Assignment 1

NAME : V S HARIKRISHNA

ROLL NO. : 160010054

Problem statement : As a part of this assignment, you will write a python program to calculate the trajectory of a rocket. For the problem statement, see Problem 3.5 (from the problems given at the end of Chapter 3) in Heister's book.

Also, compare the result obtained from using Heun's method (as mentioned in the problem) with that obtained by the inbuilt numerical integration routines in python (you will find such routines in the scipy module of python).

Problem 3.5:

Write a computer code to predict an arbitrary 2-D ballistic trajectory on a flat non-rotating Earth. Consider a rocket with the following characteristics:

$$C_d = 0.3,\ C_L = 0,\ D_{ref} = 3.1in$$

$$F = 18 - 9t,\ 0 < t < 2s\ \ (F\ lb)$$

$$m_0 = 1.1lb,\ I_{sp} = 165s$$

a)Derive values for m_f and $\dot{m}(t)$ based on the information given above.

Using Thrust equation for rockets:

$$\text{Thrust} = \dot{m}(t) g_e I_{sp}$$

Here $-\dot{m}(t)$ is assumed positive.

Since F is given in pounds, we multiply F with g_e to get thrust:

$$\text{Thrust} = F g_e$$

which gives:

$$F.g_e = \dot{m}(t) g_e I_{sp}$$

$$\dot{m}(t) = \frac{F}{I_{sp}}$$

Since F is linearly decreasing with time, burn rate is constant. Thus we can treat the mass change rate $\dot{m}(t)$ as a linearly dcreasing function as:

$$\dot{m}(t) = \frac{F}{I_{sp}} = \frac{18-9t}{165}$$

For mass m_f we need to derive the expression for $m(t)$ as shown:

$$-\frac{dm}{dt} = \dot{m}(t) = \frac{18-9t}{165}$$

The negative sign is because mass is decreasing as rocket consumes its propellant.

Integrating both sides:

$$-\int_{m_0}^{m_f} dm = \int_0^{t_b} \frac{18-9t}{165} dt$$

Putting $m_0 = 1.1lb, t_b = 2s$ gives:

$$m_f = \frac{1635}{1650} = 0.9909lb$$

b)Write your code assuming constant atmospheric conditions, arbitrary wind velocity, and negligible change in gravity. Use Huen's method for the integration and attach a listing of the code.

