MAT 460: NUMERICAL DIFFERENTIAL EQUATIONS

Mathematical Modeling of SpaceX's Falcon 9 Booster Hoverslam Landing

Wakeley Banker and Chase Francis

May 6, 2022

ABSTRACT

This paper will examine various numerical methods to model the Hoverslam maneuver performed during the landing of a Falcon 9 first stage rocket booster. The 2nd order ordinary delay differential equation with constraints consists of gravity, air resistance, and thrust terms. It will be modeled using Runge-Kutta, Euler Implicit, Explicit, and Symplectic. The differential equation represents the vertical acceleration of the rocket booster as it approaches sea level. The numerical methods employed will be tested with varying step size and presented to show that they are suitable methods to model this special type of differential equation application. They will be used to find the optimal ignition time of the booster, from a known starting position, in order to land with a vertical velocity ($\dot{h}$) and height (h) of 0 with respect to sea level.

CONTENTS

LIST OF FIGURES

LISTINGS

1 Introduction

1.1 Background

SpaceX is an American aerospace company founded in 2002 with the primary goal of advancing space travel technologies, and commercializing space flight. In order to accomplish this goals in the future, the company prioritizes the economic side of space travel. A decrease in cost per launch will result in more launches, and more availability to the public. Since 2015, SpaceX has been performing Falcon 9 rocket booster landings. This entails the booster detaching from the rocket with some remaining fuel and returning back to earth to land upright for future use. The Hoverslam maneuver is the method employed and coined by SpaceX to successfully land the boosters. It entails the free fall of the booster in an upright manner until it is ignited with the goal of achieving a vertical velocity of 0 m/s when it reaches the surface of the Earth. This method has resulted in 89 successful first stage booster landings since its implementation and it only costs the company 62 million USD to launch.

1.2 Differential Equation

The second order delay differential equation with constraints to model the vertical acceleration of the booster is given as

$$\begin{aligned} \ddot{h} &= \frac{-GM_{\oplus}}{(R_{\oplus}+h)^2} + \frac{1}{2m(t)}\rho C_D A\dot{h}^2 - \frac{T}{m(t)} \\ \ddot{h} &= Gravity(h) + AirResistance(t,h,\dot{h}) + Thrust(t) \end{aligned} \tag{1.1}$$

where G, $M_{\oplus}$, and $R_{\oplus}$ are astronomical coefficients associated with the Earth in the Gravity term in this situation. For the Air Resistance term, $\rho(h)$ represents the density of air as a function of vertical height, and C_D is the drag coefficient. The Falcon 9 booster has a circular cross-sectional area with a radius of 1.83m and it's mass varies with the function m(t) shown below in equation 1.2. Finally, the Thrust term consists of T, which is the Thrust produced by the booster in kN.

$$m(t) = m_d + m_p - b(t_{burning}) \tag{1.2}$$

The mass of the booster varies as a function of time while the booster is producing thrust and consuming the mass of the propellant. In the above equation m_d is the dry mass of the booster and m_p is the mass of the propellant. The variable b is the rate of propellant consumption in kg/s and $t_{burning}$ is the elapsed time since the booster engine has been ignited. $t_{burning}$ is derived as t - t_{ignite}, where t is the overall time since t_0 at which the velocity ($\dot{h}$) and height (h) with respect to sea level is known.

1.3 Assumptions

The assumptions made during the landing will be that the booster will be landing on Earth. The values for G, $M_{\oplus}$, and $R_{\oplus}$ will all be unique to Earth and will be different for other planets. For this situation it will be assumed that the dry mass of the booster is 27,200kg. The thrusters will be burning fuel at a rate of 1480kg/s and the thrust created will be 7600kg. While the booster is falling the drag coefficient will be assumed to be 0.5. For the purpose of this modeling the assumption will be made that the throttle will remain constant and that all motion is directed toward the center of the Earth. Finally, the starting position of the booster is 150km above the surface of the Earth with a velocity of -2km/s.

1.4 Evaluation and Constraints

This differential equation has complicated boundaries and constraints that vary depending on the test. The differential equation is described as a delay differential equation because the final term (Thrust) is only added to the equation for $t \geq t_{ignite}$. Also, the mass (m(t)) only varies after the booster has been ignited as well. Next, the boundaries of evaluation. As stated previously, t_0 is the starting position in the flight of the booster. The initial conditions of $\dot{h}$ and h are known and serve as the starting point for the numerical methods to begin their evaluation. The secondary boundary is when the integration should be terminated. This boundary is determined based on which constraint is violated first. The integration is to be halted if the ground is reached (h = 0) or if the propellant is exhausted ($m_p = 0$). These boundaries will vary as the time of ignition is changed to find the optimal ignition time to achieve a safe and successful landing.

2 Numerical Methods

2.1 2nd Order Differential Equation

The 2nd order differential equation that is used in this situation and throughout mechanics is of the form:

$$\ddot{h} = f(t, h, \dot{h}), \tag{2.1}$$

where $\ddot{h}(t)$ is the acceleration, $\dot{h}(t)$ is the velocity, and $h(t)$ is the position of the object. In this case of radial fall the motion is going to be 1 dimensional, with h being the vertical position with respect to the surface of the Earth. The motion is directed inwards towards the center of the Earth. In order to begin evaluation the initial position, $h_0 = h(t_0)$, and velocity, $v_0 = v(t_0)$ is declared. The complexity of the differential equation is increased greatly from the generic free fall of a falling mass by the adding of the gravitational potential of the Earth term and further with adding the air resistance term. These terms coupled with f changing explicitly with time and height make the process of finding an analytical solution challenging. The numerical methods that are used in the following sections will use the initial conditions mentioned previously to model the differential equation in the form of equation 2.1.

2.2 Symplectic Euler Method

2.2.1 Conversion from 1st to 2nd Order Compatibility

The symplectic or semi-implicit Euler method is a manipulation of Euler's method in order to make it applicable to second-order initial value problems. As shown in equation 2.2, Euler's method which he published in 1768, is used to evaluate a first-order differential equation of the form $\dot{h} = f(t,h)$.

$$h_{n+1} = h_n + f(t_n, h_n)\Delta t \tag{2.2}$$

In order to manipulate the method a reduction of order approach can be taken. The second order differential equation of the form shown in equation 2.1 can be represented by

$$\begin{aligned} u &= \dot{h} \\ \dot{u} &= f(t,h,\dot{h}) \end{aligned} \tag{2.3}$$

which, is now a system of two first order differential equations that are equivalent to the single second-order differential equation. The initial conditions described previously are now changed to

$$\begin{aligned} h(t_0) &= h_0 \\ u(t_0) &= u_0 \end{aligned} \tag{2.4}$$

If a graph is plotted of time vs. height starting from t_0 and h_0, the slope of the line is u_0. Next, if a time of $t_0 + \Delta t$ is defined on the horizontal axis and its corresponding h value is found, the line segment between (t_0, h_0) and $(t_0 + \Delta t, h(t_0 + \Delta t))$ is the approximation to the solution. It has a slope of u_0. Then, if a graph of time vs. u is created with (t_0, u_0) and $(t_0 + \Delta t, u(t_0 + \Delta t))$ plotted with a slope of $f(t_0, h_0, u_0)$. The line segment is the approximation of the solution between t_0 and $t_0 + \Delta t$. This yields equations 2.5,

$$\begin{aligned} t_1 &= t_0 + \Delta t \\ h_1 &= h_0 + \Delta t u_0 \\ u_1 &= u_0 + \Delta t f(t_0, h_0, u_0) \end{aligned} \tag{2.5}$$

which are then used to iterate. The values found will then be used to find t_2, h_2, and u_2. The forward or Explicit Euler method in standard form is shown below and starts from the initial values of h_0 and $\dot{h}_0$.

$$\begin{aligned} h_{n+1} &= h_n + \dot{h}_0 \Delta t, \\ \dot{h}_{n+1} &= \dot{h}_n + f(t_n, h_n, \dot{h}_n)\Delta t \end{aligned} \tag{2.6}$$

