Fall 2020 ME751 Final Project Report
University of Wisconsin-Madison

SimEngine3D Development

Logan Rapp

December 12, 2020

Abstract

This report describes the progress on the development of a Python code - SimEngine 3D – that is used to simulate multibody dynamic problems.

Link to github repo:
https://github.com/lrapp/simEngine3D

Contents

1. General information

- Home Department: Mechanical Engineering – Solar Energy Lab
- Current Status: PhD student
- Individuals working on the Final Project:
    - Logan Rapp
- I am not interested in releasing my code as open source code.

2. Problem statement

I continued development of my simEngine3D code to include support for all joints and driving constraints discussed in class and an improved method of providing model definition. My research is not closely aligned with topics covered in this course, so I was not able tie this final project with my research.

3. Solution description

3.1 System object & body/constraint information

I used a custom python class definition called "sys" to hold all the components of the problem and the results. This includes a list of the bodies in the simulation, a list of the constraints, and other important items. A screen shot of the "sys" class definition is shown in Figure 1.

```
22  #custom class definition to hold all components of a simulation
23  class sys():
24      def __init__(self):
25          self.nb=0 #number of bodies
26          self.bodies=[] #list of body objects
27          self.constraints=[] #list of constraints
28          self.time=0 #current timestep
29          self.h = 0 #time_step
30          self.nc=0 #number of constraints
31          self.outputs = np.nan #holds results information
32          self.n = 0 #index
```

Figure 1 - Definition of custom class to hold information related to the simulation

I've created a .txt file input system in order to facilitate inputting the body and constraint information. The file parser function I use is called "data_file" and is located in the Python script "simEngine3D_dataload.py". I first define a custom body class, as shown in Figure 2.

```
#%% Define new class to hold body data
class body():
    """Object that holds information about bodies"""
    def __init__(self):
        self.name = ""
        self.ID = ""
        self.mass = np.float()
        self.q = np.array([0,0,0,0,0,0,0])
        self.a_bar= np.array([0,0,0],dtype=np.float64)
        self.r_dot= np.array([0,0,0],dtype=np.float64)
        self.r_ddot= np.zeros([3,1])
        self.s_bar = np.array([0,0,0],dtype=np.float64)
        self.normalized= bool()
        self.p_dot= np.zeros([4,1])
        self.p_ddot= np.zeros([4,1])
        self.ground=""
```

Figure 2 - Definition of custom class to hold body information

The “data_file” function requires a filepath to be passed to it, and the .txt file must follow the format shown in Figure 3.

```
#Enter body information
can enter either normalized or not normalized p values, if not normalized you must mark the 'Need normalized' flag 'True'
must enter normalized p_dot values
bodies:
[
        {
        'name': 'body i'
        'ID': 1
        'mass': 78
        'type': bar
        'dimensions' : [2,0.05,0.05]
        'q(x,y,z,e0,e1,e2,e3)': [0,1.41421356,-1.41421356,0.653281459,0.270598060,0.653281506,0.27059804]
        'p_dot' : [0,0,0,0]
        'Need normalized': False
        'ground' : False
        'gravity' : [0,0,-9.81]
        }
        ,
        {
        'name': 'body j'
        'ID': 2
        'mass': 0
        'q(x,y,z,e0,e1,e2,e3)': [0,0,0,1,0,0,0]
        'p_dot' : [0,0,0,0]
        'Need normalized': False
        'ground' : True

        }
]


#Enter constraints: CD and/or DP1 -case sensitive, must be seperated by comma, order does not matter
constraints:
[
        {
        'name': 'X CD'
        'ID': 1
        'type': 'CD'
        's_bar_i': [-2,0,0]
        's_bar_j': [0,0,0]
        'c':[1,0,0]
        'f': 0
        'between': 'body i and body j'
        }
        {
        'name': 'Y CD'
        'ID': 2
        'type': 'CD'
        's_bar_i': [-2,0,0]
        's_bar_j': [0,0,0]
        'c':[0,1,0]
        'f':0
        'between': 'body i and body j'
        }
```

