Ojas Bardiya
UID: 505145284
CEE/MAE 20
07/25/20

1. Split-and-Average

1.1. Introduction

We find the shape a set of points converges to by repeatedly splitting the set of points to their midpoints for each adjacent pair and calculating the weighted average for the resulting array which we then reassign the points to. We use two functions - **splitPts** and **averagePts** to split the points and find their average respectively.

1.2. Model and Theory

The splitPts function takes an array of points and finds the midpoint of each 2 neighbors. We consider the last and first element to 'wrap around' each other, the array can be considered circular, therefore the last and first point are neighbors.

We use the formula –

$$\text{(i)} \qquad xa(k) = w1 \times xs(k-1) + w2 \times xs(k) + w3 \times xs(k+1)$$

where we have

- xa = array consisting of calculated averages
- xs = split-array
- w1, w2 and w3 are the 3 weight values in the [1*3] array w.
-

1.3. Pseudocode

For splitPts we have –

```
Calculate the length of the input array
Loop through each element of the input array
    Populate new points with the previous ones for every other index
    If the index is the last, assume it cycles to the first again
    Else, Populate remaining cells at the midpoint of every pair of adjacent cells
End
```

For averagePts we have –

```
Check to see if the sum of weights is valid
If so
    Normalize the weight array w
```

```
        Calculate the length of the split array
Loop through each element of the split array
        Determine the right and left neighbors and use formula (i) accordingly
end
```

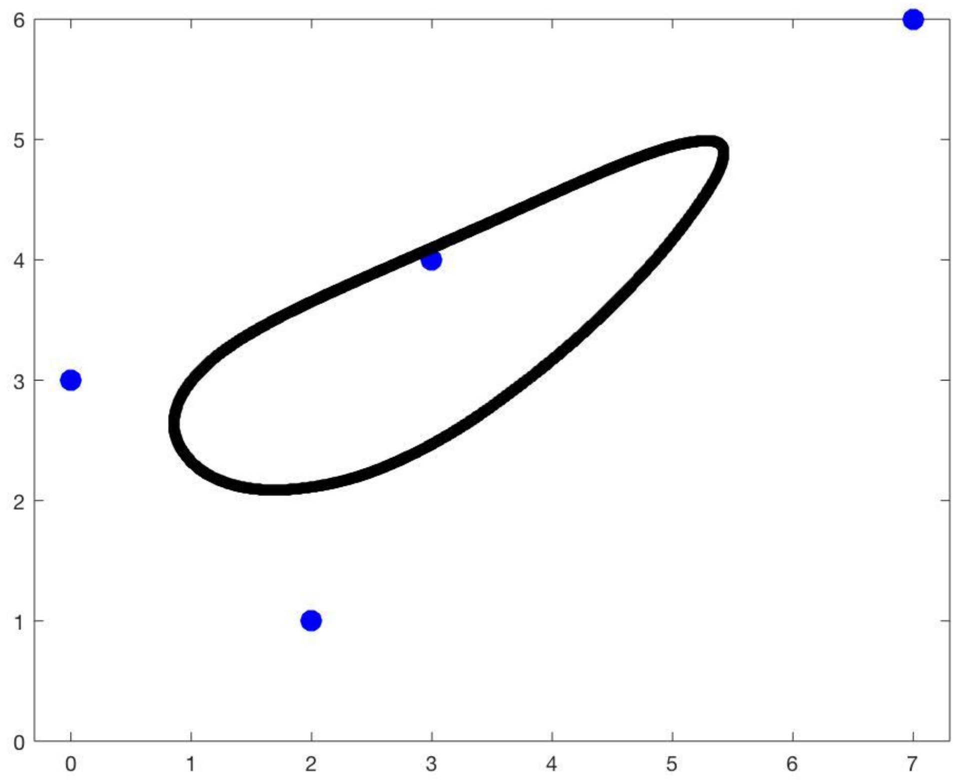
Main code –

```
Set Initial conditions
Set the conditions for the while-loop
Plot the initial figure
Until maximum node displacement is less than given threshold continue to iterate
        Call the split function for each resulting set of points
        Call the average function for each resulting set of points
        Calculate maximum error
        Update the initial set(s) of points
Plot the final values
```