2.2.2 Improvements to Explicit Euler Scheme (Symplectic)

The forward Euler scheme that was derived in the previous section is an example of an explicit scheme. An explicit scheme finds values at a later time based on the values found at previous times. It estimates h_{n+1} and $\dot{h}_{n+1}$ at the time $t_n + \Delta t$, by using $\dot{h}_n$ and $f(t_n, h_n, \dot{h}_n)$ found at the previously time t_n. This method can be improved by combining both explicit and implicit schemes. An implicit scheme finds a solution by solving an equation at the current time and a later time. These methods are both used extensively in numerical methods. However, the approximation can be improved by combining the two types. This is called a symplectic method and it allows for the limiting of error by being phase space preserving. The accumulated error in the approximation doesn't grow over time as a result of this method. The symplectic Euler scheme is shown below in equation 2.7.

$$\begin{aligned} \dot{h}_{n+1} &= \dot{h}_n + f(t_n, h_n, \dot{h}_n)\Delta t \\ h_{n+1} &= h_n + \dot{h}_{n+1}\Delta t \end{aligned} \tag{2.7}$$

The top equation is the explicit Euler forward method derived in Section 2.2.1 and the bottom equation is the implicit Euler equation used to find the height h_{n+1} using the velocity $\dot{h}_{n+1}$ that is already found at that time using the explicit equation. This symplectic Euler method will be the first numerical method employed to model the Hoverslam maneuver.

2.3 Implicit Euler Method

2.3.1 Conversion from 1st to 2nd Order Compatibility

The Euler method has now been derived to be applicable to 2nd order differential equations in both symplectic and explicit schemes. In this section it will be a fully implicit scheme of the Euler method also known as the backward Euler method. An implicit scheme that calculates the system at a future time from the given system at present and future times. The backward Euler method for first order ODEs is given as:

$$\begin{aligned} h_n &= h_{n+1} - f(t_{n+1}, h_{n+1})\Delta t \\ h_{n+1} &= h_n + f(t_{n+1}, h_{n+1})\Delta t \end{aligned} \tag{2.8}$$

The second order applicable Euler implicit is more complex than the explicit. It can only be applied by solving two sets of linear equations. The linear system is derived by solving for h_{n+1} in the equation $h_{n+1} = h_n + \Delta t A h_{n+1}$. This equation also holds true for $\dot{h}_{n+1}$ if you change all h terms to $\dot{h}$ terms. These two equations are shown below under equation 2.9. They are solved each individually in each step and they are an entirely implicit scheme.

$$\begin{aligned} [I - \Delta t A][h_{n+1}] &= h \\ [I - \Delta t A][\dot{h}_{n+1}] &= \dot{h} \end{aligned} \tag{2.9}$$

2.4 Runge-Kutta Method

2.4.1 Conversion From 1st To 2nd Order Compatibility

The Euler method is one of the simplest methods of solving ordinary differential equations. It is only first-order accurate, while the Runge-Kutta method is fourth-order accurate. This higher-order accuracy is computed by estimating the slopes at different times on the interval between $t+\Delta t$.These estimates are then combined using a weighted average to achieve a numerical scheme that is more accurate. The Runge-Kutta scheme for first order differential equations is given in equation 2.10.

$$\begin{aligned}
k_1 &= f(t,h)\Delta t,\\
k_2 &= f(t+\Delta t/2, h+k_1/2)\Delta t,\\
k_3 &= f(t+\Delta t/2, h+k_2/2)\Delta t,\\
k_4 &= f(t+\Delta t, h+k_3)\Delta t,\\
h(t+\Delta t) &= h(t)+\frac{1}{6}(k_1+2k_2+2k_3+k_4)
\end{aligned} \tag{2.10}$$

In a similar fashion to the Euler method the given Runge-Kutta method above can be transformed to be applicable to a second order ordinary differential equation. The reduction of order method can be utilized to represent a second order ODE as two first orders, as shown in equation 2.3. The following equations make the Runge-Kutta method valid for a 2nd order ODE. They are in the order that they will have to be computed and implemented into Python and they calculate the k factors to be used.

$$\begin{aligned}
k_{1h} &= f_1(t_n, h_n, \dot{h}_n)\\
k_{1\dot{h}} &= f_2(t_n, h_n, \dot{h}_n)\\
k_{2h} &= f_1(t_n+\frac{\Delta t}{2}, h_n+\frac{\Delta t k_{1h}}{2}, \dot{h}_n+\frac{\Delta t k_{1\dot{h}}}{2})\\
k_{2\dot{h}} &= f_2(t_n+\frac{\Delta t}{2}, h_n+\frac{\Delta t k_{1h}}{2}, \dot{h}_n+\frac{\Delta t k_{1\dot{h}}}{2})\\
k_{3h} &= f_1(t_n+\frac{\Delta t}{2}, h_n+\frac{\Delta t k_{2h}}{2}, \dot{h}_n+\frac{\Delta t k_{2\dot{h}}}{2})\\
k_{3\dot{h}} &= f_2(t_n+\frac{\Delta t}{2}, h_n+\frac{\Delta t k_{2h}}{2}, \dot{h}_n+\frac{\Delta t k_{2\dot{h}}}{2})\\
k_{4h} &= f_1(t_n+\Delta t, h_n+\Delta t k_{3h}, \dot{h}_n+\Delta t k_{3\dot{h}})\\
k_{4\dot{h}} &= f_2(t_n+\Delta t, h_n+\Delta t k_{3h}, \dot{h}_n+\Delta t k_{3\dot{h}})
\end{aligned} \tag{2.11}$$

These k factors are then combined in the weighted average equations shown below.

$$\begin{aligned}
h_{n+1} &= h_n+\frac{\Delta t}{6}(k_{1h}+2k_{2h}+2k_{3h}+k_{4h})\\
\dot{h}_{n+1} &= \dot{h}_n+\frac{\Delta t}{6}(k_{1\dot{h}}+2k_{2\dot{h}}+2k_{3\dot{h}}+k_{4\dot{h}})
\end{aligned} \tag{2.12}$$

3 Stability and Error

Stability in mathematics is the condition in which a slight disturbance in a system does not produce too disrupting an effect on that system. In terms of the solution of a differential equation, a function f(x) is said to be stable if any other solution of the equation that starts out sufficiently close to it when x = 0 remains close to it for succeeding values of x. If the difference between the solutions approaches zero as x increases, the solution is called asymptotically stable. If a solution does not have either of these properties, it is called unstable.

3.1 Euler Stability

The Euler methods are used to evaluate first order differential equations and can be numerically unstable, especially for stiff equations, meaning that the numerical solution grows very large for equations where the exact solution does not. This could cause a large error but, decreasing the step size is one way to minimize the error in the Euler methods. There are more complicated methods that can achieve a higher stability and more accurate solutions.

3.2 Runge-Kutta Stability

The Runge-Kutta method is a fourth order accurate numerical method. This means the error of the method doesn't amplify as easily as the Euler method. The instability of explicit Runge–Kutta methods motivates the development of implicit methods. The advantage of implicit Runge–Kutta methods over explicit ones is their greater stability, especially when applied to stiff equations. The stability function of an explicit Runge–Kutta method is a polynomial. The accuracy of the Runge-Kutta method is much higher compared to the Euler methods and allows for a method that can model more complicated differential equations.

4 Method Testing and Comparison

As discussed in the previous sections each numerical method has its limitations and employs a different approach to modeling a ordinary differential equation. In this section the numerical methods are going to be tested for a second order differential equation that has an analytical solution. This will provide a basis for the accuracy comparison. Also, the graph will give a visualization of the error and limitations of each.