Figure 3 - Example of input file

The function reads in all the information and then finds the “bodies” and the “constraints” headings. For the bodies, it counts the number of “{“ to determine how many bodies are in the system. I then create a list of the body object as described in Figure 2. Looping through the bodies data, I use the “{“ and “}” to separate the information for the different bodies. For each body, I create a list of dictionaries that hold all of the body information listed in the input file such as “name”, “ID”, “mass”, etc. Then looping through the list of dictionaries I assign the correct information to the variable in the body object. The function returns a list of the body objects with length equal to the number of bodies in the system.

The constraints are read into Python in a similar manner as described for the bodies above. The function that reads in the constraint information is titled “constraints_in” and is also part of the “simEngine3D_dataload.py” Python script. The “constraints_in” file is passed a path to the input file and returns a list of constraint objects. The constraint object contains all information regarding the constraint and the class definition is shown in Figure 4.

```
#%%
class constraint():
    """Object that holds information about constraints"""
    def __init__(self):
        self.name = ""
        self.ID = ""
        self.type = ""
        self.a_bar_i= np.array([0,0,0],dtype=np.float64)
        self.a_bar_j= np.array([0,0,0],dtype=np.float64)
        self.s_bar_i = np.array([0,0,0],dtype=np.float64)
        self.s_bar_j = np.array([0,0,0],dtype=np.float64)
        self.f=float()
        self.body_i_name=""
        self.body_j_name=""
```

Figure 4 - Definition of the constraint custom class

The other Python script needed to run "smEngine3D.py" is titled "simEngine3D_functions.py". This file holds all of the auxiliary function definitions required to perform the analysis including construction of matrices such as A, G, E, etc., as well as the BDF implementation used for dynamic analysis. I decided to keep these functions separate from the main "simEngine3D.py" file for some added clarity when reviewing the code.

3.2 Using the simEngine3D

To begin a simulation, I first create the "SYS" object. I then pass that object, along with the start time, end time, time step, and path to the input file to one of the two analysis tools implemented thus far. Those two options are inverse dynamics analysis and dynamic analysis. For example, Figure 5 shows the first few lines of the inverse dynamics function.

```
def inverse_dynamics_analysis(SYS,t_start,t_step,t_end,file):
    SYS.h=t_step
    num_steps=int((t_end-t_start) / t_step)
    time_list=np.arange(t_start,t_end,t_step)

    SYS.bodies=data_file(file) #list of body objects
    SYS.constraints=constraints_in(file) #list of constraints

    SYS.nb=body_count(SYS.bodies)
    SYS.nc=len(SYS.constraints)
```

Figure 5 - First few lines of inverse dynamics function

You can see that first I assign SYS.h to equal the inputted t_step, then I calculate the number of steps and create an array of time for this solution. Next I use the functions "data_file" and "constraints" to read in the bodies and constraints as described in section 3.1. I then proceed with the inverse dynamics analysis steps outlined during lecture. This includes solving for the accelerations using only the set of constraints, computing the LaGrange multipliers using the Newton-Euler form of the equations of motion, and finally computing the reaction forces/torques of the prescribed motion. All of this can be found in the "inverse_dynamics_analysis" function included in the "simEngine3D.py" Python script.

The "dynamic_analysis" function looks very similar to the "inverse_dynamics_analysis" for the first several lines but utilizes the BDF method to solve the differential algebraic equations. I have made use of a Pandas DataFrame structure to hold portions of the solution - I'm not sure if this was a good idea or not. It makes it easy to store and retrieve data based on column names and indexing, but just dealing with arrays

might be a faster/more efficient method. One advantage of the DataFrame structure is the ease of saving files and accessing that data later. For large datasets, I like to use the pickle format which is already built into Pandas. For large simulations, this would be useful for reading in simulation results later to do analysis/plotting.