> The code is listed as below:
>
> We are using a class 'Trajectory_sol () that takes in the following inputs for its objects upon initialization:
>
> n : a number to decide the approximate time span for our trajectory in seconds. Usually kept large. Given in s.
>
> t_b : burn time. Given in s.
>
> C_d : aerodynamic coefficient of drag (default zero)
>
> C_L : aerodynamic coefficient of lift (default zero)
>
> Const_thrust : a flag to specify whether thrust is constant (True or False, default True)
>
> Const_thrust_value : value of the constant thrust if above flag is true. Given in pounds. (default zero)
>
> v_0 : initial velocity of vehicle. Given in ft/s. (default zero)
>
> θ_0 : initial launch angle of vehicle from horizontal measured counter-clockwise from positive x. Given in radians. (default $\frac{\pi}{2}$)
>
> v_w : a 2 by 1 numpy array specifying components of wind veoicty vector along the positive x and y direction respectively as [x-component, y-component]. Given in ft/s. (default $[0,0]$)
>
> h_launchrod : height of launch rod used. Given in feet. (default zero)
>
> The method returns the solved values of vehicle velocity magnitude v and vehicle orientation θ as a 1d numpy arrays of the form $[v]$, $[\theta]$ for every time step. The time step is chosen as 0.01s for this method.
>
> A callable plotting method is also implemented to the class as shown in the case for part c):

```
import numpy as np
from scipy.integrate import solve_ivp, odeint
from math import pi
import matplotlib.pyplot as plt
from tabulate import tabulate



class Trajectory_sol (object):

  def __init__(self, n, t_b, C_d = 0, C_L = 0, Const_thrust = True, \
               Const_thrust_value = 0, v_0 = 0, theta_0 = pi/2, \
               v_w = np.array([0,0]), h_launchrod = 0):
    self.n = n
    # creating an array 't' for time instants between 0s and 'n'seconds with
    # 100n divisions:
    self.t = np.linspace(0, self.n, 100*self.n+1)

    # creating a scalar to denote burn time t_b :
    self.t_b = t_b

    # defining time step dt as:
    self.dt = float(self.t[1]-self.t[0])
```

```
    # defining given values of I_sp, coefficients of drag and lift, reference
    # diameter atmospheric density and gravity as constants in FPS units:
    self.I_sp = 165
    self.C_d = C_d
    self.C_L = C_L
    self.D_ref = 0.258333
    self.rho = 0.074887
    self.g = 32.2
    self.Const_thrust = Const_thrust
    self.Const_thrust_value = Const_thrust_value

    # defining vehicle velocity array 'v', and horizontal angle array 'theta'
    # (measured anticlockwise from horizontal)  as  zero arrays like time array 't':
    self.v = np.zeros_like(self.t)
    self.theta = np.zeros_like(self.t)

    # wind velocity array 'v_w' for storing the horizontal and vertical components
    # of wind velocity in ground frame, taking positive direction as per
    # cartesian system:
    self.v_w = v_w
    self.h_launchrod = h_launchrod

    # defining an array 'F for storing thrust force values at a given time instant
    # in 't'. Given expression is multiplied by g as force is given in pounds
    # in question:


    if self.Const_thrust == True :
      self.F = np.ones_like(self.t)*self.Const_thrust_value*self.g

    else :
      self.F = (18-9*self.t)*self.g

    # calculating the reference area for lift and drag calculation:
    self.A = (pi*self.D_ref**2)/4

    #defining initial mass m_0 and storing mass at any time instant in
    # 't' as an array 'm':
    self.m_0 = 1.1

    if self.Const_thrust == True :
      self.m = self.m_0 - 9*self.t/self.I_sp

    else :
      self.m = self.m_0 + (4.5*self.t**2 - 18*self.t)/self.I_sp



    #defining initial values of vehicle velocity 'v' and vehicle horizontal
    # angle 'theta' in their arrays:
    self.v[0] = v_0
    self.theta[0] = theta_0

    # define a vector array 'u' with 'v' and 'theta' for applying Heun's method :
    self.u = np.array([self.v, self.theta])


    # We can visualise drag due to wind in vehicle frame. While doing that we
    # neglect the effect of drag due to component of wind velocity
    # perpendicular to vehicle (lift) as C_l = 0. So we define relative
    # velocity with respect to vehicle :
    self.v_rel = self.u[0]-(self.v_w[0]*np.cos(self.u[1]) + self.v_w[1]* \
                            np.sin(self.u[1]))


    # define a vector array 'Dudt' as :
    self.Dudt = np.array([self.F/self.m - \
                          0.5*self.rho*self.v_rel*np.abs(self.v_rel)* \
                          self.A*self.C_d/(self.m) \
                          - self.g*np.sin(self.u[1]),\
                          -self.g*np.cos(self.u[1])/self.u[0]])


  # define a function/method DUdt that takes a vector array u and index i to
  # return a 2 by m array where u is a 2 by m array.
  # This is done for convenience in implementing Heun's approximation for Dudt
  def DUdt(self, u, i):
    f = np.array([self.F[i]/self.m[i] - 0.5*self.rho*self.v_rel[i-1]* \
                  np.abs(self.v_rel[i-1])*self.A*self.C_d/(self.m[i]) \
                  - self.g*np.sin(u[1]), \
                  -self.g*np.cos(u[1])/u[0]])
```

```
    return (f)



# define a function/method Huen_s_Iteration that takes a vector array u to return a
# 2 by m array where u is a 2 by m array.
# This is done for convenience in implementing Heun's approximation for Dudt
def Huen_s_Iteration(self):
  # employing Heun's iteration algorithm in a for loop:

  self.t = np.linspace(0, self.n, 100*self.n+1)

  for i in range(0,len(self.t)-1):

    # Since thrust force 'F' vanishes and mass 'm' remains constant after burn
    # time of 2s, we define 'F' values after 2s and onwards as 0 and 'm' values
    # after 2s onwards as constant:
    if self.t[i] >= self.t_b :

      self.F[i] = 0
      self.F[i+1] = self.F[i]
      self.m[i] = self.m[i-1]
      self.m[i+1] = self.m[i]

      #end of if condition

    # separate implementation for the first iteration since most quantities
    # like v_rel, u, Dudt needs to be initialised:
    if i == 0 :
      self.Dudt[:,i] = np.array([self.F[i]/self.m[i] - \
                                 0.5*self.rho*self.v_rel[i]* \
                                 np.abs(self.v_rel[i])*self.A*\
                                 self.C_d/(self.m[i])\
                                - self.g*np.sin(self.u[1,i]),\
                                  0])

      # defining a 2 by 1 array u* = ui + dt(Du_by_dt i ) to approximate
      # suceeding point:
      self.u_star = self.u[:, i] + self.dt*(self.Dudt[:, i])


      self.v_rel[i] = self.u[0,i]-(self.v_w[0]*np.cos(self.u[1,i]) + \
                                   self.v_w[1]*np.sin(self.u[1,i]))

      # defining a 2 by 1 array Dudt_star for Heun's approximation of Dudt:
      self.Dudt_star = np.zeros_like(self.u_star)
      self.Dudt_star = self.DUdt(self.u_star, i+1)
      self.Dudt_star[1] = 0

      # using Heun's method to approximate the succeeding values for 'u' :
      self.u[:, i+1] = self.u[:, i] + 0.5*self.dt*(self.Dudt[:, i] + \
                                                   self.Dudt_star)

      #end of if condition

    elif self.u[0,i] == 0 :

      # To avoid overflow error due to dividing with zero velocity, we
      # define that the angular acceleration term is zero when velocity is
      # zero:
      self.Dudt[:,i] = np.array([self.F[i]/self.m[i] - \
                                 0.5*self.rho*self.v_rel[i]* \
                                 np.abs(self.v_rel[i])* \
                                 self.A*self.C_d/(self.m[i])\
                           - self.g*np.sin(self.u[1,i]), 0])

      # defining a 2 by 1 array u* = ui + dt(Du_by_dt i ) to approximate
      # suceeding point:
      self.u_star = self.u[:, i] + self.dt*(self.Dudt[:, i])

      self.v_rel[i] = self.u[0,i]-(self.v_w[0]*np.cos(self.u[1,i]) +\
                                   self.v_w[1]*np.sin(self.u[1,i]))

      # defining a 2 by 1 array Dudt_star for Heun's approximation of Dudt:
      self.Dudt_star = np.zeros_like(self.u_star)
      self.Dudt_star = self.DUdt(self.u_star, i+1)

      # setting angular acceleration term zero when velocity is zero
      self.Dudt_star[1,i] = 0

      # using Heun's method to approximate the succeeding values for 'u' :
      self.u[:, i+1] = self.u[:, i] + 0.5*self.dt*(self.Dudt[:, i] +\
```

```
                                                    self.Dudt_star)
      #end of elif condition



    # Regular implementation of Heun's approximation for non-zero
    # values of velocity u[0,i]
    else :

      self.Dudt[:,i] = np.array([self.F[i]/self.m[i] - 0.5*self.rho*\
                                 self.v_rel[i]*np.abs(self.v_rel[i])*self.A*\
                                 self.C_d/(self.m[i])\
                           - self.g*np.sin(self.u[1,i]),\
                            -self.g*np.cos(self.u[1,i])/self.u[0,i]])

      # defining a 2 by 1 array u* = ui + dt(Du_by_dt i ) to approximate
      # suceeding point:
      self.u_star = self.u[:, i] + self.dt*(self.Dudt[:, i])

      self.v_rel[i] = self.u[0,i]-(self.v_w[0]*np.cos(self.u[1,i]) + \
                                   self.v_w[1]*np.sin(self.u[1,i]))


      # defining a 2 by 1 array Dudt_star for Heun's approximation of Dudt:
      self.Dudt_star = np.zeros_like(self.u_star)
      self.Dudt_star = self.DUdt(self.u_star, i+1)

      # using Heun's method to approximate the succeeding values for 'u' :
      self.u[:, i+1] = self.u[:, i] + 0.5*self.dt*(self.Dudt[:, i] + \
                                                   self.Dudt_star)

      #end of else condition



    # updating values of v_rel in each iteration of for loop:
    self.v_rel[i+1] = self.u[0,i+1]-(self.v_w[0]*np.cos(self.u[1,i+1]) + \
                                     self.v_w[1]*np.sin(self.u[1,i+1]))

    #end of for loop

  # Since iteration runs only for 100n - 1 times, last values of mass (m)
  # and force (F) arrays needs to be updated as below after loop completes:
  self.m[-1]= self.m[-2]
  self.F[-1]= self.F[-2]
  self.Dudt[-1] = self.Dudt[-2]



  # performing integration of vertical velocity component to find height:
  self.h = np.zeros_like(self.t)

  for i in range(0,100*self.n-1):

    sum = self.u[0,i]*np.sin(self.u[1,i])*self.dt
    self.h[i+1] = self.h[i] + sum


  # Since our code assumes that vehicle continues to be in motion even at
  # negative height, we truncate the height solution array to only values
  # greater than 0
  self.h>0
  self.tt = self.t[self.h>0]
  self.h_max = self.h[self.h>0]



  # performing integration of horizontal velocity component to find horizontal
  # displacement (range):
  self.r = np.zeros_like(self.t)

  for i in range(0,100*self.n-1):

    sum = self.u[0,i]*np.cos(self.u[1,i])*self.dt
    self.r[i+1] = self.r[i] + sum

  # Again, since we need only values of range r and u for the time duration
  # when h is greater than 0, we truncate arrays r and u as before and plot
  # it vs time. :
  self.rr = self.r[self.h>0]

  self.uu = np.zeros([2, len(self.rr)])
```

```
    self.uu[:] = self.u[:, self.h>0]




    # defining acceleration and its tangential and centripetal components :
    self.accn = np.zeros_like(self.rr)
    self.a_c = np.zeros_like(self.rr)
    self.a_t = np.zeros_like(self.rr)
    self.a_t = self.Dudt[0, self.h>0]
    self.a_c = self.uu[0]*self.Dudt[1, self.h>0]
    self.accn = np.sqrt(self.a_t**2 + self.a_c**2)


    # Method returns the solution for v and theta as a 2 by n vector where n is
    # the number of time instances used in numerical solution for which height
    # is greater than or equal to 0
    return self.uu

  def plot_m_vs_t (self, label = ''):
  #plotting mass vs time for verification:
    plt.plot(self.t, self.m, label = label)
    plt.xlabel("t (s)")
    plt.ylabel("m (pounds)")
    plt.grid()
    plt.title("m vs t for theta_0 = {} degrees"\
              .format(int(180*self.theta[0]/pi)))

  def plot_v_vs_t (self, label = ''):
  # plotting velocity magnitude vs time :
    plt.plot(self.tt, self.uu[0,:], label = label)
    plt.xlabel("t (s)")
    plt.ylabel("v (ft/s)")
    plt.grid()
    plt.title("v vs t for theta_0 = {} degrees"\
              .format(int(180*self.theta[0]/pi)))

  def plot_v_x_vs_t (self, label = ''):
  # plotting horizontal velocity magnitude vs time :
    plt.plot(self.tt,self.uu[0,:]*np.cos(self.uu[1,:]), label = label)
    plt.xlabel("t (s)")
    plt.ylabel("v_x (ft/s)")
    plt.grid()
    plt.title("v_x vs t for theta_0 = {} degrees"\
              .format(int(180*self.theta[0]/pi)))

  def plot_v_y_vs_t (self, label = ''):
  # plotting vertical velocity magnitude vs time :
    plt.plot(self.tt,self.uu[0,:]*np.sin(self.uu[1,:]), label = label)
    plt.xlabel("t (s)")
    plt.ylabel("v_y  (ft/s)")
    plt.grid()
    plt.title("v_y vs t for theta_0 = {} degrees"\
              .format(int(180*self.theta[0]/pi)))

  def plot_theta_vs_t (self, label = ''):
  # plotting theta vs time :
    plt.plot(self.tt, 180*self.uu[1,:]/pi, label = label)
    plt.xlabel("t (s)")
    plt.ylabel("theta (degrees)")
    plt.grid()
    plt.title("theta vs t for theta_0 = {} degrees"\
              .format(int(180*self.theta[0]/pi)))


  def v_max (self):
  # for finding maximum veocity :
    return print("v_max = {} ft/s".format(np.max(self.uu[0])))

  def height (self) :
  # for finding total height :
    return print("total height = {} ft".format(np.max(self.h_max) \
                                              + self.h_launchrod))

  def plot_h_vs_t (self, label = ''):
  # plotting theta vs time :
    plt.plot(self.tt,self.h_max + self.h_launchrod, label = label)
    plt.xlabel("t (s)")
    plt.ylabel("h (ft)")
    plt.grid()
    plt.title("vertical height vs t for theta_0 = {} degrees"\
```

```
                .format(int(180*self.theta[0]/pi)))




    def h_bo (self) :
    # for finding burnout height:
      self.h_bo = self.h[self.t >= self.t_b]
      return print("burnout height = {} ft".format(self.h_bo[0] + self.h_launchrod))

    def range (self) :

      return print("range = {} ft".format(self.rr[-1]))

    def plot_r_vs_t (self, label = '') :
    # plotting Range vs time :
        plt.plot(self.tt, self.rr, label = label)
        plt.xlabel("t (s)")
        plt.ylabel("horizontal displacement (ft)")
        plt.grid()
        plt.title("horizontal displacement vs t for theta_0 = {} degrees"\
                  .format(int(180*self.theta[0]/pi)))

    def plot_h_vs_r (self, label = '') :
    # plotting Altitude vs Range :
        plt.plot(self.rr, self.h_max, label = label )
        plt.xlabel("Range (ft)")
        plt.ylabel("Altitude (ft)")
        plt.grid()
        plt.title("Altitude vs Range for theta_0 = {} degrees"\
                  .format(int(180*self.theta[0]/pi)))

    def plot_accn_vs_t (self, label = '') :
    # plotting vehicle acceleration magnitude vs Range :
        plt.plot(self.tt, self.accn, label = label )
        plt.xlabel("t (s)")
        plt.ylabel("Acceleration (ft/s^2)")
        plt.grid()
        plt.title("Acceleration magnitude vs t for theta_0 = {} degrees"\
                  .format(int(180*self.theta[0]/pi)))

    def plot_a_c_vs_t (self, label = '') :
    # plotting vehicle acceleration magnitude vs Range :
        plt.plot(self.tt, self.a_c, label = label )
        plt.xlabel("t (s)")
        plt.ylabel("Centripetal acceleration (a_c) (ft/s^2)")
        plt.grid()
        plt.title("Centripetal acceleration (a_c) vs t for theta_0 = {} degrees"\
                  .format(int(180*self.theta[0]/pi)))

    def plot_a_t_vs_t (self, label = '') :
    # plotting vehicle acceleration magnitude vs Range :
        plt.plot(self.tt, self.a_t, label = label )
        plt.xlabel("t (s)")
        plt.ylabel("Tangential acceleration (a_t) (ft/s^2)")
        plt.grid()
class visuallyCompare(object):
    """ Class that has a list of trajectory solution objects as attributes, may
        be from the time of its constuction, or from later; and does something
        with them

        In this case it takes three trajectory solutions, and plots the
        Altitude vs range on the same axes for comparison.
    """
    def __init__(self,A):

        self.A = []
        for i in range(0, len(A)):
          self.A.append(A[i])


    def compare(self):

        A = self.A
        for i in range(0, len(self.A)):
          self.plot(self.A[i])
          axes = plt.gca()
          plt.grid()
          plt.legend(bbox_to_anchor = (1.03, 0.5));
```