The test case that is going to be used for the comparison is the displacement and velocity of simple harmonic motion. The 2nd order differential equation is rather rudimentary and will provide a simple and clean visualization. The ODE is shown below,

$$\ddot{x} = (-\frac{k}{m})x \tag{4.1}$$

where, k is the spring constant in Newtons/meter and m is the mass in kilograms. The variable x is the position and $\ddot{x}$ is the 2nd time derivative of x, which is the acceleration. The analytical solution is common knowledge and it is shown below for the position and velocity.

$$\begin{gathered} x = x_0 cos(\sqrt{\frac{k}{m}}t) \\ \dot{x} = -x_0 sin(\sqrt{\frac{k}{m}}t) \end{gathered} \tag{4.2}$$

Using Python, the analytical solutions were graphed on separate graphs for position and velocity. Next, all four of the methods were coded to work for two functions. The first function (f_1) was the velocity (v) and the second function (f_2) was the acceleration differential equation. The 2nd order applicable methods derived in section 2 were all set to various step sizes to test the accuracy of the methods. Finally, they were all overlaid onto the graphs with the analytical solution for the same step size and the graphs are shown in Figure 7.6 in the appendix.

The test case confirmed that Runge-Kutta is the most accurate model tested. The fourth order accurate Runge-Kutta model (red) is almost perfectly overlaid onto the analytical solution (green) line in both graphs. Mean absolute percentage error or MAPE is a measure of prediction accuracy of a forcasting method in statistics. This is commonly used with machine learning and predicting models.The equation is shown below and it is modeled in Python for each method,

$$MAPE = \frac{1}{n}\sum_{t=1}^{n} |\frac{A_t - F_t}{A_t}| \tag{4.3}$$

where, n is the number of samples taken for the method (domain/step size), A_t is the analytical value, and F_t is the predicted value. Runge-Kutta had a mean percentage error of $7.18X10^{-9}$ with a step size of 0.1. The Euler Symplectic modeled the differential equation the second best with a MAPE of $7.18X10^{-6}$. It proved to be the most accurate Euler method with Euler implicit being a severe underestimation (MAPE = $3.47X10^{-3}$) and explicit a severe overestimation (MAPE =$5.63X10^{-3}$). The two primary methods of modeling going forward were Euler Symplectic and Runge-Kutta because their MAPE values are shown to be many times better.

5 Modeling and Numerical Simulation

5.1 Python Set-Up

The methods that have been derived, analyzed, and tested are now being applied to the 2nd order delay differential equation that models the vertical acceleration of the Falcon 9 first stage booster Hoverslam Landing maneuver. The differential equation was modeled in Python using a plethora of functions for all four methods. The top two methods Runge-Kutta and Euler symplectic are possible to visualize and the code is detailed below and in the appendix under Listing 4. These functions include gravity(h), rho air(h), mass(t), fuel amount term(t) and Thrust(t). These functions were then referenced in a function called hddot(t,h,hdot), which combined them to represent the differential equation. It had an if-else statement built in to only add the Thrust term for when $t > t_{ignite}$. This was then run through the Runge-Kutta and Euler symplectic functions in hddotwithRkn(dt) and hddotwithEulerSymplectic(dt). These functions had a while loop that only ran the code for when the height of the booster is greater than zero and $m_p > 0$. It then stored the data in a three column list (t,h,hdot), graphed time vs, height, and graphed height vs vertical velocity for each function.

5.2 COMPARISON

In order to compare the two models they were overlapped on the same graphs and compared with varying step sizes and ignition times. It became very obvious that the Runge-Kutta could handle the rapid changes in the position of the booster when the maneuver took place and Euler symplectic could not. The graphs of the motion are shown in the appendix under Figure 7.3. The numerical methods both have a step size of 0.02 and an ignition time of 35.88 seconds. In the appendix under Figures 7.7-7.9 there are tables that compare the step size and ignition time for each method.

5.3 NUMERICAL SIMULATION

In order to make the visualization more clear, an animation was created in Python using matplotlib.animation. The animation was created for both Euler Symplectic and Runge-Kutta. The animation is created using similar functions to update the frame. However, the data was split in order to get a functioning visual. The animation is shown for about the last 20 seconds of rocket booster motion. It plots time vs. height and the rocket is represented by the circle. The images of the overall motion of the rocket are attached in the appendix under Figures 7.4 and 7.5.

6 CONCLUSION

As predicited, the Runge-Kutta method outperformed all other numerical methods. It showed the most competent results in trying to model the quickly changing motion of the Hoverslam maneuver. The best trial is shown in Figure 7.7 for an ignition time of 35.88 seconds. The Runge-Kutta method shows a velocity of 0m/s at a height of 2.6m with a step size of 0.02. These are almost optimum landing parameters for landing the boosters and the best results that were computed. The rocket then has a positive velocity and gains altitude before $m_p = 0$ due to the excess fuel left in the booster. So, the engine can be throttled down at that time to perform a safe landing. This trial is the one graphed in all motion figures for the rocket and illustrate this flight path. However, when the Euler symplectic model gets close to a height of 0m the rapid changes prove to be too much for the numerical method to adapt with because it is only first order accurate.

The Figures 7.7-7.9 illustrate the overall stability of the methods as well. It is shown that both methods can be extremely volatile and vary greatly when just the step size or time is changed slightly. The second half of Figure 7.7 shows the height when the velocity is 0m/s to show just how close the booster was to landing safely. The explicit Runge-Kutta method proved to be the best method out of the four employed. However, it could get out performed by another high order accurate method or an implicit Runge-Kutta scheme in the future.

