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 CCLE.** This can be done through the same CCLE "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)


tVals = solveTime(1):stepSize:solveTime(2); % ** Fill in code here to generate a vector of the relevant

% t points, using your range of interest and step size. % Allocate an array

yout = zeros(size(tVals));

% Start us off at our initial conditions

yout(1) = y0;

% How many steps will I be taking?
nSteps = length(tVals)-1; % ** Fill in code here to determine how many steps you'll take.

% Keep in mind that we are already starting at y_0.**

stepNum = 1:nSteps

% Grab the last value of y and current value of t (we'll need this).

y_n = yout(stepNum); t_n = tVals(stepNum);

% Calculate dydt using the ODE function you passed in dydt = ODEfunIn(t_n,y_n);

kn1 = ODEfunIn(t_n,y_n);
kn2 = ODEfunIn(t_n + stepSize/2, y_n + kn1*stepSize/2);
kn3 = ODEfunIn(t_n + stepSize/2, y_n + kn2*stepSize/2);
kn4 = ODEfunIn(t_n + stepSize,y_n + stepSize*kn3);
% Calculate y_nplus1 using euler's method, and store it in your output
% array
y_nplus1 = y_n+stepSize*(kn1+2*kn2+2*kn3+kn4)/6; %** Fill this in according to RK4's formula.**
yout(stepNum+1) = y_nplus1;

end

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 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:

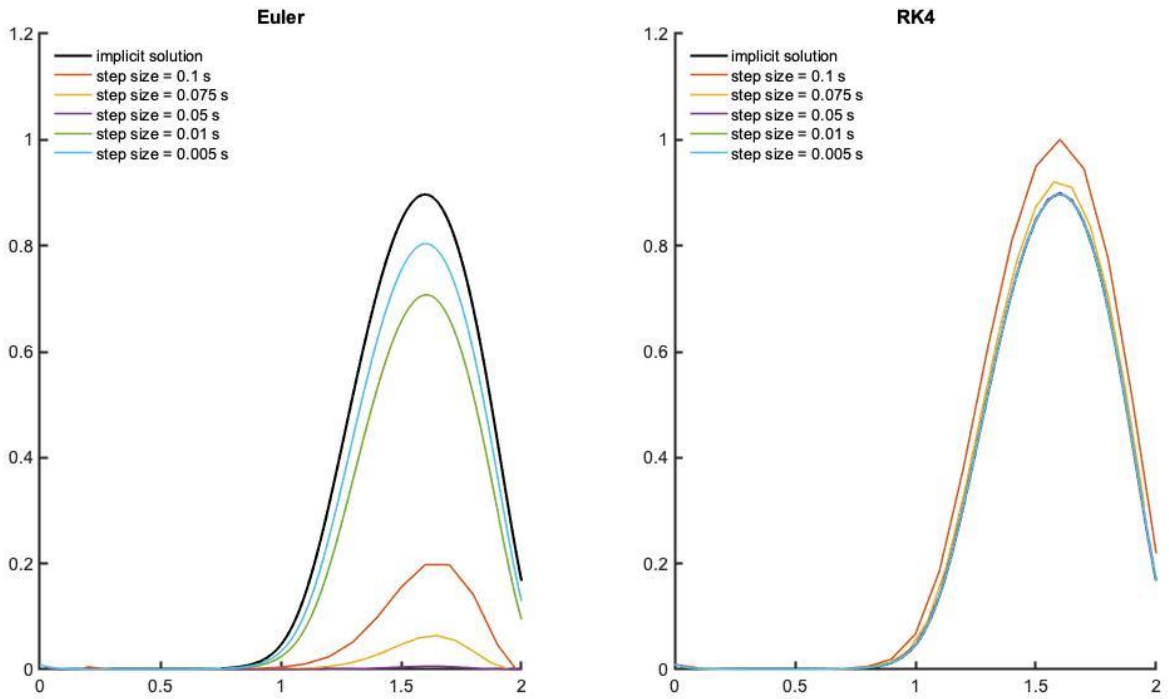
$$\frac{da}{dt} = \frac{u(t) - a(t)}{t_{act}\left(0.5 + 1.5a(t)\right)}$$

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

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?

It is very clear how much more accurate RK4 is than Euler. Every timestep of RK4 resembles the implicit solution much better than the Euler does. Only the smallest step size of 0.005s is remotely close to the implicit in the Euler plot. Having a larger step size like 0.1 causes a severe underestimation in the Euler method but a slight overestimation in the RK4 method. The RK4 is also so accurate that it is hard to distinguish other timesteps in the plot since they are basically plotted on top of each other.

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?

As mentioned before, scenarios where accuracy is not as important as compared to speed favors Euler over RK4. Although super accurate, RK4 will need more time for computation compared to Euler. Cases like running multiple simulations to see if for example, the red ball will always be the first one to drop, only care about the overall picture not every detail like how fast, the order of balls that dropped.