4. Overview of results. Demonstration of your project

Here I've included some results from using my "simEngine3D" code for a couple of homework problems. First, I re-did HW7 P1 using the new code. It can be found in my git repo under Homework/Fixes. When run, it will output a plot of reaction torque which was calculated using the inverse dynamics function of my "simEngine3D" code. Figure 6 shows the plot of reaction torque.

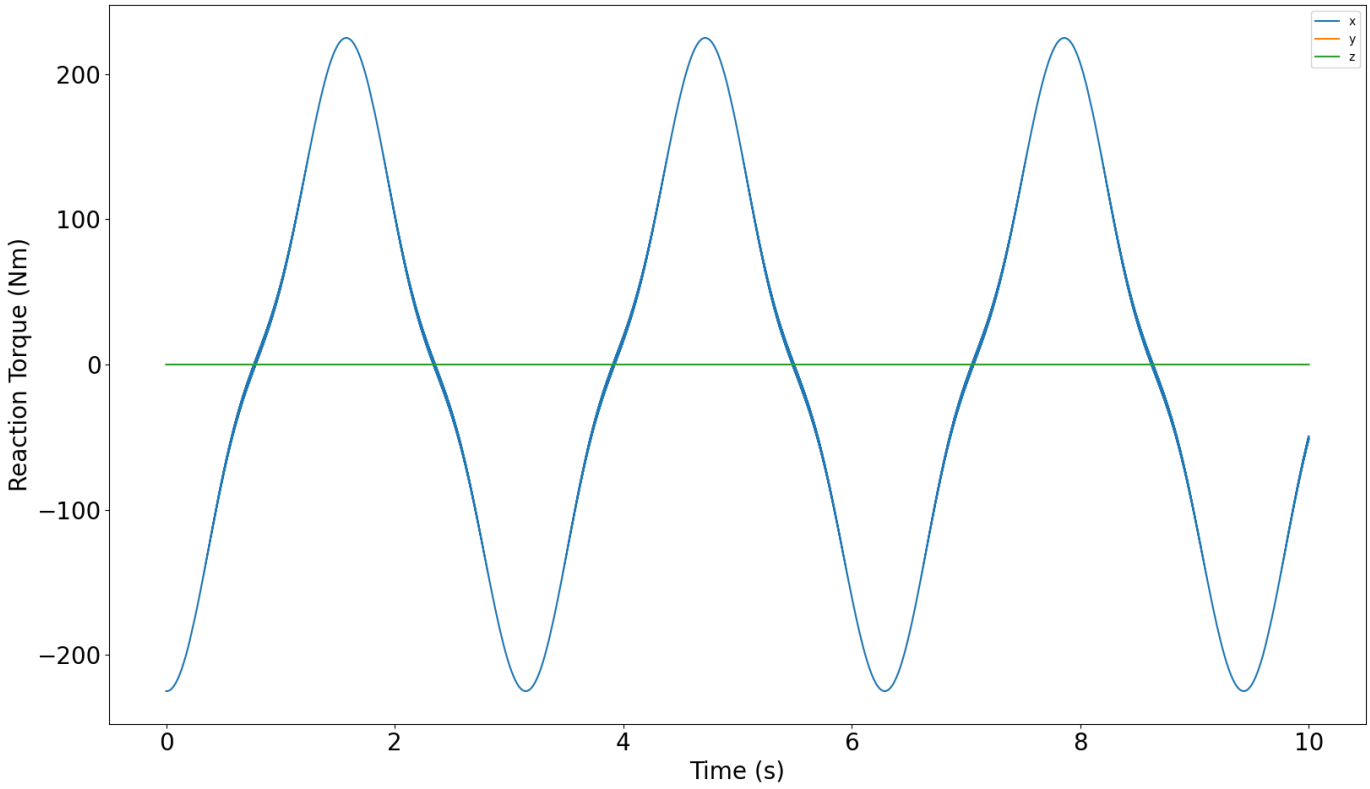

Figure 6 - Reaction Torque result from HW7 P1.