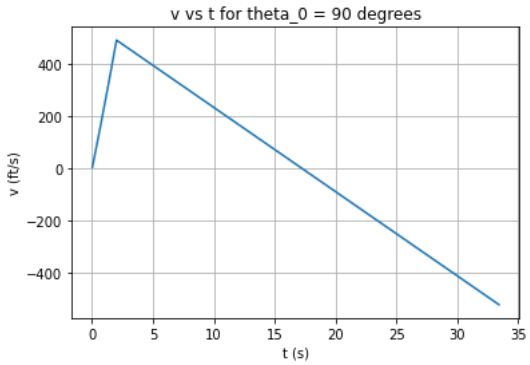
```
    def plot(self, B, axes=None):

          self.B = B
          R = self.B.rr
          H = self.B.h_max
          if axes is None:
              axes = plt.gca()
          axes.plot(R,H, label = \
                    "theta_0 = {}".format(int(180*self.B.theta[0]/pi)))
          axes.set_xlabel('Range (ft)');  axes.set_ylabel('Altitude (ft)')
          axes.set_aspect('equal', 'box')
          axes.set_title( \
                    'Comparing Altitude vs Range plots for different theta_0')
```

c) Verify the algorithm by comparing the altitude obtained with the vertical trajectory solution discussed in class by setting CD = 0 and F = 9 constant, with t_b = 2 s. Tabulate your burnout and total heights using both techniques.