7 Appendix

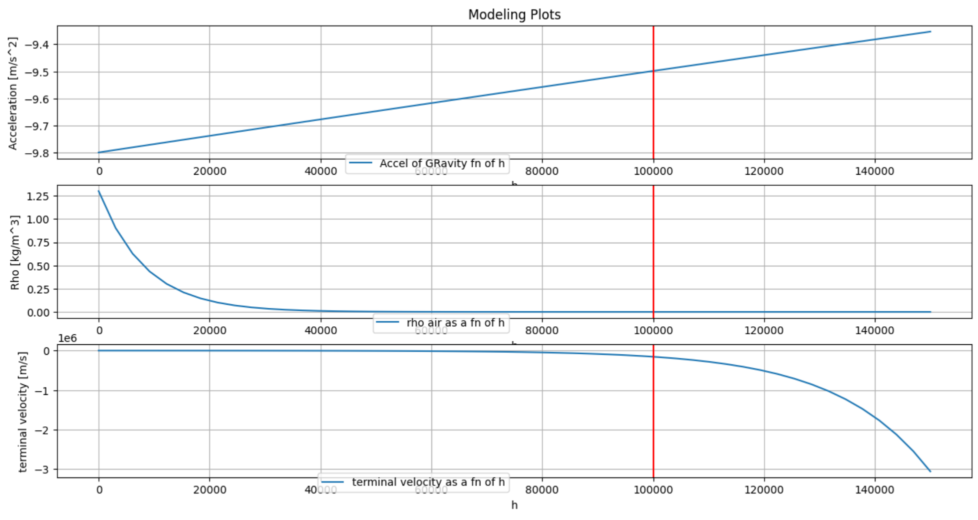


Figure 7.1: Modeling Plots for ρ_{air}, g, and v_{term}

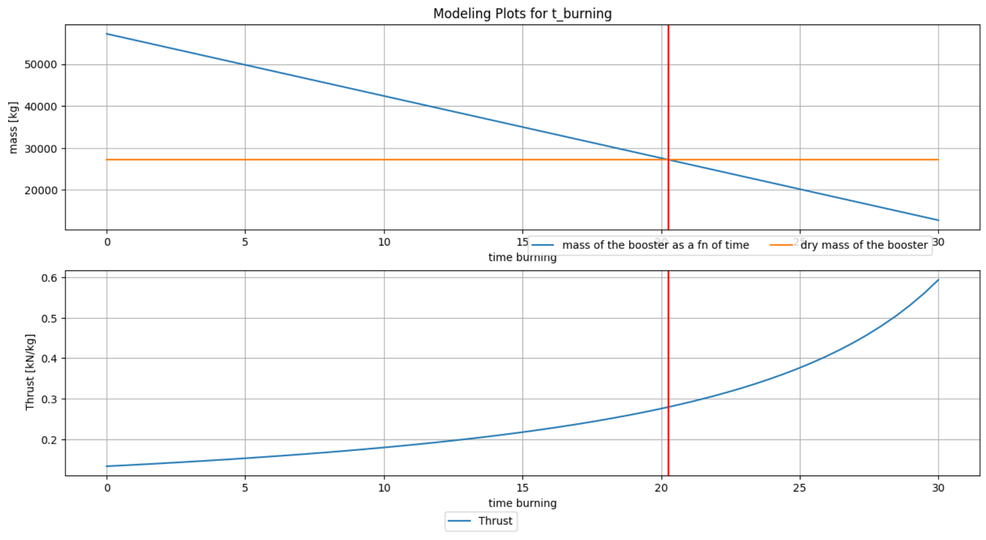


Figure 7.2: Modeling Plots for booster thrust

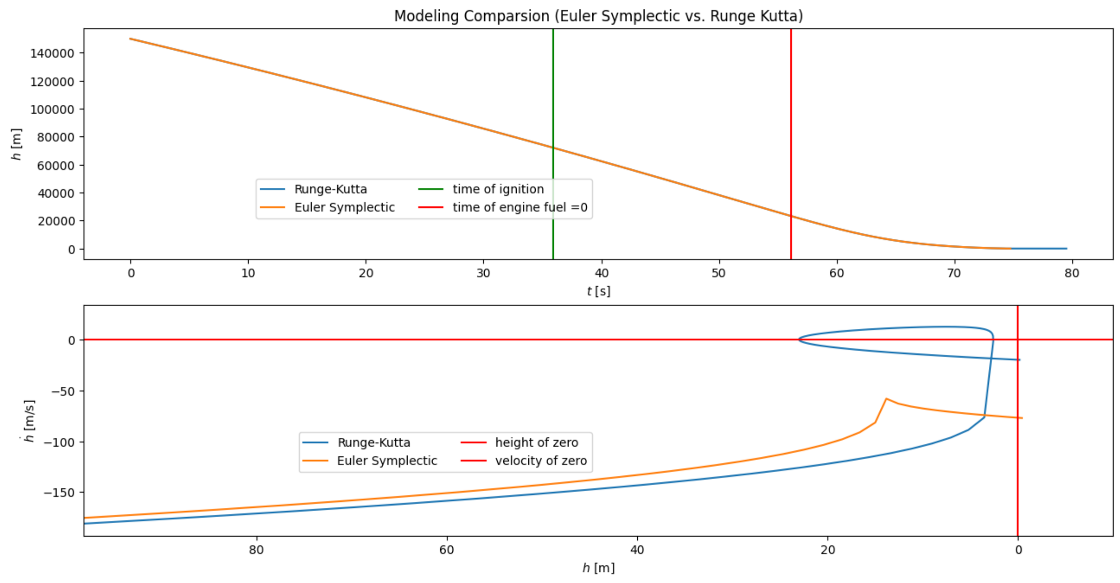


Figure 7.3: Modeling Comparison for Hoverslam Maneuver

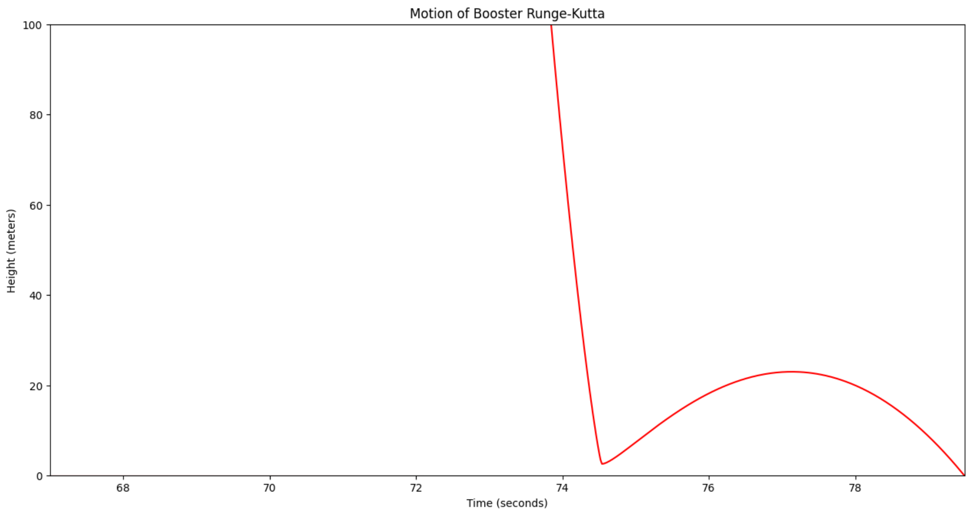


Figure 7.4: Time vs. Height Hoverslam Maneuver (Runge-Kutta)

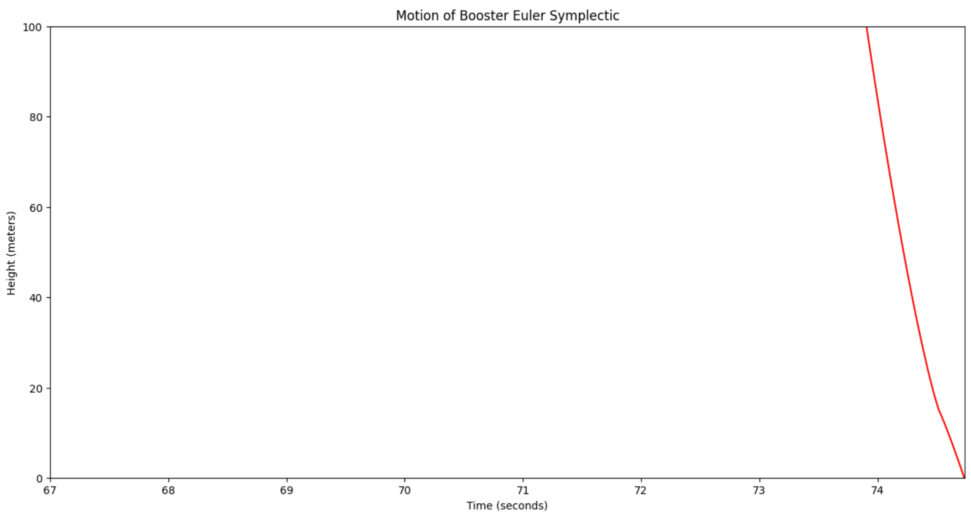


Figure 7.5: Time vs. Height Hoverslam Maneuver (Euler Symplectic)

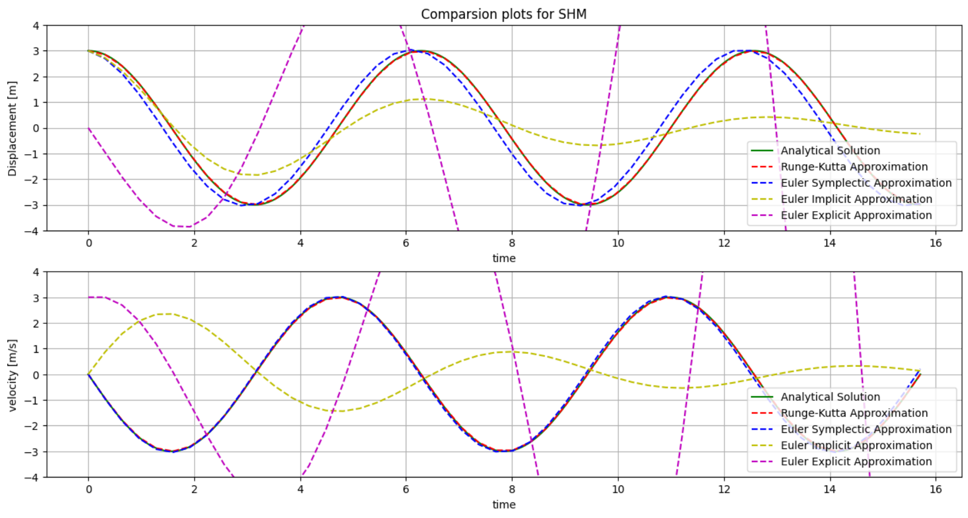
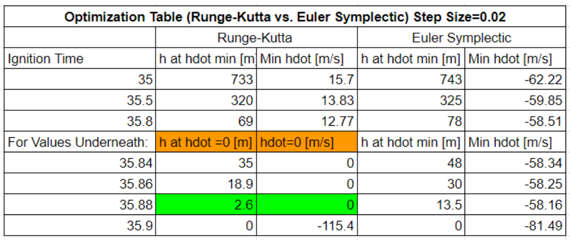
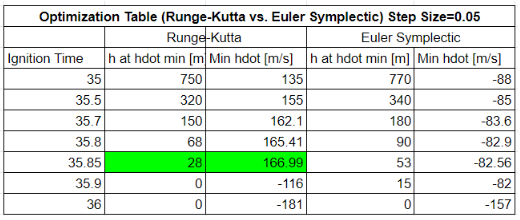