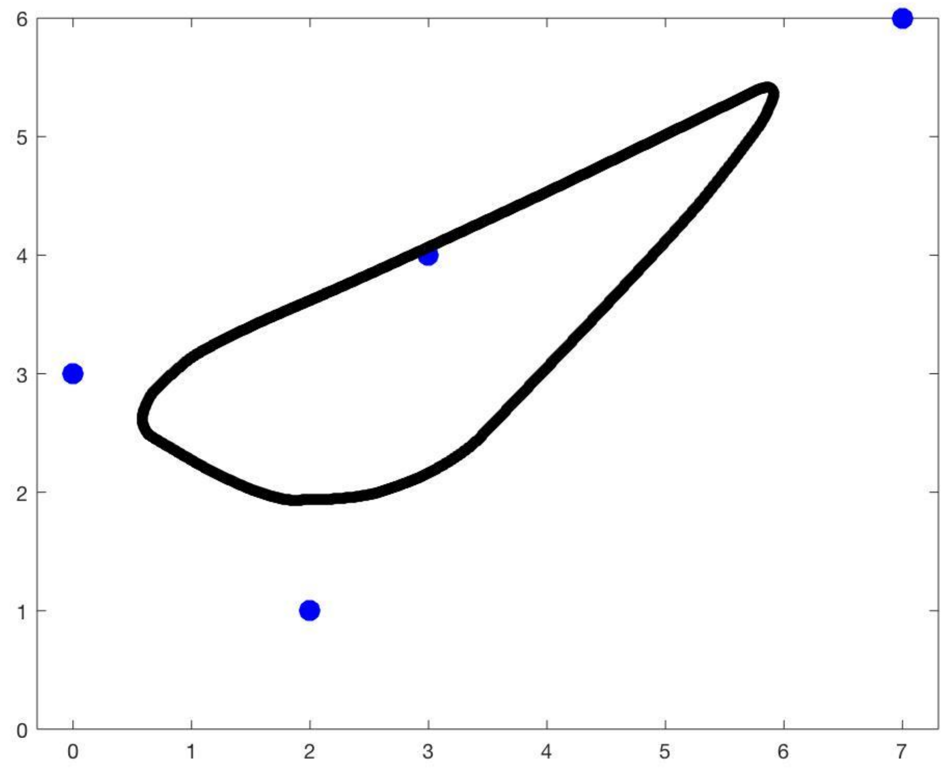
1.4. Calculations and Results

We use the initial sets of points **x = [0 3 7 2]** and **y = [3 4 6 1]**

Using the array **w = [1 2 2]** we observe the figure –

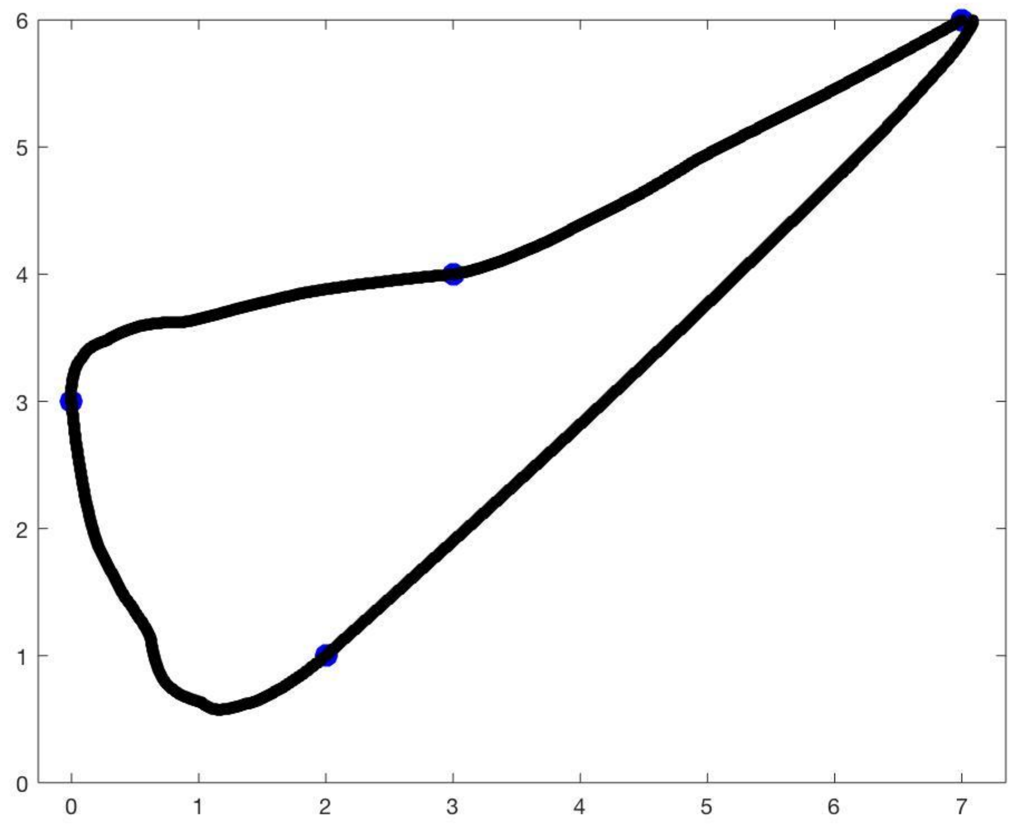


We have for the array **w = [1 2 10]** –



Clearly, we see that there is a greater distribution of points towards the bottom-left corner in the 2nd case, and also indicates a lower degree of smoothness of the curve. However, the curves retain a high degree of similarity, and both converge.

For the array **w = [-1 2 3]** we have –



There is a degree of similarity between this and the first 2 cases, but this shape is more deformed and less smooth than the previous 2 cases. The negative weight value because of a movement away from the top-right corner points. This also takes a larger number of iterations to converge.

We observe for the **array w = [-1 2 -2],** the points do not converge.

Typically, it takes **5-10 iterations** for the points to converge to a shape.

1.5. Conclusion

We see that inputting negative weights may prevent a set of points from converging to a shape and doing so increases the total number of iterations required for convergence.