> We use our code to create an object with given conditions.

```
Trajectory_1 = Trajectory_sol( n = 35, t_b = 2, C_d = 0, C_L = 0, Const_thrust = True,\
                              Const_thrust_value = 9, v_0 = 0, theta_0 = pi/2, \
                              v_w = np.array([0,0]), h_launchrod = 0)
Trajectory_1.Huen_s_Iteration()
Trajectory_1.plot_v_vs_t()
```

```
<ipython-input-2-882eb2d79e42>:90: RuntimeWarning: divide by zero encountered in true_divide
  -self.g*np.cos(self.u[1])/self.u[0]])
```



> We see that the velocity is increasing linearly for constant thrust burn (9 pounds) and constant g for $t < t_b$ and then linearly decreasing due to constant acceleration after burnout in the absence of drag. We are assuming vertical motion with no crosswind.

Burnout velocity comparison:

> The maximum velocity thus obtained is found out to be 490.503 ft/s

```
Trajectory_1.v_max()
```

```
 v_max = 490.50389576190287 ft/s
```

> On using the vertical trajectory solution in the absence of drag:

$$v_{bo} = -g_e I_{sp} ln(\frac{m_f}{m_o})$$

> gives :

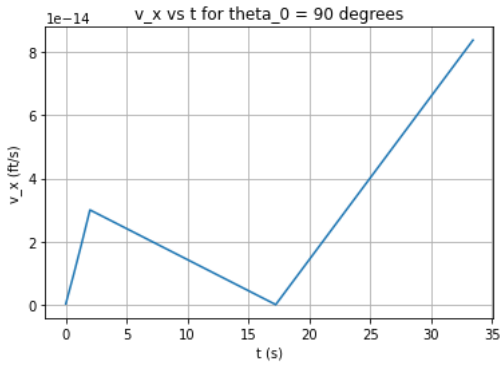
$$v_{bo} = 490.503$$

> which agrees closely with our Heun's method solution with an error of 0 %

Horizontal velocity comparison:

> We plot the horizontal velocity component obtained from our solution vs time.

```
Trajectory_1.plot_v_x_vs_t()
```



> We see above that the horizontal velocity component $vcos\theta$ is more or less constant and close to zero in case of vertical trajectory. (Please note that the scale above is in 10^{-14} so all variations in graph can be assumed negligible and can be attributed to numerical error due to integration)

Integrating velocity for height and range:

> The total height thus obtained is found to be 4216.779 feet.

```
Trajectory_1.height()
```

```
total height = 4216.779750695573 ft
```

> For burnout height, we compute height at burnout time $t_b = 2$s which comes out as 478.39 ft

```
Trajectory_1.h_bo()
```

```
burnout height = 478.3940992842061 ft
```

Comparison of total and burnout heights with analytically solved values :

> On using the vertical trajectory solution in the absence of drag:

$$h_{bo} = -g_e I_{sp}(\frac{m_f}{m_o - m_f} ln(\frac{m_f}{m_o}) + 1) - g_e t_b^2$$

> gives :

$$h_{bo} = 480.84 \text{ ft}$$

> which agrees closely with our Heun's method solution with an error of 0.4 %

> for coasting height

> div align = "center" >$h_c = \frac{1}{2} v_{bo}^2 / g_e$

> gives :

$$h_c = 3735.93 \text{ ft}$$

> adding both give the total height as:

$$h_{max} = 4216.78 \text{ ft}$$

> which agrees closely with our Heun's method solution with an error of 0.07 %

> Similarly, we verify if range is zero for vertical case.

```
Trajectory_1.range()
```

```
range = 9.361685057925715e-13 ft
```

> Again for vertical case, we see above that the horizontal range zero. (Please note that the scale above is in 10^{-13} so all variations in graph can be assumed negligible and can be attributed to numerical error due to integration)

d) Having verified the code, predict and plot the vehicle trajectory (altitude vs. range) for initial launch angles of 70^0, 75^0, and 80^0 from the horizon directed into a 10 mph crosswind. Include drag in the simulation and provide a tabulated listing of vehicle velocity, acceleration, range, and altitude as a function of time for the $\theta = 75^0$ case. Assume a five foot launch rod for your calculations.

> We first run the method with no drag.

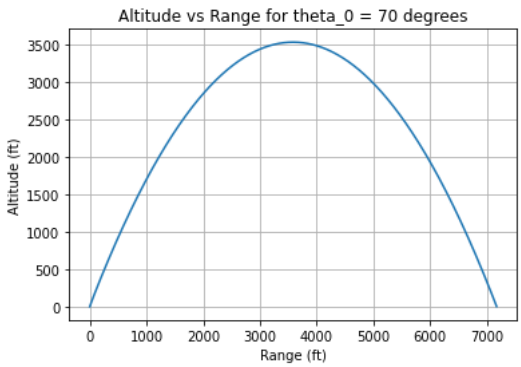
Without drag:

> Case 1: $\theta_0 = 70^0$

> We solve for trajectory using the above class method for $\theta = 70^0, C_d = 0, F = 18 - 9t$, a crosswind of 10 mph $= 14.667$ ft/s against launch direction and a launch rod height of 5 feet.

```
Trajectory_70 = Trajectory_sol( n = 35, t_b = 2, C_d = 0, C_L = 0, Const_thrust = False,\
                               Const_thrust_value = 9, v_0 = 0, theta_0 = 70*pi/180, \
                               v_w = np.array([-14.667,0]), h_launchrod = 5)
Trajectory_70.Huen_s_Iteration()
Trajectory_70.plot_h_vs_r()
```

```
<ipython-input-2-882eb2d79e42>:90: RuntimeWarning: divide by zero encountered in true_divide
  -self.g*np.cos(self.u[1])/self.u[0]])
```

