MATLAB Problem 4

A quick note before you start – you will answer all questions for this problem within this Word document. When finished, please save the document as a .pdf, which you will upload directly to Gradescope. **For this and future MATLAB problems, we will also ask you to upload your code to BruinLearn.** This can be done through the same BruinLearn "assignment" from which you downloaded the problem set files. You should turn in all of the .m files that you use in this problem.

This week, you'll be implementing a Runge-Kutta Solver and comparing it to the Euler Solver you built in MATLAB Problem 2.

4a: Implement RK4 Solver

You now have the formula for the fourth-order Runge-Kutta method, and you have a solver framework from the Euler Solver code you built in MATLAB Problem 2. These are the only parts you need to implement your very own Runge-Kutta solver. Implement this solver, and call your function RK4solver.m. **Copy the whole RKsolver.m function code, and paste it as text below.**

```
function [tout, yout] = RK4solver(ODEfunIn, solveTime, y0, stepSize)
% This is an implementation of the fourth-order Runge_Kutta method. Inputs are as follows:
% - ODEfunIn is a function handle to a differential equation
% - solveTime is the range of interest of the independent variable
% - y0 is the initial condition
% - stepSize is step size for the solver, in the same units as solveTime
% Define the values of t at which we will evaluate our function
tVals = solveTime(1):stepSize:solveTime(2);
% Allocate an array
yout = zeros(size(tVals));
% Start us off at our initial conditions
yout(1) = y0;
% Amount of steps taken
nSteps = width(yout)-1;

for stepNum = 1:nSteps

   % Grab the last value of y and current value of t.
   y_n = yout(stepNum);
   t_n = tVals(stepNum);

   % Calculate the dydt's using the ODE function passed in
   kn1=ODEfunIn(t_n, y_n);
   kn2=ODEfunIn(t_n+1/2*stepSize, y_n+1/2*stepSize*kn1);
   kn3=ODEfunIn(t_n+1/2*stepSize, y_n+1/2*stepSize*kn2);
   kn4=ODEfunIn(t_n+stepSize, y_n+stepSize*kn3);

   % Calculate y_nplus1 using the Runge-Kutta method, and store it in the output array
   y_nplus1 = y_n+stepSize*(kn1+2*kn2+2*kn3+kn4)/6;
   yout(stepNum+1) = y_nplus1;
end
% Rename this to be more intuitive as output
tout = tVals;
```

Hint: For the love of all that is good, please just copy eulerSolver.m, rename it, and change the algorithm parts to make it an RK4. I do not recommend reinventing the wheel here.

4b: Compare RK4 to Euler Method

Let's compare the performance of your new RK4 solver with that of your Euler solver from MATLAB Problem 2, by solving the same differential equation we tackled in that problem. As a reminder, the equation is:

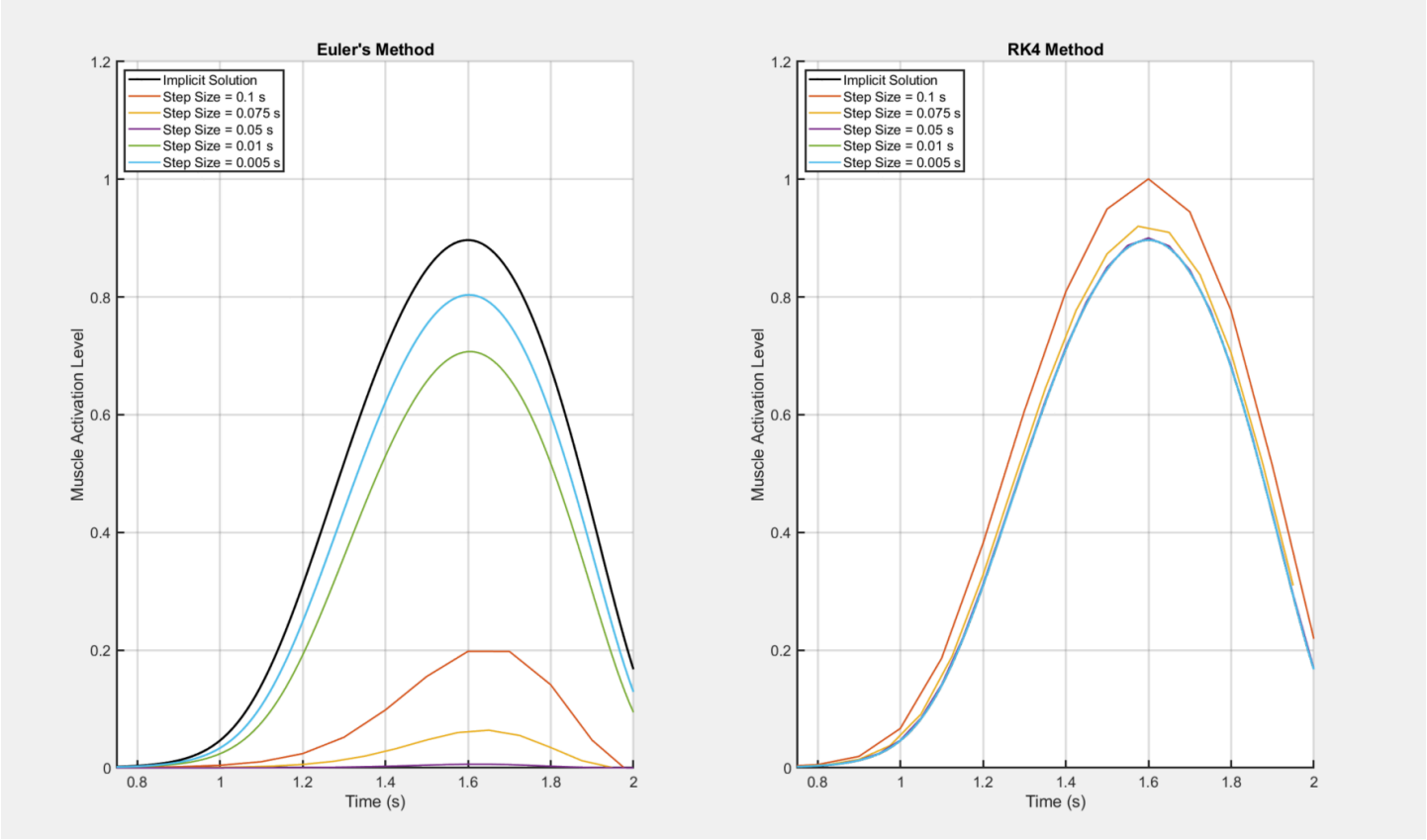
$$\frac{da}{dt} = \frac{u(t) - a(t)}{t_{act}(0.5 + 1.5a(t))}$$

$$u(t) = 1.7a(t) * \sin sin\left(\frac{\pi}{2}t\right), \quad a(0) = 0.01$$

Write a script in MATLAB to compare performance of your RK solver with that of your Euler solver at the following step sizes: 0.1 s, 0.075 s, 0.05 s, 0.01 s, and 0.005 s. The following instructions will get you pointed in the right direction:

1. Pull in the code from MATLAB Problem 2b to calculate the implicit solution to this equation – this should be a simple copy-paste job. This will serve as our ground truth.
2. Create a new figure, with two subplots (side-by-side). Turn hold on in each subplot.
3. In each subplot, plot the implicit solution in black.
4. Iterate through the step sizes, calculating an Euler solution and an RK4 solution for each. Within your loop, plot the Euler solutions in the left subplot and the RK4 solutions in the right subplot. Again, this should be a copy-paste job with minor modifications to account for the second solver and subplot.
5. Switch to the left subplot, turn hold off, add our formatting, add a legend (look at the code for MATLAB Problem 2d to see how to set up the dynamic legend), and set the y-axis limits to [0 1.2]. **Hint:** type "help ylim" in the command window to learn more about this.
6. Repeat step 5 for the right subplot.

Make your figure large enough so the legend doesn't interfere with the plots, save it as a .png, and insert it here.



Describe what you see in each plot. What does this tell us about the RK4 method vs. the Euler method?

For the Euler Method plot, the graph grows close to the implicit solution with step sizes of 0.01s and 0.005s. For the RK4 method plot, as the step size decreases, the graph grows closer and closer to the implicit solution. The step size does not need to be too small in order to get precise results using the RK4 method. The RK4 method gets us more accurate results at much larger step sizes compared to Euler's method.

Believe it or not, there are still cases when we might choose to use the Euler method rather than RK4. Can you think of a potential situation where Euler might be a better choice?

Euler might be a better choice where computational speed is valued more than precision. In a situation where you don't want to use too much computational power and do not care about precision, it would be better to use the Euler method rather than RK4. The Euler method has less calculations per step than RK4.