This result was consistent with the reference solution provided for HW7, thus I have high confidence in the accuracy. My simulation for this problem took ~113 seconds which was more than double the ~49 seconds the reference code took to solve. I have not had an opportunity to optimize any of my code and there is likely room for significant improvements.

Next, I used HW8 P2, the double pendulum, to test the dynamic analysis portion of my code. An image of the double pendulum is shown in Figure 7.

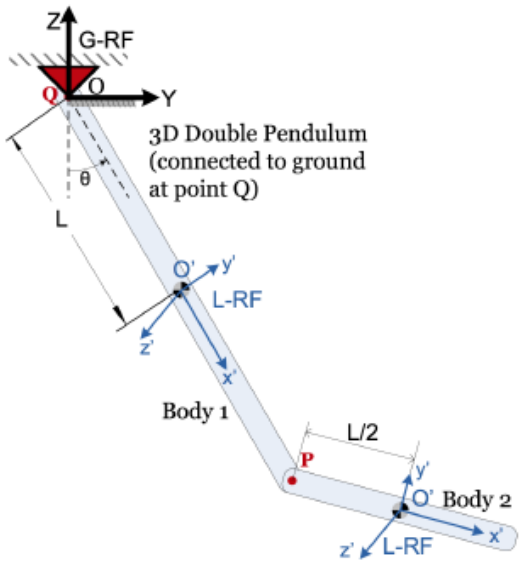

Figure 7 – Double Pendulum diagram from HW8 P2

The code used to produce the following results can be found it my git repo at Homework\Fixes\HW8_P2.py. The simulation is set up for t=0:10 seconds with a step size of 0.001 seconds. The position, velocity, and acceleration vs time plots are shown in Figure 8.