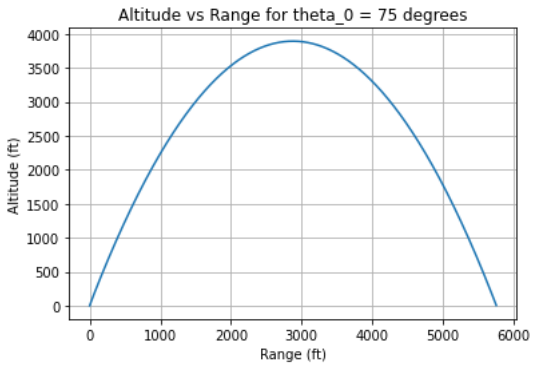


> Case 2: $\theta_0 = 75^0$

> We solve for trajectory using the above class method for $\theta = 75^0, C_d = 0, F = 18 - 9t$, a crosswind of 10 mph $= 14.667$ ft/s against launch direction and a launch rod height of 5 feet.

```
Trajectory_75 = Trajectory_sol( n = 35, t_b = 2, C_d = 0, C_L = 0, Const_thrust = False,\
                               Const_thrust_value = 9, v_0 = 0, theta_0 = 75*pi/180, \
                               v_w = np.array([-14.667,0]), h_launchrod = 5)
Trajectory_75.Huen_s_Iteration()
Trajectory_75.plot_h_vs_r()
```

```
<ipython-input-2-882eb2d79e42>:90: RuntimeWarning: divide by zero encountered in true_divide
  -self.g*np.cos(self.u[1])/self.u[0]])
```

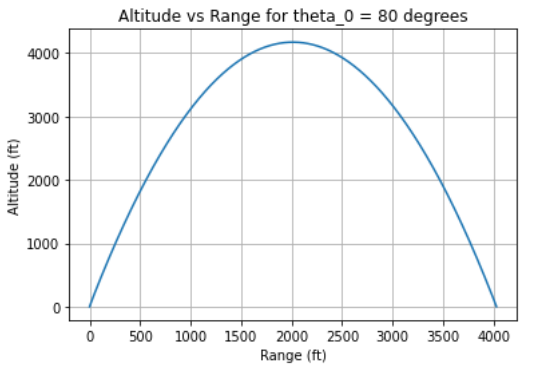


> Case 3: $\theta_0 = 80^0$

> We solve for trajectory using the above class method for $\theta = 80^0$, $C_d = 0, F = 18 - 9t$, a crosswind of 10 mph $= 14.667$ ft/s against launch direction and a launch rod height of 5 feet.

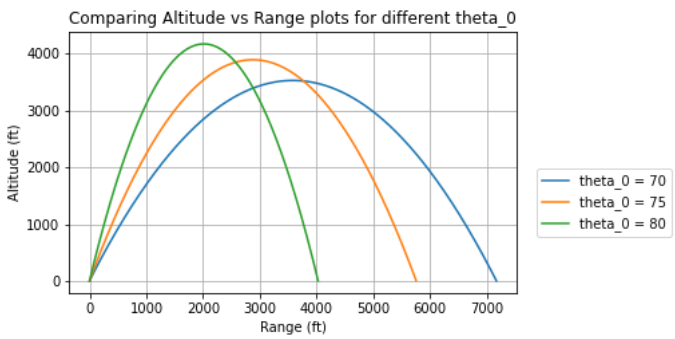
```
Trajectory_80 = Trajectory_sol( n = 35, t_b = 2, C_d = 0, C_L = 0, Const_thrust = False,\
                               Const_thrust_value = 9, v_0 = 0, theta_0 = 80*pi/180, \
                               v_w = np.array([-14.667,0]), h_launchrod = 5)
Trajectory_80.Huen_s_Iteration()
Trajectory_80.plot_h_vs_r()
```

```
<ipython-input-2-882eb2d79e42>:90: RuntimeWarning: divide by zero encountered in true_divide
  -self.g*np.cos(self.u[1])/self.u[0]])
```



> Comparing all 3 cases without considering drag:

```
W_o_drag = [Trajectory_70, Trajectory_75, Trajectory_80]
Comparison = visuallyCompare(W_o_drag)
Comparison.compare()
```



> We see that as initial launch angle increases, the range decreases and the maximum altitude increases.

Introduction of drag with $C_d = 0.3$:

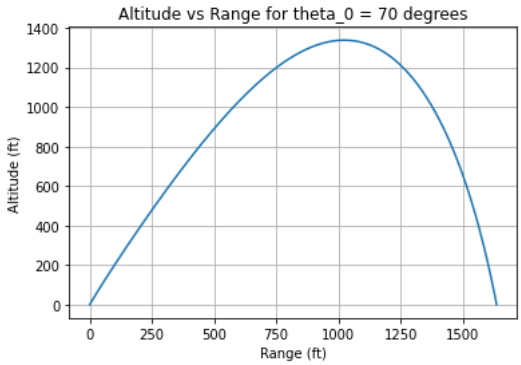
> For the above 3 cases with drag coefficient C_d set to 0.3.

> Case 1: $\theta_0 = 70^0$

> We solve for trajectory using the above class method for $\theta = 70^0$, $C_d = 0.3$, $F = 18 - 9t$, a crosswind of 10 mph $= 14.667$ ft/s against launch direction and a launch rod height of 5 feet.

```
Trajectory_70_drag = Trajectory_sol( n = 35, t_b = 2, C_d = 0.3, C_L = 0, Const_thrust = False,\
                              Const_thrust_value = 9, v_0 = 0, theta_0 = 70*pi/180, \
                              v_w = np.array([-14.667,0]), h_launchrod = 5)
Trajectory_70_drag.Huen_s_Iteration()
Trajectory_70_drag.plot_h_vs_r()
```

```
<ipython-input-2-882eb2d79e42>:90: RuntimeWarning: divide by zero encountered in true_divide
  -self.g*np.cos(self.u[1])/self.u[0]])
```

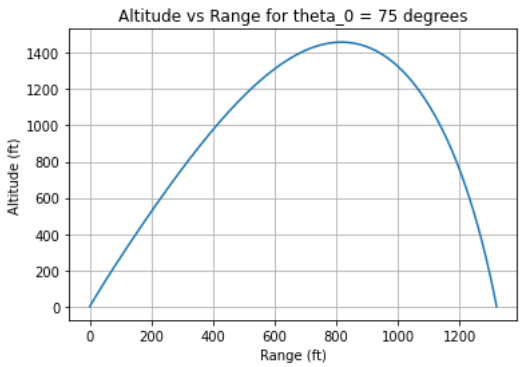


> Case 3: $\theta_0 = 75^0$

> We solve for trajectory using the above class method for $\theta = 75^0$, $C_d = 0.3$, $F = 18 - 9t$, a crosswind of 10 mph $= 14.667$ ft/s against launch direction and a launch rod height of 5 feet.

```
Trajectory_75_drag = Trajectory_sol( n = 35, t_b = 2, C_d = 0.3, C_L = 0, Const_thrust = False,\
                              Const_thrust_value = 9, v_0 = 0, theta_0 = 75*pi/180, \
                              v_w = np.array([-14.667,0]), h_launchrod = 5)
Trajectory_75_drag.Huen_s_Iteration()
Trajectory_75_drag.plot_h_vs_r()
```

```
<ipython-input-2-882eb2d79e42>:90: RuntimeWarning: divide by zero encountered in true_divide
  -self.g*np.cos(self.u[1])/self.u[0]])
```