Figure 7.6: Test Case: Simple Harmonic Motion



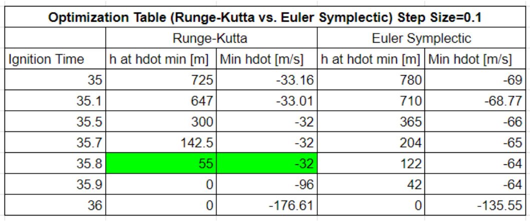
Optimization Table (Runge-Kutta vs. Euler Symplectic) Step Size=0.02				
	Runge-Kutta		Euler Symplectic	
Ignition Time	h at hdot min [m	Min hdot [m/s]	h at hdot min [m]	Min hdot [m/s]
35	733	15.7	743	-62.22
35.5	320	13.83	325	-59.85
35.8	69	12.77	78	-58.51
For Values Underneath:	h at hdot =0 [m]	hdot=0 [m/s]	h at hdot min [m]	Min hdot [m/s]
35.84	35	0	48	-58.34
35.86	18.9	0	30	-58.25
35.88	2.6	0	13.5	-58.16
35.9	0	-115.4	0	-81.49

Figure 7.7: t_{ignite} Optimization Table for step size of 0.02



Optimization Table (Runge-Kutta vs. Euler Symplectic) Step Size=0.05				
	Runge-Kutta		Euler Symplectic	
Ignition Time	h at hdot min [m	Min hdot [m/s]	h at hdot min [m]	Min hdot [m/s]
35	750	135	770	-88
35.5	320	155	340	-85
35.7	150	162.1	180	-83.6
35.8	68	165.41	90	-82.9
35.85	28	166.99	53	-82.56
35.9	0	-116	15	-82
36	0	-181	0	-157

Figure 7.8: t_{ignite} Optimization Table for step size of 0.05



Optimization Table (Runge-Kutta vs. Euler Symplectic) Step Size=0.1				
	Runge-Kutta		Euler Symplectic	
Ignition Time	h at hdot min [m]	Min hdot [m/s]	h at hdot min [m]	Min hdot [m/s]
35	725	-33.16	780	-69
35.1	647	-33.01	710	-68.77
35.5	300	-32	365	-66
35.7	142.5	-32	204	-65
35.8	55	-32	122	-64
35.9	0	-96	42	-64
36	0	-176.61	0	-135.55

Figure 7.9: t_{ignite} Optimization Table for step size of 0.1

```python
import numpy as np
import scipy
import matplotlib.pyplot as plt
from astropy.constants import G,M_earth,R_earth

#Modeling Plots for refrences and understanding of the mechanics that
    go into the differential equation
#Global Variables
g = 9.81
def main():
    a = 0
    b = 150000
    n = 50

    h = np.linspace(a, b, n)
    ygrav = Gravity(h)
    yrhoair = rho_air(h)
    yterminalvelocity = -1 * terminal_velocity(h)

    plt.subplot(3, 1, 1)
    plt.title("Modeling Plots")
    plt.plot(h, ygrav, label='Accel of GRavity fn of h')
    plt.axvline(x=100000, color='r')
    plt.legend(loc= 'lower right' , bbox_to_anchor=(0.5, -0.15), ncol
   =2)
    plt.grid(True)
    plt.xlabel('h')
    plt.ylabel('Acceleration [m/s^2]')

    plt.subplot(3, 1, 2)
    plt.plot(h, yrhoair, label='rho air as a fn of h')
    plt.axvline(x=100000, color='r')
    plt.legend(loc="lower right", bbox_to_anchor=(0.5, -0.15), ncol=2)
    plt.grid(True)
    plt.xlabel('h')
    plt.ylabel('Rho [kg/m^3]')


    plt.subplot(3, 1, 3)
    plt.plot(h,yterminalvelocity , label='terminal velocity as a fn of
   h')
    plt.axvline(x=100000, color='r')
    plt.legend(loc="lower right", bbox_to_anchor=(0.5, -0.15), ncol=2)
    plt.grid(True)
    plt.xlabel('h')
    plt.ylabel('terminal velocity [m/s]')
    plt.show()

    c = 0
    d = 30
    n = 60
    t_burning = np.linspace(c,d,n)
    ymass = mass(t_burning)
    ydrymass = 27200 * ((t_burning)**0)
    boosterThrust = Thrust(t_burning)

    plt.subplot(2, 1, 1)
    plt.title("Modeling Plots for t_burning")
    plt.plot(t_burning, ymass, label='mass of the booster as a fn of
   time')
    plt.plot(t_burning, ydrymass,label = 'dry mass of the booster')
    plt.axvline(x=20.27, color='r')
    plt.legend(loc="lower left", bbox_to_anchor=(0.5, -0.15), ncol=2)
    plt.grid(True)
    plt.xlabel('time burning')
    plt.ylabel('mass [kg]')
    #Intersection at 20.27s


    plt.subplot(2, 1, 2)
    plt.plot(t_burning, boosterThrust, label='Thrust')
    plt.axvline(x=20.27, color='r')
    plt.legend(loc="upper right", bbox_to_anchor=(0.5, -0.15), ncol=2)
    plt.grid(True)
    plt.xlabel('time burning')
    plt.ylabel(' Thrust [kN/kg]')

    plt.show()

    e = 0
    f = 500
    n = 50
    t = np.linspace(e,f,n)
    yposition = vertical_position(t)

    plt.subplot(3, 1, 1)
    plt.title("Modeling Plots for time")
    plt.plot(t, yposition, label='position of object no AR')

    plt.axvline(x=64.5, color='r')
    plt.legend(loc="lower left", bbox_to_anchor=(0.5, -0.15), ncol=2)
    plt.grid(True)
    plt.xlabel('time [s]')
    plt.ylabel('height [m]')

    yvelocity = vertical_velocity(vertical_position, t, 0.01)
    plt.subplot(3,1,2)
    plt.plot(t, yvelocity, label='velocity of object no AR')
    plt.axvline(x=64.5, color='r')
    plt.legend(loc="lower left", bbox_to_anchor=(0.5, -0.15), ncol=2)
    plt.grid(True)
    plt.xlabel('time [s]')
    plt.ylabel('velocity [m/s]')

    yacceleration = vertical_acceleration(vertical_position, t, 0.01)
    plt.subplot(3, 1, 3)
    plt.plot(t, yacceleration, label='acceleration of object no AR')
    plt.axvline(x=64.5, color='r')
    plt.legend(loc="lower left", bbox_to_anchor=(0.5, -0.15), ncol=2)
    plt.grid(True)
    plt.xlabel('time [s]')
    plt.ylabel('acceleration [m/s^2]')


    plt.show()
    print("DONE...")
def Gravity(h):
    gravtop = -1*G.value*M_earth.value
    gravbottom  = (R_earth.value + h)**2
    gravity = gravtop / gravbottom
    return gravity
def rho_air(h):
    return 1.3*np.exp((-h)/ (8.4e3))
def mass(t_burning):
    m_d = 27200
    m_p = 3.0e4
    b = 1480
    t_ignite = 50
    return m_d + (m_p -( b * t_burning))
def Thrust(t_burning):
    T = 7.6e3
    return T / mass(t_burning)
def terminal_velocity(h):
    #for full weight of booster
    mass_total = 57200
    C_D = 0.5
    R = 1.83
    A = np.pi * ((R)**2)
    vmaxtop = 2*g*mass_total
    vmaxbottom = rho_air(h)*C_D*A
    vmax = np.sqrt(vmaxtop/vmaxbottom)
    return vmax
def vertical_position(t):
    V0y = -2000
    h0 = 150000
    return V0y*(t) - (0.5*g*(t)**2) + h0
def vertical_velocity(vertical_position, xderv,h):
    return (vertical_position(xderv + h) - vertical_position(xderv - h)
   ) / (2 * h)
def vertical_acceleration(vertical_position, xderv,h):
    return (vertical_position(xderv-h)-2*vertical_position(xderv)+
   vertical_position(xderv+h)) / (h**2)

if __name__ == "__main__":
    main()