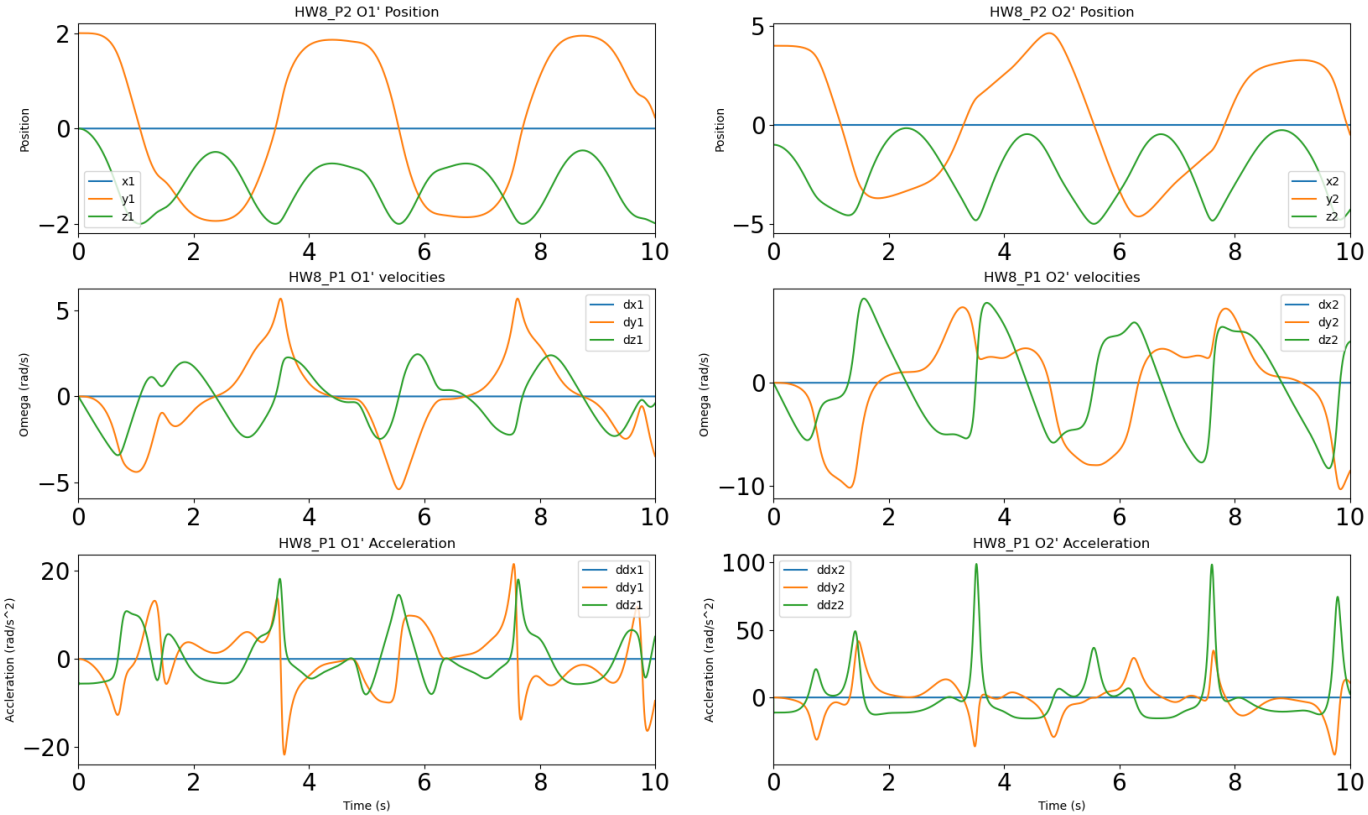

Figure 8 - Position, Velocity, Acceleration vs time plots for HW8_P2

I also created an “animation” of the position of the double pendulum which should display when HW8_P2.py is run. A screen shot from the end of this “animation” is shown in Figure 9.