2. Runge-Kutta method for C-15 Decay

2.1. Introduction

We observe C-15 decay using the Runge-Kutta method to analyze the differential equation that governs radioactive decay. We also compare this analytical solution to the exact solution of the differential equation and determine the difference between the two, i.e., the absolute error. We also plot graphs for three different values for time-step: 1, 0.1 and 0.01 and observe the resulting differences.

2.2. Model and Theory

We use the First, Second and Fourth Runge-Kutta Methods to estimate the amount of C-15 left after a certain period of time. We use the equations

Runge-Kutta 1

$c_1 = \Delta t * f(t_k, y_k)$

$y_{k+1} = y_k + c_1$

Runge-Kutta 2

$c_1 = \Delta t * f(t_k, y_k)$

$c_2 = \Delta t * f(tk + ½ * \Delta t, yk + ½ * c_1)$

$y_{k+1} = y_k + c_2$

Runge-Kutta 4

$c_1 = \Delta t * f(t_k, y_k)$

$c_2 = \Delta t * f(tk + ½ * \Delta t, yk + ½ * c_1)$

$c_3 = \Delta t * f(tk + ½ * \Delta t, yk + ½ * c_2)$

$c_4 = \Delta t * f(tk + ½ * \Delta t, yk + c_3)$

$y_{k+1} = y_k + 1/6 * c_1 + 1/3 * c_2 + 1/3 * c_3 + 1/6 * c_4$

for the differential equation

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

where $t_{1/2}$ denotes the half-life of C-15.