```

Listing 1: Modeling Plots Code

```python
#Imported libraries
import numpy as np
import matplotlib.pyplot as plt
#Global variables of k constant, and runtime of 5pi
k = 1
Runtime = 5 * np.pi
#Samples for a setep size of 0.1 and to plot the analytical solution
Samples = 157
Step_Size = Runtime / (Samples - 1)
#intial conditions and mass=1
m = 1
x0 = 3
v0 = 0

#The samples below are for each of the methods and for a step of 0.1
SamplesforRunge = 157
t = np.linspace(0, Runtime, SamplesforRunge)
Step_size_Runge = Runtime / (SamplesforRunge - 1)

SamplesforSymplectic = 157
t2 = np.linspace(0, Runtime, SamplesforSymplectic)
Step_size_symplectic = Runtime / (SamplesforSymplectic - 1)

SamplesforImplicit = 157
t3 = np.linspace(0, Runtime, SamplesforImplicit)
Step_size_implicit = Runtime / (SamplesforImplicit - 1)

SamplesforExplicit = 157
t4 = np.linspace(0, Runtime, SamplesforExplicit)
Step_size_explicit = Runtime / (SamplesforExplicit - 1)


time = np.linspace(0, Runtime, Samples)


def main():
    #Plot analytical soltuion
    plt.subplot(2,1,1)
    x_exactfn = np.vectorize(x_exact)(time)
    plt.plot(time,x_exactfn,color='green',label = 'Analytical Solution'
   )
    plt.title("Comparsion plots for SHM")
    plt.grid()
    plt.xlabel("time")
    plt.ylabel("Displacement [m]")
    plt.ylim(-4,4)

    #Below each of the methods is ran and labeled (Position)
    x,v = Runge_Kutta()
    plt.plot(t,x, 'r--',label = "Runge-Kutta Approximation")


    xsym,vsym = Euler_Symplectic()
    plt.plot(t2, xsym, 'b--', label="Euler Symplectic Approximation")


    ximp, vimp = Euler_Implicit()
    plt.plot(t3, ximp, 'y--', label="Euler Implicit Approximation")

    xexp, vexp = Euler_Explicit()
    plt.plot(t4, xexp, 'm--', label="Euler Explicit Approximation")
    plt.legend(loc='lower right')

    #(Velocity)
    plt.subplot(2,1,2)
    v_exactfn = np.vectorize(v_exact)(time)
    plt.plot(time, v_exactfn, color='green', label='Analytical Solution
   ')
    plt.xlabel("time")
    plt.ylabel("velocity [m/s]")
    plt.grid()
    plt.ylim(-4, 4)


    x, v = Runge_Kutta()
    plt.plot(t, v, 'r--', label="Runge-Kutta Approximation")


    xsym, vsym = Euler_Symplectic()
    plt.plot(t2, vsym, 'b--', label="Euler Symplectic Approximation")

    ximp, vimp = Euler_Implicit()
    plt.plot(t3, vimp, 'y--', label="Euler Implicit Approximation")


    xexp, vexp = Euler_Explicit()
    plt.plot(t4, vexp, 'm--', label="Euler Explicit Approximation")
    plt.legend(loc='lower right')
    plt.show()

    #This outputs the error for each of functions by taking the error
   at each point and adding then dividing by # points
    #would have been simplier to make a function
    for j in range(SamplesforRunge):
        MAPERungeconstant = 1/SamplesforRunge
        errorrunge = sum((x_exactfn[j]-x[j])/x_exactfn[j])
        MAPERunge = MAPERungeconstant*errorrunge
    print("This is the MAPE for Runge:" + str(MAPERunge))

    for j in range(SamplesforSymplectic):
        MAPESymConstant = 1/SamplesforSymplectic
        errorsym = sum((x_exactfn[j]-xsym[j])/x_exactfn[j])
        MAPEsym = MAPESymConstant*errorsym
    print("This is the MAPE for Symplectic:" + str(MAPEsym))
```

```python

    for j in range(SamplesforExplicit):
        MAPEExpconstant = 1 / SamplesforExplicit
        errorexp = sum((x_exactfn[j] - xexp[j]) / x_exactfn[j])
        MAPEexp = MAPEExpconstant * errorexp
    print("This is the MAPE for Euler Explicit:" + str(MAPEexp))

    for j in range(SamplesforImplicit):
        MAPEImpConstant = 1 / SamplesforImplicit
        errorimp = sum((x_exactfn[j] - ximp[j]) / x_exactfn[j])
        MAPEimp= MAPEImpConstant * errorimp
    print("This is the MAPE for Implicit:" + str(MAPEimp))

    #Prints step Sizes
    print("This is current step size for Runge-Kutta: " + str(round(Step_size_Runge,3)))
    print("This is current step size for Euler Symplectic: " + str(round(Step_size_symplectic,3)))
    print("This is current step size for Euler Implicit: " + str(round(Step_size_implicit, 3)))
    print("This is current step size for Euler Explicit: " + str(round(Step_size_explicit, 3)))
    print("DONE....")

#Analytical position
def x_exact(time):
    return x0*np.cos(np.sqrt(k/m)*time)

#Analytical velocity
def v_exact(time):
    return -np.sqrt(k/m)*x0*np.sin(np.sqrt(k/m)*time)

#Differential equation f1=velocity
def f1(t,x,v):
    return v
#Differential equation: f2=-k/m(x)
def f2(t,x,v):
    return -k/m*x

#Fourth order runge kutta for 2nd order diffential equations
def Runge_Kutta():

    #creating the matrices to be filled in by x and v
    h = Step_size_Runge
    x = np.zeros((len(t),1))
    v = np.zeros((len(t),1))
    #initial conditions
    x[0] = x0
    v[0] = v0

    #Runge Kutta
    for i in range(SamplesforRunge-1):
        K1x = f1(t[i],x[i],v[i])
        K1v = f2(t[i], x[i], v[i])
        K2x = f1(t[i] + h/2, x[i]+h*K1x/2, v[i] + h*K1v/2)
        K2v = f2(t[i] + h / 2, x[i] + h * K1x / 2, v[i] + h * K1v / 2)
        K3x = f1(t[i] + h / 2, x[i] + h * K2x / 2, v[i] + h * K2v / 2)
        K3v = f2(t[i] + h / 2, x[i] + h * K2x / 2, v[i] + h * K2v / 2)
        K4x = f1(t[i]+h,x[i]+h*K3x,v[i]+h*K3v)
        K4v = f2(t[i] + h, x[i] + h * K3x, v[i] + h * K3v)
        x[i+1] = x[i] + h/6*(K1x+2*K2x+2*K3x+K4x)
        v[i + 1] = v[i] + h / 6 * (K1v + 2 * K2v + 2 * K3v + K4v)

    return x,v

#Euler Implicit
def Euler_Implicit():
    #Same method approach as Runge
    h = Step_size_implicit
    ximp = np.zeros((len(t3), 1))
    vimp = np.zeros((len(t3), 1))
    ximp[0] = x0
    vimp[0] = v0
    for k in range(SamplesforImplicit - 1):
    #Solving the linear equations system
        A = np.array([[0, 1], [-1, 0]])
        I = np.identity(2)
        b = np.array([vimp[k], ximp[k]])
        vimp[k + 1], ximp[k + 1] = np.linalg.solve(I - (
   Step_size_implicit * A), b)

    return ximp,vimp

#Euler Symplectic
def Euler_Symplectic():
    h = Step_size_symplectic
    xsym = np.zeros((len(t2), 1))
    vsym = np.zeros((len(t2), 1))
    xsym[0] = x0
    vsym[0] = v0
    for k in range(SamplesforSymplectic - 1):

        vsym[k + 1] = vsym[k] + h * f2(t2[k], xsym[k], vsym[k])
        xsym[k+1] = xsym[k]+h*vsym[k+1]

    return xsym,vsym

#Euler Explicit
def Euler_Explicit():
    h = Step_size_explicit
    xexp = np.zeros((len(t4), 1))
    vexp = np.zeros((len(t4), 1))
    xexp[0] = x0
    vexp[0] = v0


    for k in range(SamplesforExplicit - 1):
        xexp[k + 1] = xexp[k] + Step_size_explicit * vexp[k]
        vexp[k+1] = vexp[k]+ Step_size_explicit * f2(t4[k],xexp[k],vexp
    [k])

    return vexp,xexp

if __name__ == "__main__":
        main()