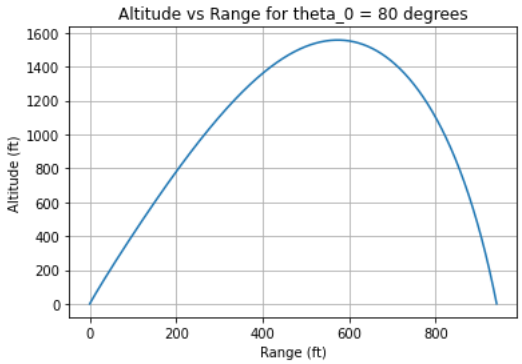


> Case 3: $\theta_0 = 80^0$

> We solve for trajectory using the above class method for $\theta = 80^0$, $C_d = 0.3$, $F = 18 - 9t$, a crosswind of 10 mph $= 14.667$ ft/s against launch direction and a launch rod height of 5 feet.

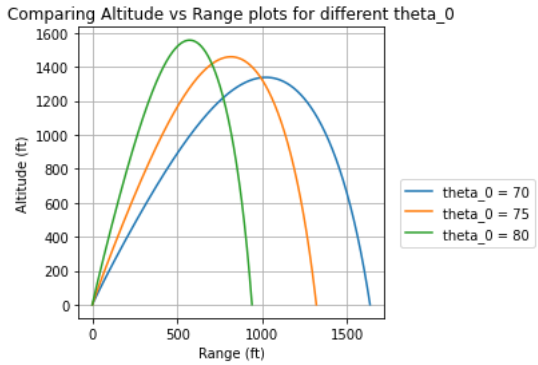
```
Trajectory_80_drag = Trajectory_sol( n = 35, t_b = 2, C_d = 0.3, C_L = 0, Const_thrust = False,\
                              Const_thrust_value = 9, v_0 = 0, theta_0 = 80*pi/180, \
                              v_w = np.array([-14.667,0]), h_launchrod = 5)
Trajectory_80_drag.Huen_s_Iteration()
Trajectory_80_drag.plot_h_vs_r()
```

```
<ipython-input-2-882eb2d79e42>:90: RuntimeWarning: divide by zero encountered in true_divide
  -self.g*np.cos(self.u[1])/self.u[0]])
```



> Comparing all 3 cases after considering drag:

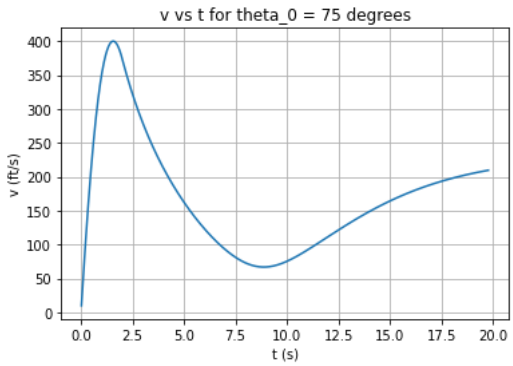
```
With_drag = [Trajectory_70_drag, Trajectory_75_drag, Trajectory_80_drag]
Comparison = visuallyCompare(With_drag)
Comparison.compare()
```



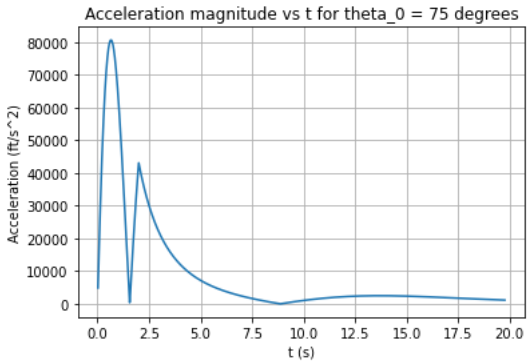
Plots and tabulation of data for $\theta = 75^0$ case with drag:

> The plots of v, a, a_c, a_t, θ, H and R as a function of time are given as follows:

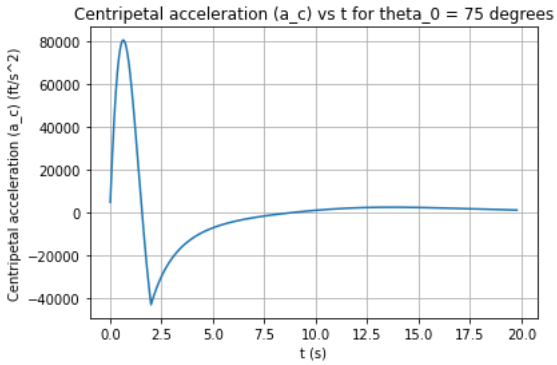
```
Trajectory_75_drag.plot_v_vs_t()
```



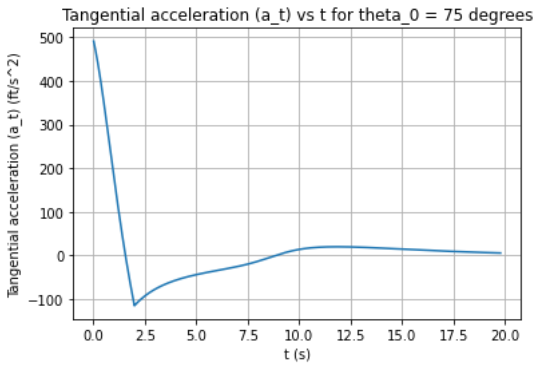
```
Trajectory_75_drag.plot_accn_vs_t()
```



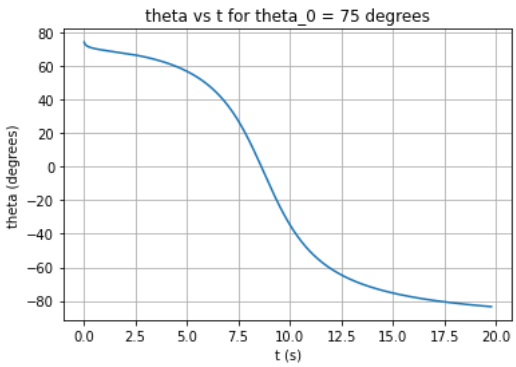

```
Trajectory_75_drag.plot_a_c_vs_t()
```



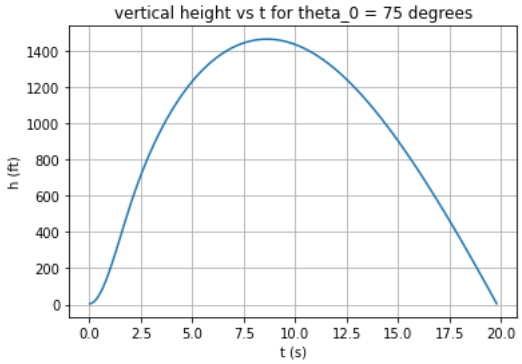

```
Trajectory_75_drag.plot_a_t_vs_t()
```



