

# Project 1, Spring Mass Damper

10/17/25 - Spring Mass Damper to learn 6DOF simulations

Following this tutorial by Fast Lab Tutorials:

<https://fastlabtutorials.blogspot.com/2015/04/everything-you-need-to-know-to-code.html?m=1>

**Goals:** To get started in simulations. This is a simple set-up and appears to be a comprehensive tutorial - hoping that this is a good jumping-off point that will provide the building blocks to move on to more complicated simulations, specifically for flight simulations and ultimately spaceflight/rocket/spacecraft simulations.

## Setting Up the Environment

Starting on the first steps of the program. I struggled in VSCode to get it to connect to Anaconda python rather than the default(?) python3, specifically so that I could import packages like numpy which I have installed with Anaconda, but apparently not for the other versions of python on my computer.

What ended up working was opening up my Anaconda dashboard, and opening VSCode *directly* from there. This seemed to force the version of Python being used to be the Anaconda one, and importing the needed libraries was successful. Will update if a more intuitive solution comes up? Though I even tried pip installing numpy into the version VSCode was originally in, and it said that numpy was already installed. Unsure why VSCode couldn't access it.

Next steps: Actually starting the programming now that the environment is set up.

Questions I have: Is this considered a 2 degree of freedom simulation? The listed states in the tutorial are the position of the spring and the velocity, and states that arrays are used for "higher degree" sims.

Answers: Yes, two degrees. Position and velocity. The size of the initial array is the number of states that will be occurring in the simulation. Still not sure of the process for a full 6DOF simulation - the tutorial states that this is the entire set up for the basics, that a 6DOF simulation is simply built off of this.

## Coding the Simulation

First, I created a numpy array of size 2 to accommodate the two states (degrees) used in this simulation. As stated before, those states are the position of the mass, and the velocity at a given time. The next step is to create a function that takes the derivatives of the states, specifically using the mathematical model for a spring mass damper given in the tutorial,

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -k/m & -c/m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1/m \end{bmatrix} u$$

Where  $x_1$ ,  $x_2$  are positions,  $\dot{x}_1$ ,  $\dot{x}_2$  are the velocities (derivatives of x). k is the spring constant, c is the damping coefficient, m is the mass, u is the control input (?).

These variables are defined with the derivative function and given arbitrary values to start. The derivative function returns the derivative of the position array - the velocity array. I then set time variables, an initial and final time as well as a step value. A for loop over the length of the time then creates a state\_out variable, which is the final array of our position (0th row) and velocity (1st row) - the loop adds these values for each time step into subsequent columns (the state\_out array was initialized as an array of zeros with the length of the number of time steps.)

The state array then increments by adding the current derivatives times the time step. All of this is modeled off of Euler's method:

$$x_{n+1} = x_n + \dot{x}_n \Delta t$$

We then plot separate plots for position vs. time and velocity vs. time.

Ways to improve it: Use RK4 as a better integration method than Euler's method!

Adding in the control factor: modeling as a simple PD (proportional derivative) controller. This type of controller tracks a given reference point, and adjusts the forces needed to get it to that point based off of the difference between the actual position and the desired position - changing proportional to the changes in the position and velocity as the simulation goes on.

### Replacing Euler's Method with RK4 Integration

What is RK4 integration? RK4 stands for fourth-order Runge-Kutta integration - it is another numerical method similar to Euler's method used for approximating solutions to differential equations. It is considered to be more accurate than Euler's method, though Euler's method is easier to implement.

Like Euler's method, it calculates a series of proceeding values based off of the current one and a given step size - in this case, the step size is the value given for the time increments in the simulation. It differs from Euler's method in that it calculates *four* different slopes to complete each approximation rather than simply one (which is why it is generally more accurate). The slopes,  $k_1$ ,  $k_2$ ,  $k_3$ , and  $k_4$  are used in the final approximation:

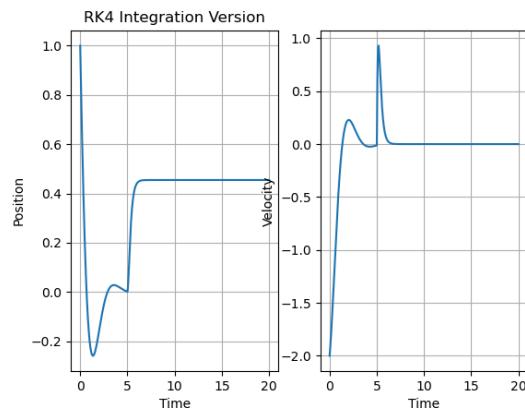
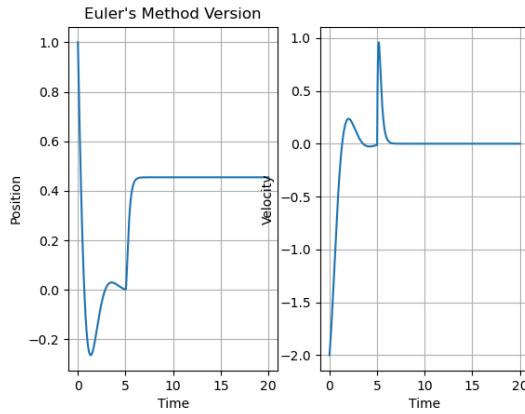
$$x_{n+1} = x_n + \frac{1}{6} (k_1 + 2k_2 + 2k_3 + k_4)$$

To implement this into the code, I used the following article by Souvik Ta as a starting point:

<https://medium.com/@souvikat/numerical-solutions-of-odes-with-python-euler-runge-kutta-and-beyond-421d4926d1fd>

My goal, though, was to be able to implement RK4 into my existing program without having to entirely rewrite it, so I did some extra research into how that would look. I also had to figure out how to implement my PD control factor, which was not considered in the above article. I found formulas for each RK4 coefficient, and found that I was able to use my already written derivatives() and control() functions to calculate each one for each time step, and save it into the same array as I did with Euler's method.

The final plot of the simulation ended up looking effectively identical, which I am unsure if that is a sign that something is faulty, or if that is expected for the conditions I am using to test both. I intend on varying the initial conditions and the control factor, and comparing to the graph for the analytical solutions as well, as is done in the above article. Below are the plots for both versions, with all variables exactly the same.



## Sources

<https://www.computrols.com/pid-control-explained/>

<https://medium.com/@souvikat/numerical-solutions-of-odes-with-python-euler-runge-kutta-and-beyond-421d4926d1fd>

Fastlabtutorials link above