```

Listing 2: Test Case: SHM Code

```
This is the MAPE for Runge:7.183196393916714e-09
This is the MAPE for Symplectic:2.2736460451955488e-06
This is the MAPE for Euler Explicit:0.005631552584641939
This is the MAPE for Implicit:0.0034736158245725255
This is current step size for Runge-Kutta: 0.101
This is current step size for Euler Symplectic: 0.101
This is current step size for Euler Implicit: 0.101
This is current step size for Euler Explicit: 0.101
DONE....

```

Listing 3: MAPE and Step Size (SHM)

```
#Imported libraries
import numpy as np
import scipy
import matplotlib.pyplot as plt
import astropy.units as unit
from astropy.constants import G,M_earth,R_earth
import matplotlib.animation as animation
#Global Variables
T = 7.6e3
g = 9.81
m_d = 27200
C_d = 0.5
R = 1.83
A = np.pi * (R) ** 2
#Initial Values
h0 = 150000
v0 = 2000
#Ignition Time
t_ignite = 35.88
def main():
    #Calling the functions to compute t,h,hdot
    data = hddotwithRkn(0.02)
    data1 = hddotwithEulerSymplectic(0.02)

    #Plotting Modeling Comparsion of Euler Symplectic and Runge-Kutta
    and using the limits you can zoom in
    plt.figure(figsize=(12, 4), dpi=100)
    #Top subplot time vs height
    plt.subplot(2, 1, 1)
    plt.plot(data[:, 0], data[:, 1], label='Runge-Kutta')
    plt.plot(data1[:, 0], data1[:, 1], label='Euler Symplectic')
    #Labeling ignition time and when mp=0
    plt.axvline(x=t_ignite, color='g', label='time of ignition')
    plt.axvline(x=t_ignite + 20.27, color='r', label='time of engine
fuel =0')
    plt.xlabel("$t$ [s]")
    plt.ylabel("$h$ [m]")
    plt.legend(loc="lower right", bbox_to_anchor=(0.5, 0.15), ncol=2)

    plt.title("Modeling Comparsion (Euler Symplectic vs. Runge Kutta)")
    #Bottom subplot height vs velocity for the two
    plt.subplot(2, 1, 2)
    plt.plot(data[:, 1], data[:, 2], label='Runge-Kutta')
    plt.plot(data1[:, 1], data1[:, 2], label='Euler Symplectic')
    plt.axvline(x=0, color='r', label='height of zero')
    plt.axhline(y=0, color='r', label='velocity of zero')
    plt.xlim(h0 + 10, -10)
    plt.xlabel("$h$ [m]")
    plt.ylabel("$\dot{h}$ [m/s]")
    #zoomed in limits
    # plt.xlim(250, 0)
    # plt.ylim(-200, 200)
    plt.legend(loc="lower right", bbox_to_anchor=(0.5, 0.25), ncol=2)
    #Need to zoom in to see end of motion
    plt.show()

    #Split data for end of motion animation
    datarungeani = split(data)
    datasymplecticani = split(data1)

    #animation for Runge Kutta
    t = datarungeani[:, 0]
    h = datarungeani[:, 1]
    fig, ax = plt.subplots()
    line, = ax.plot(t, h, color='r')

    #Setting min and max
    tmin = datarungeani[0, 0]
    hmin = datarungeani[0,1]
    tmax = max(t)
    hmax = max(h)
    xysmall = min(tmax, hmax)
    maxscale = max(tmax, hmax)
    plt.xlabel("Time (seconds)")
    plt.ylabel("Height (meters)")
    plt.title("Motion of Booster Runge-Kutta")

    #Create circle to represent booster
```

```python
    circle = plt.Circle((tmin, hmin), radius=np.sqrt(0.1))
    ax.add_patch(circle)
    ani = animation.FuncAnimation(fig, update, frames = len(t), fargs=[t, h, line,circle],
                                  interval=1, blit=True)
    #ani.save('RungeKuttaEndMotion.gif', fps=60)

    plt.show()

    # animation for Euler Symplectic
    t = datasymplecticani[:, 0]
    h = datasymplecticani[:, 1]
    fig, ax = plt.subplots()
    line, = ax.plot(t, h, color='r')

    # Setting min and max
    tmin = datasymplecticani[0, 0]
    hmin = datasymplecticani[0, 1]
    tmax = max(t)
    hmax = max(h)
    xysmall = min(tmax, hmax)
    maxscale = max(tmax, hmax)
    plt.xlabel("Time (seconds)")
    plt.ylabel("Height (meters)")
    plt.title("Motion of Booster Euler Symplectic")

    # Create circle to represent booster
    circle = plt.Circle((tmin, hmin), radius=np.sqrt(0.1))
    ax.add_patch(circle)
    ani = animation.FuncAnimation(fig, update, frames=len(t), fargs=[t, h, line, circle],
                                  interval=1, blit=True)
    # ani.save('EulerSymplecticEndMotion.gif', fps=60)

    plt.show()
    print("DONE....")



#Update frame which is used for animation and sets axis limits for viewing
def update(num, t,h, line, circle):
    line.set_data(t[:num], h[:num])
    circle.center = t[num], h[num]
    line.axes.axis([67,max(t), 0, 100])

    return line, circle

#Function to split data for the end of motion animation takes last 15 %
def split(data):
    n = len(data[:,0])
    dim = int(n*0.85)
    #print(n)
    #print(n*0.85)
    #print(int(n*0.15))
    datanew = np.zeros((dim, 3))
    for i in range(int(n*.85), int(n)):
        datanew[i-3380] = data[[i]]
    #print(datanew)
    return datanew

#refrences all of the functions and method to return t,h,hdot for symplectic
def hddotwithEulerSymplectic(dt):
    #set data1 as the data and this sets the initial values
    data1 = [[0, h0, -v0]]

    # initialization of loop variables
    t, h, hdot = tuple(data1[0])
    #just for refrence to ensure same starting position as runge
    print("Initial acceleration (Symplectic) = {:.2f} m/s^2".
          format(hddot(0, h, hdot)))
    #hddot is the differnetial equation that we want to model

    #only runs for while height is greate than 0
    while h > 0:
        #calling the Euler symplectic function below
        h,hdot = euler_symplectic(hddot,t,h,hdot,dt)
        #adding the step size to it
        t += dt
    #attaching each line to the data1 set
        data1 = np.append(data1, [[t, h, hdot]], axis=0)
        #print(data1)
    #prints the min velocity to add to the table in Appendix
    print("The minimum velocity (Symplectic) is: " + str(max(data1[:,2])))

    #Graphing same two plots just for Euler Symplectic and can adjust the bounds to show the ending motion
    plt.figure(figsize=(12, 4), dpi=100)
    plt.subplot(2, 1, 1)
    plt.title("Euler Symplectic Modeling")
    plt.plot(data1[:, 0], data1[:, 1], label='height as a fn of time')
    plt.axvline(x=t_ignite, color='g', label='time of ignition')
    plt.axvline(x=t_ignite + 20.27, color='r', label='time of engine fuel =0')
    plt.xlabel("$t$ [s]")
    plt.ylabel("$h$ [m]")
    plt.legend(loc="lower right", bbox_to_anchor=(0.5, -0.15), ncol=2)

    plt.subplot(2, 1, 2)
    plt.plot(data1[:, 1], data1[:, 2], label='the velocity over time vs the height')
    plt.axvline(x=max(data1[:,2]) , color='r', label='minimum velocity'
    )

    plt.xlim(h0, 0)
    plt.xlabel("$h$ [m]")
    plt.ylabel("$\dot{h}$ [m/s]")
    #plt.xlim(200,0)
    #plt.ylim(-100,0)
    plt.legend(loc="lower right", bbox_to_anchor=(0.5, -0.15), ncol=2)
    plt.show()
    return data1


#refrences all of the functions and method to return t,h,hdot for Runge
    -Kutta
def hddotwithRkn(dt):
    #Same method as before except data instead of data1
    data = [[0, h0, -v0]]

    # initialization of loop variables
    t, h, hdot = tuple(data[0])
    print("Initial acceleration (Runge) = {:.2f} m/s^2".
          format(hddot(0, h, hdot)))


    #Only for height is greater than 0
    while h > 0:
        h, hdot = Runge_Kutta(hddot, t, h, hdot, dt)

        t += dt

        data = np.append(data, [[t, h, hdot]], axis=0)

    print("The minimum velocity (Runge) is: " + str(max(data[:, 2])))

    plt.figure(figsize=(12, 4), dpi=100)
    plt.subplot(2,1,1)
    plt.title("Runge-Kutta Modeling")
    plt.plot(data[:, 0],  data[:, 1],label = 'height as a fn of time')
    plt.axvline(x=t_ignite, color='g', label = 'time of ignition')
    plt.axvline(x=t_ignite + 20.27, color='r', label='time of engine
    fuel =0')
    plt.xlabel("$t$ [s]")
    plt.ylabel("$h$ [m]")
    plt.legend(loc="lower right", bbox_to_anchor=(0.5, -0.15), ncol=2)

    plt.subplot(2,1,2)
    plt.plot(data[:, 1], data[:, 2],label = 'the velocity over time vs
    the height')
    #plt.axvline(x=0, color='r', label='height of zero')
    plt.axhline(y=0, color='r', label='velocity of zero')
    plt.xlim(h0 + 10, -10)
    plt.xlabel("$h$ [m]")
    plt.ylabel("$\dot{h}$ [m/s]")
    #plt.xlim(250, 0)
    #plt.ylim(-200, 200)
    plt.legend(loc="lower right", bbox_to_anchor=(0.5, -0.15), ncol=2)
    plt.show()
    return data

#Below are all the functions refrenced above

#Gravity term as a fn of height
def gravity(h):
    gravtop =  G.value * M_earth.value
    gravbottom = (R_earth.value + h) ** 2
    gravity = gravtop / gravbottom
    return gravity

#Density of air as a fn of height
def rho_air(h):
    return 1.3 * np.exp(-h / (8.4e3))

#Mass of booster over time function
def mass(t):
    m_d = 27200
    m_p = 3.0e4
    b = 1480
    burning_max = 21

    if t > t_ignite:
        t_burning = t - t_ignite
    else:
        t_burning = 0
    return m_d + (m_p -( b * t_burning))

#how much fuel is left in the booster function
def fuelamountterm(t):
    m_d = 27200
    m_p = 3.0e4
    b = 1480
    burning_max = 21
    if t > t_ignite:
        t_burning = t - t_ignite
    else:
        t_burning = 0
    fuelamount = (m_p -( b * t_burning))
    if fuelamount <= 0:
        return 0

#Thrust term in the differential equation
def Thrust(t):
    T = 7.6e3
    return T / mass(t_burning)