The exact solution is given by –

$y(t) = y_0 * e^{\frac{-log(2)}{t_{1/2}} * t}$

The mean error is calculated using the formula –

Error = abs(mean(value of exact solution – value of particular Runge-Kutta method))

2.3. Pseudocode

advanceRV function –

```
determine which method to use
        Apply the relevant equations for the method chosen
Update the value of y
```

Main code –

```
Set the initial values for start and stop time
Set the value for Carbon-15 half-life
Create an array that stores the relevant values for the time-step
Loop through each of the time-step values
    Initialize and fill the array for time values
    Calculate the length of time vector/array
    Initialize arrays for each of the Runge-Kutta methods as well as one for the exact
    solution
    Set the initial amount of C-15 in each array
    Loop through each time value
        Apply the exact solution
        Apply each Runge-Kutta method
    Calculate the mean error for each Runge-Kutta method
    Plot the final values for each time-step value
end
```

2.4. Calculations and Results

The final result is shown below –

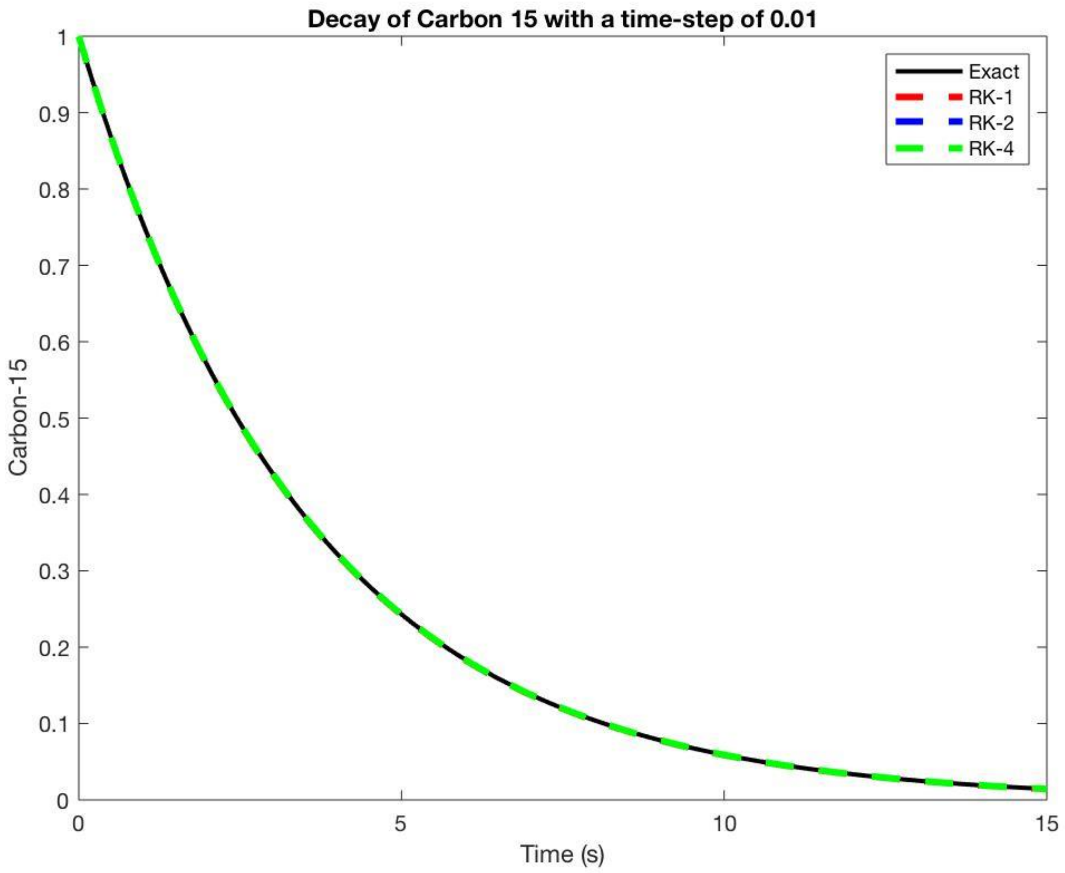
```
  dt        RK1        RK2       RK4
 1.00:  3.11e-02   3.42e-03  1.38e-05
 0.10:  3.09e-03   2.95e-05  1.18e-09
 0.01:  3.08e-04   2.91e-07  1.16e-13
```