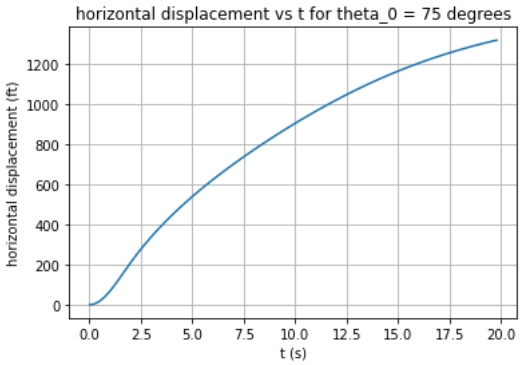

```
Trajectory_75_drag.plot_theta_vs_t()
```



```
Trajectory_75_drag.plot_h_vs_t()
```




```
Trajectory_75_drag.plot_r_vs_t()
```



> The above quantities are tabulated at specific intervals of time for 25 instances as shown below:

```
number = np.linspace(1, 26, 26)

tdata = np.zeros(26)
vdata = np.zeros(26)
adata = np.zeros(26)
acdata = np.zeros(26)
atdata = np.zeros(26)
thetadata = np.zeros(26)
hdata = np.zeros(26)
rdata = np.zeros(26)

for i in range(0,26):
  tdata[i] = Trajectory_75_drag.tt[round(79.04*i)]
  vdata[i] = Trajectory_75_drag.uu[0, round(79.04*i)]
  adata[i] = Trajectory_75_drag.accn[round(79.04*i)]
  acdata[i] = Trajectory_75_drag.a_c[round(79.04*i)]
  atdata[i] = Trajectory_75_drag.a_t[round(79.04*i)]
  thetadata[i] = Trajectory_75_drag.uu[1, round(79.04*i)]
  hdata[i] = Trajectory_75_drag.h_max[round(79.04*i)]
  rdata[i] = Trajectory_75_drag.rr[round(79.04*i)]


p = [number, tdata, \
      vdata,  \
      adata, \
      acdata, \
      atdata, \
      thetadata, \
      hdata, \
      rdata]

data = np.transpose(p)

head =["No.", "time (s)", "v(ft/s)", "a(ft/s^2)", "a_c(ft/s^2)", "a_t(ft/s^2)", \
       "theta (degrees)", "H(ft)", "R(ft)"]

print(tabulate(data, headers = head, tablefmt = "simple", maxcolwidths= 0.1))
```

No.	time (s)	v(ft/s)	a(ft/s^2)	a_c(ft/s^2)	a_t(ft/s^2)	theta (degrees)	H(ft)	R(ft)
1	0.02	9.87402	4878.66	4853.83	491.576	1.29609	0.0477883	0.0128048
2	0.81	309.449	76801.5	76801.1	248.186	1.21733	129.642	45.8317
3	1.6	399.765	4226.91	-4226.9	-10.5735	1.19272	403.397	150.91
4	2.39	332.466	32442	-32441.9	-97.5796	1.16655	675.303	262.676
5	3.18	265.726	19422	-19421.8	-73.0896	1.13142	890.182	358.649
6	3.97	214.466	12345.4	-12345.2	-57.5625	1.08373	1059.56	442.746
7	4.76	173.391	8130.82	-8130.69	-46.8922	1.01818	1192.46	518.217
8	5.55	139.616	5421.02	-5420.88	-38.8271	0.926079	1294.69	587.224
9	6.34	111.72	3552.02	-3551.87	-31.7927	0.793286	1370.16	651.264
10	7.13	89.4762	2171.35	-2171.21	-24.2658	0.59817	1421.51	711.391
11	7.92	73.9226	1081.55	-1081.45	-14.6294	0.317187	1450.55	768.322
12	8.71	67.0237	173.478	-173.459	-2.58802	-0.0427401	1458.58	822.51
13	9.5	69.6234	602.875	602.813	8.6582	-0.407464	1446.59	874.215
14	10.3	79.7442	1264.72	1264.62	15.8584	-0.701204	1415.02	924.17
15	11.09	93.7425	1784.72	1784.62	19.0375	-0.904716	1365.6	971.139
16	11.88	109.202	2160.09	2160	19.7798	-1.04513	1299.17	1015.69
17	12.67	124.661	2389.23	2389.16	19.1653	-1.14456	1217.01	1057.72
18	13.46	139.316	2483.36	2483.3	17.8249	-1.2176	1120.5	1097.13
19	14.25	152.754	2463.69	2463.64	16.1282	-1.27316	1011.13	1133.88
20	15.04	164.788	2356.71	2356.67	14.3013	-1.31669	890.395	1167.95
21	15.83	175.375	2189.61	2189.57	12.4851	-1.35168	759.723	1199.38
22	16.62	184.56	1986.92	1986.89	10.7655	-1.38036	620.471	1228.24
23	17.41	192.441	1768.73	1768.7	9.19091	-1.40427	473.875	1254.65
24	18.2	199.142	1550.1	1550.08	7.78381	-1.42448	321.047	1278.72
25	18.99	204.799	1341.37	1341.35	6.54961	-1.44174	162.964	1300.6
26	19.78	209.546	1148.87	1148.86	5.4826	-1.45661	0.473992	1320.44

Comparison of Huen's Iteration with inbuilt routines

Case 1: For vertical case solved for part c)

We use the integrate.solve_ivp() method to integrate the RHS of the differential equation:

$$\frac{dv}{dt} = -\frac{Fg_e}{m} - \frac{1}{2}\frac{\rho v_{rel}^2 A C_D}{m} - g_e sin(\theta)$$

$$\frac{d\theta}{dt} = \frac{1}{2}\frac{\rho v_{rel}^2 A C_L}{m} - \frac{g_e cos(\theta)}{v}$$

where F is the force given in pounds (and thus need to be multiplied with g_e) and v_{rel} is the component of vehicle velocity relative to wind along the direction of vehicle motion given as:

$$v_{rel} = v - v_{w,x} cos(\theta) - v_{w,y} sin(\theta)$$

where $v_{w,x}$, $v_{w,y}$ are the positive x and y components of wind velocity.

Putting C_L = 0, C_d = 0 gives :

$$\frac{dv}{dt} = -\frac{Fg_e}{m} - g_e sin(\theta)$$

$$\frac{d\theta}{dt} = -\frac{g_e cos(\theta)}{v}$$

We use a vector u to hold both v and theta as $u = [v, \theta]$

We thus define the RHS of the above system in terms of u as shown:

$$\frac{du}{dt} = [-\frac{Fg_e}{m} - g_e sin(\theta), -\frac{g_e cos(\theta)}{v}]$$

This is defined as a function below to integrate via solve_ivp() routine.