```

```python
#Combining all of the functions to create the differential equation to
    be used
def hddot(t, h, hdot):

    ARterm = (1/(2*mass(t)))*(rho_air(h) * C_d * A * (hdot) ** 2)
    thrustterm = T / mass(t)
    #Only adding the Thrust term to the equation if t>tignite
    if t > t_ignite:
        hdouble = -gravity(h) + ARterm - thrustterm
        return hdouble
    else:
        hdouble = -gravity(h) + ARterm
        return hdouble
    return hdouble

#Runge-Kutta Method for a 2nd order differential equation in function
    form
def Runge_Kutta(f, t, x, xd, h):
    hsq = h*h
    a = np.array( [0, 0.25, 0.375, 12/13, 1.] )
    c = np.array( [253/2160, 0, 4352/12825, 2197/41040, -0.01] )
    cd = np.array( [25/216, 0, 1408/2565, 2197/4104, -0.2] )
    g = np.array( [[0, 0, 0, 0, 0],
                   [1/32, 0, 0, 0, 0],
                   [9/256,9/256, 0, 0, 0],
                   [27342/142805, -49266/142805, 82764/142805, 0, 0],
                   [5/18, -2/3, 8/9, 0, 0]] )
    b = np.array( [[0, 0, 0, 0, 0],
                   [0.25, 0, 0, 0, 0],
                   [3/32, 9/32, 0, 0, 0],
                   [1932/2197, -7200/2197, 7296/2197, 0, 0],
                   [439/216, -8.,3680/513, -845/4104, 0]] )

    fi = np.zeros(5)
    fi[0] = f(t, x, xd)
    cifisum = fi[0] * c[0]
    cdifisum = fi[0] * cd[0]

    for i in range(1,5):
        ti = t + a[i]*h
        gijfj = 0
        bijfj = 0
        for j in range(0,i):
            gijfj += g[i][j] * fi[j]
            bijfj += b[i][j] * fi[j]
        xi = x + xd*a[i]*h + hsq*gijfj
        xdi = xd + h * bijfj
        #Function form
        fi[i] = f(ti, xi, xdi)
        cifisum += fi[i] * c[i]
        cdifisum += fi[i] * cd[i]
    #Returning the height, velocity in this case
    return ( x + h*xd + hsq*cifisum, xd + h*cdifisum )

#Euler symplectic for a second order differntial equation used
    functions
def euler_symplectic(f, t, x, xdot, h):
    v = xdot + h*f(t, x, xdot)
    #returns h, hdot
    return ( x + h*v, v )


if __name__ == "__main__":
    main()

```

Listing 4: Runge-Kutta and Euler Symplectic Hoverslam Maneuver Modeling/Numerical Simulation Code

8 References

"Falcon 9." SpaceX, https://www.spacex.com/vehicles/falcon-9/. Iserles, Arieh. A First Course in the Numerical Analysis of Differential Equations. Cambridge University Press, 2009.

Schmidt, Wolfram, and Volschow Marcel. Numerical Python in Astronomy and Astrophysics a Practical Guide to Astrophysical Problem Solving. Springer, 2021.

SpaceX Falcon 9 Data Sheet - NASA.

https://sma.nasa.gov/LaunchVehicle/assets/spacex-falcon-9-data-sheet.pdf.

Strang, Gilbert. Computational Science and Engineering. Wellesley-Cambridge Press, 2019.