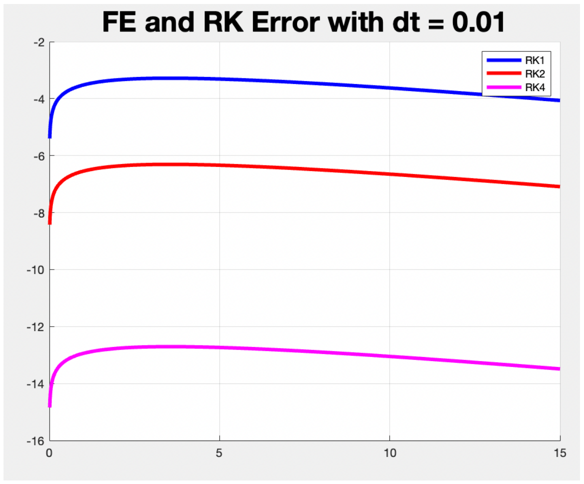
2.5 Discussion and Conclusion

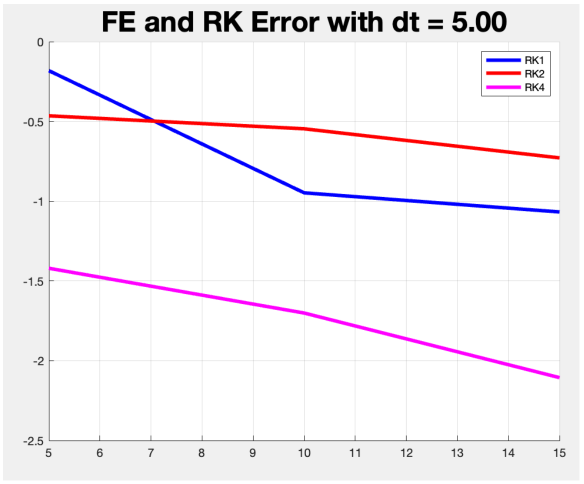
It can be shown that the fourth order Runge-Kutta method is the most accurate at approximating a differential function at any size of time step compared to lower orders. At $t = 15s$, both the exact function and the RK4 method with a time step of 0.01s agree that there is 1.4353546157% of the original Carbon-15 left. With this radioactive decay example, the methods all approximated nearer to the actual value as they continued, regardless of time step or order of approximation.

The relationships between the average error values loosely followed the error approximation of the orders for the default input values as shown below. These graphs have the error values plotted on a logarithmic scale to measure order of magnitude. In the time steps 1, 0.1, and 0.01, the order of magnitude of the error approximately doubles (negatively) when the order of the approximation doubles. With a time step of 5 seconds, the RK1 and RK2 errors are not orders of magnitude smaller. RK4 is, but not by a factor of 4 from RK1. This suggests that for any (if not, most) time steps, the fourth order RK method will produce the closest graph out of any of the three tested methods, and for steps at least as small as 1 second, RK2 will have an error with a smaller order than RK1 by a factor of around 2.









For 5 seconds specifically, RK1's approximations are drastically above and below the actual values, while RK2 and RK4's graphs are smoother and close to the exact solution's graph. At the end, they still appear to be at their closest point to the graph than at any of their other approximation points; nevertheless, their average errors are still within orders of magnitude of each other and do not scale with the orders of the approximations.