This is for a half-life value of $t_{1/2}$ = 2.45 and initial amount of y = 1.
The final time is given by t_f = 15.

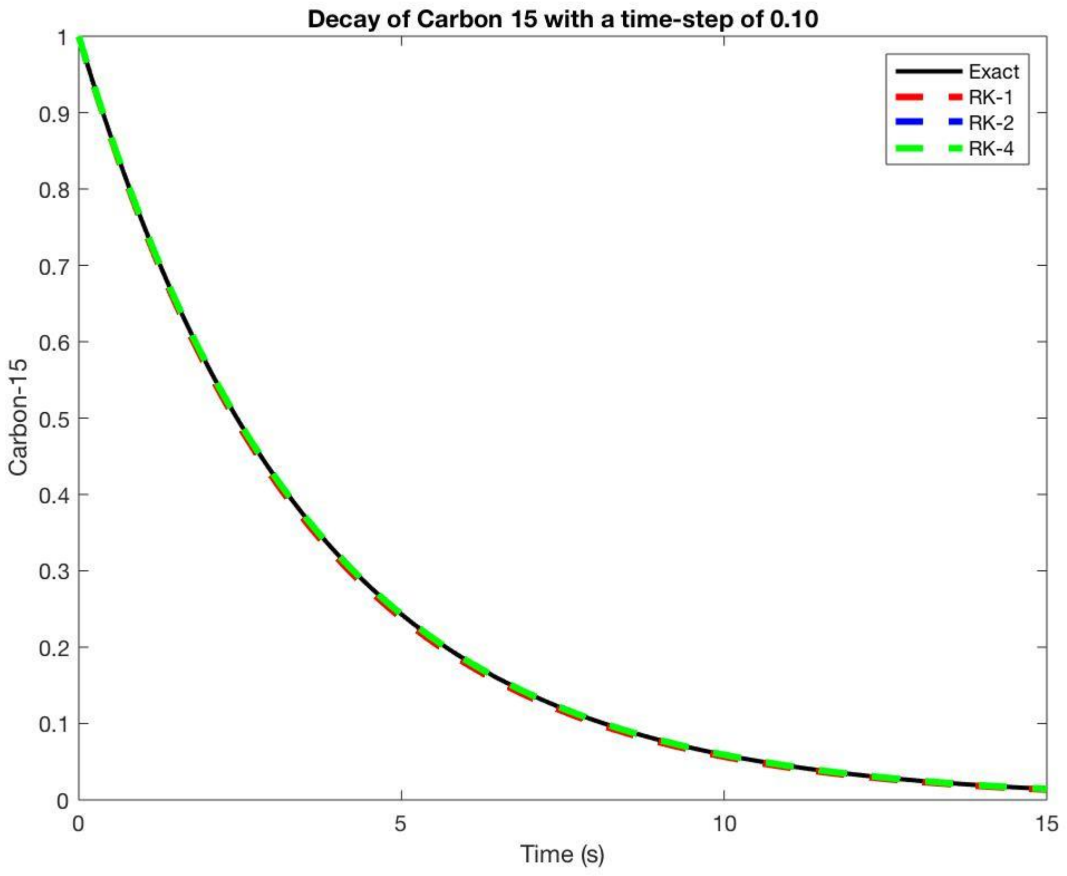
From this table, we see that error values scale at about a rate 10^-1 for Runge-Kutta 1, 10^-2 for Runge-Kutta 2 and 10^-4 for Runge-Kutta 4 for a change in dt by a factor of 10.
Thus, we can infer the time complexities of the error scales are O(dt), O(dt^2) and O(dt^4).