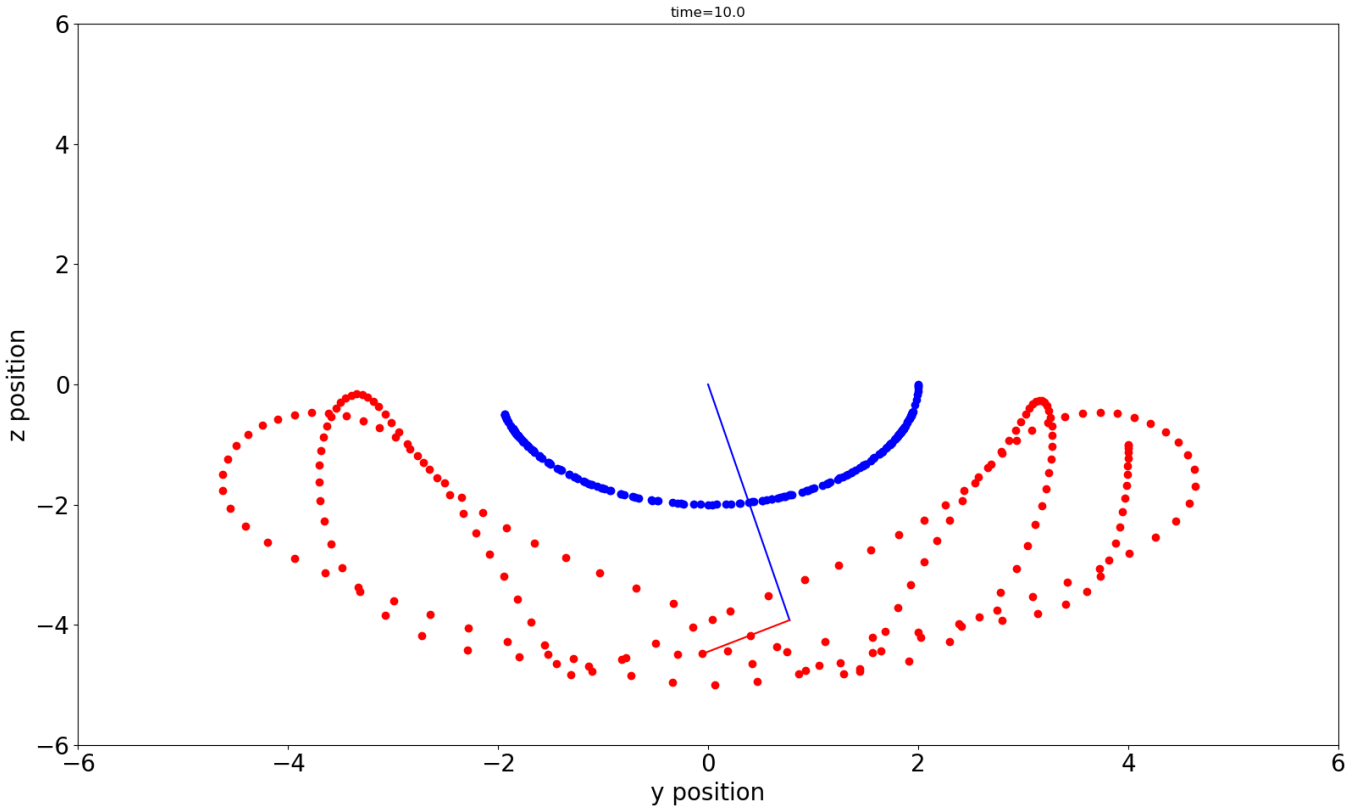

Figure 9 - Screenshot of animation of double pendulum. Dots show traces of O'1 and O'2.

These results seem reasonable, and the simulation took ~266 seconds to solve on my machine. I have not been able to optimize this code, so I could likely decrease the solution time with some more careful examination of the code.

5. Deliverables:

This report will be uploaded to canvas and will also be available in my git repo. My git repo contains the functions needed to run the example problems which are found in Homework/Fixes. The Python code was developed using Spyder 4.1.5. I foolishly upgraded my Anaconda distribution at the beginning of December and had to spend hours fixing my environments so my existing code would work. Everything seems to be working with Python 3.7.9. I would suggest running my code in an editor like Spyder, but the example problems can be run from the command line as well. An example of running from the command line is shown in Figure 10.

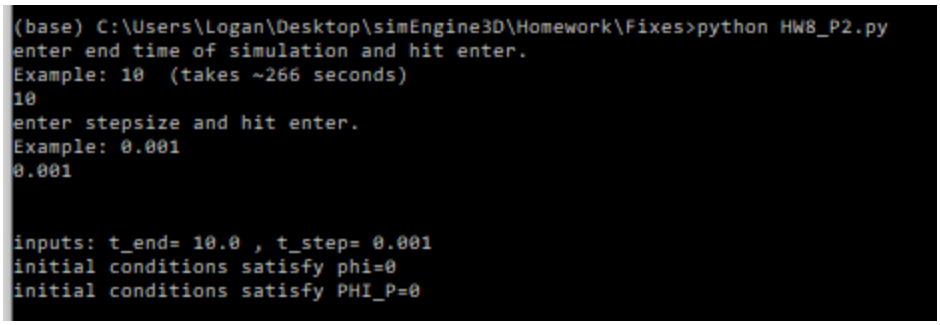

Figure 10 - Command line example of running code

6. Conclusions and Future Work

I was hoping to be able to compare some of my results with results from Pychrono but was unable to get Pychrono working to a level at which I could compare the results shown in Section 4. I was also hoping to make the body/constraint information a little easier for the user to input. Unfortunately, I ran out of time to implement a better method than the .txt input file. My plan was to use a GUI style interface where one could add bodies and enter in all relevant information interactively. This would obviously be a pretty significant undertaking, but if I were to continue working on this code, I think it would be cool/useful to do.

References

I used a figure from HW8 and used some other students' homework code that was posted to github for inspiration for my own code (no code was directly copied).