We also have the plots –

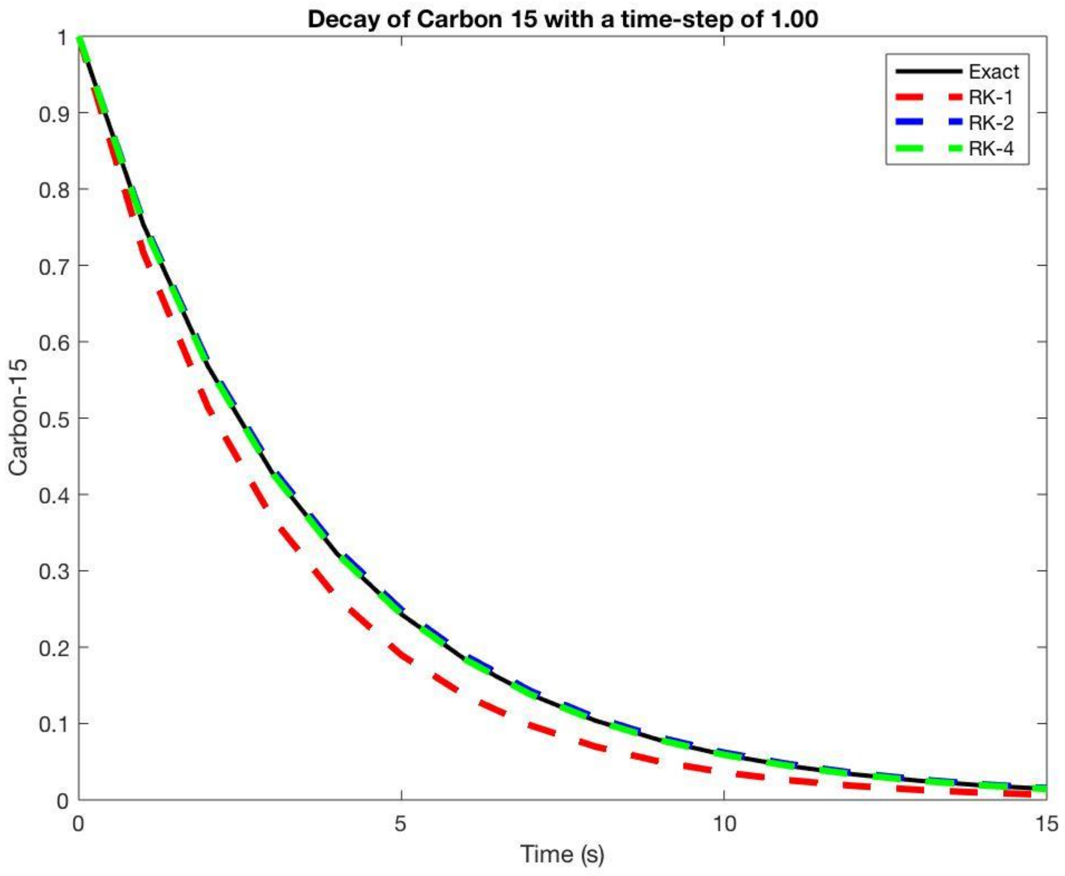
For Runge-Kutta 4 –



For Runge-Kutta 2 –



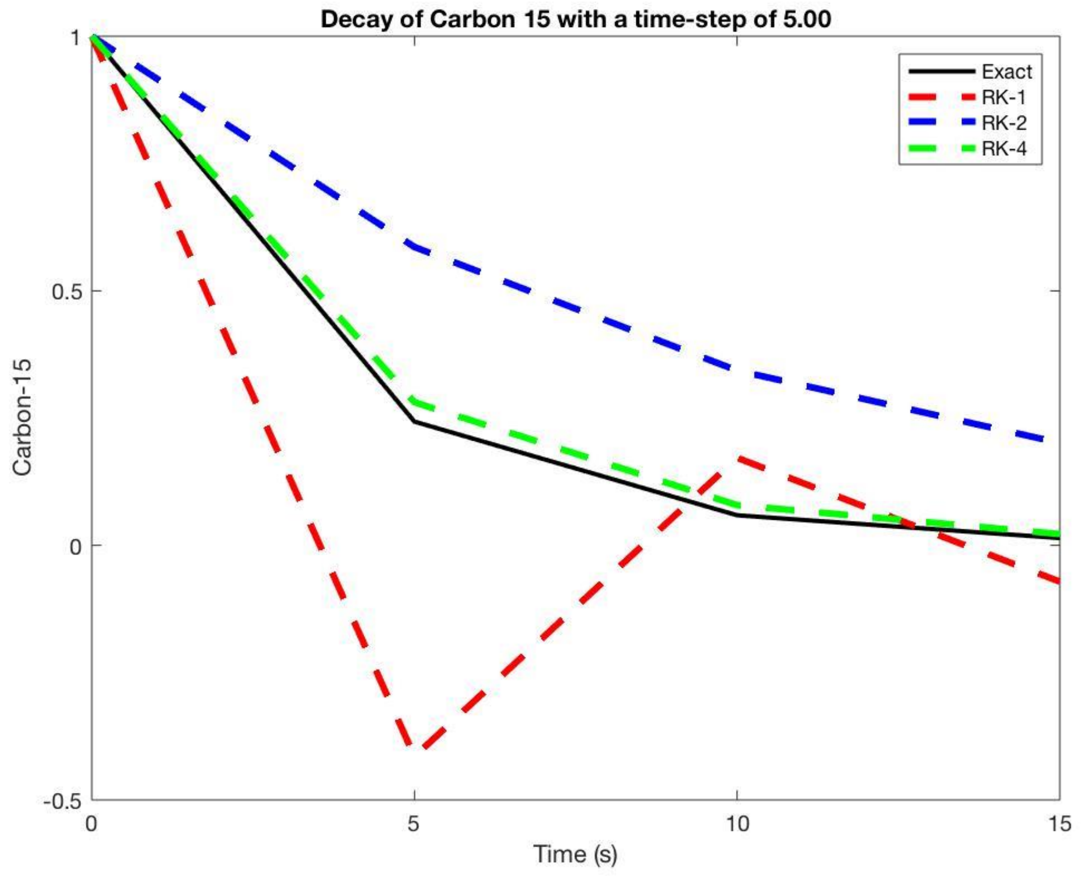
For Runge-Kutta 1 –



Clearly, we see that the error is most in the case of Runge-Kutta 1 and least in case of Runge-Kutta 4 from both the plots and final table of results. All 3 methods nearly coincide for time-steps of dt = 0.1 and dt = 0.01 but there is a noticeable divergence for the time-step of dt = 1.

Taking the most accurate numerical method (RK-4) and a time-step = dt = 0.01 we see that there is ~0.01435 of C-15 left.

For dt = 5, we observe



There is a much greater degree of error in this case, and far more so for Runge-Kutta 1 and Runge-Kutta 2 than Runge-Kutta 4 in this case, as their errors scale faster.

Runge-Kutta 1 essentially behaves like different pieces of a linear function for larger and larger values of dt. In this case, it first decreases then increases rapidly due to underestimation and overestimation respectively, which is a consequence of a greater degree of error.

2.5. Conclusion

We see that the order of accuracy for the given methods is given by
RK-4 > RK-2 > RK-1.