We are using Runge-Kutta approximation to solve as specified by method = 'RK45' in the solve_ivp() routine.

Initial conditions are:

$$v_0 = 0, \theta_0 = 90^0$$

On solving, the routine ouputs the values of input u at time values specified in t_eval as an array.

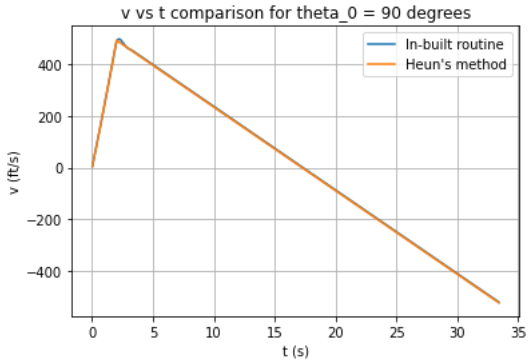
```
def RHS_Du_by_dt_1(t,u):
  g = Trajectory_1.g
  u=u
  if t <= 2:
    F = 9*g
    m = 1.1 - 9*t/165
  else :
    F = 0
    m = 1.1 - 9*2/165

  I_sp = Trajectory_1.I_sp
  C_d = Trajectory_1.C_d    #  0 since we kept C_d = 0
  C_L = Trajectory_1.C_L    #  0 since we kept C_L = 0
  D_ref = Trajectory_1.D_ref
  rho = Trajectory_1.rho
  v_rel = u[0]
  A = (pi*D_ref**2)/4

  Dvdt = F/m - 0.5*rho*v_rel*np.abs(v_rel)*A*C_d/m\
                   - g*np.sin(u[1])

  D_thetadt = -g*np.cos(u[1])/u[0]

  if u[0] == 0:
    return np.array([Dvdt, 0])
  else:
    return np.array([Dvdt, D_thetadt])



sol = solve_ivp(RHS_Du_by_dt_1, t_span = np.array([0, Trajectory_1.tt[-1]]), y0 \
                = [0, pi/2], method = 'RK45', t_eval =  Trajectory_1.tt)
```

```
<ipython-input-26-2f6d147e149c>:22: RuntimeWarning: divide by zero encountered in double_scalars
  D_thetadt = -g*np.cos(u[1])/u[0]
```

> We compare both solutions by plotting v vs t and θ vs t plots.

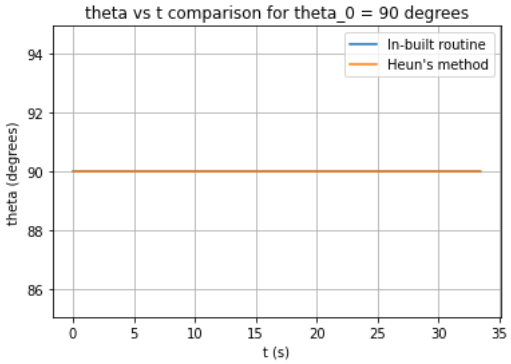
```
plt.plot(sol.t, sol.y[0], label = "In-built routine")
Trajectory_1.plot_v_vs_t(label = "Heun's method")
plt.legend()
plt.title('v vs t comparison for theta_0 = 90 degrees' )
```

```
Text(0.5, 1.0, 'v vs t comparison for theta_0 = 90 degrees')
```



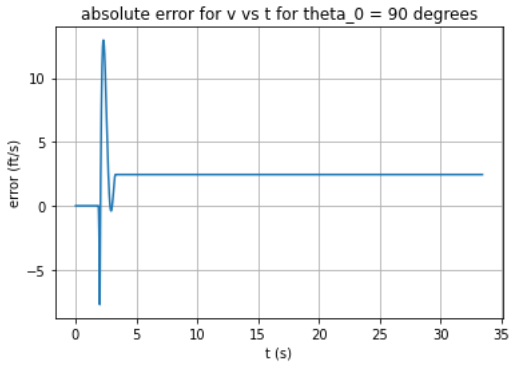
```
plt.plot(sol.t, 180*sol.y[1]/pi, label = "In-built routine")
Trajectory_1.plot_theta_vs_t(label = "Heun's method")
plt.legend()
plt.title('theta vs t comparison for theta_0 = 90 degrees')
```

```
Text(0.5, 1.0, 'theta vs t comparison for theta_0 = 90 degrees')
```



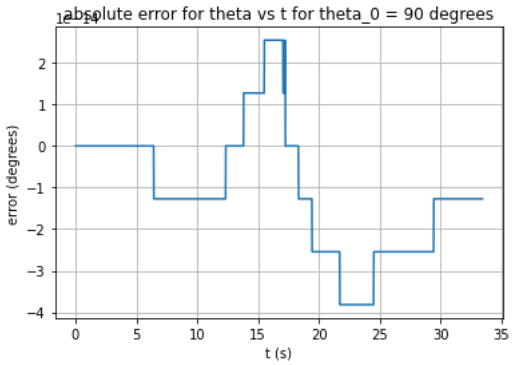

```
error_v = (sol.y[0] - Trajectory_1.uu[0])

plt.plot(sol.t, error_v)
plt.xlabel("t (s)")
plt.ylabel("error (ft/s)")
plt.grid()
plt.title('absolute error for v vs t for theta_0 = 90 degrees')
```

```
Text(0.5, 1.0, 'absolute error for v vs t for theta_0 = 90 degrees')
```




```
error_theta = 180*(sol.y[1] - Trajectory_1.uu[1])/pi

plt.plot(sol.t, error_theta)
plt.xlabel("t (s)")
plt.ylabel("error (degrees)")
plt.grid()
plt.title('absolute error for theta vs t for theta_0 = 90 degrees')
```

```
Text(0.5, 1.0, 'absolute error for theta vs t for theta_0 = 90 degrees')
```



> Note that in the theta vs t comparison the scale on y axis is 10^{-14}. Thus we can see that the error in theta is negligible.

> For the v vs t plot, the maximum error is about 10ft/s which can be attributed to numerical error due to Huen's method.

Case 2: For vertical case solved for part d)

Again RHS of the differential equation in this case is:

$$\frac{dv}{dt} = -\frac{Fg_e}{m} - \frac{1}{2}\frac{\rho v_{rel}^2 A C_D}{m} - g_e sin(\theta)$$

$$\frac{d\theta}{dt} = -\frac{g_e cos(\theta)}{v}$$

$$v_{rel} = v - v_{w,x} cos(\theta) - v_{w,y} sin(\theta)$$

Here, due to crosswind, $v_{w,x}$ = -14.667$ ft/s

Again, we use vector u to hold both v and theta as $u = [v, \theta]$

We thus define the RHS of the above system in terms of u as shown:

$$\frac{du}{dt} = [-\frac{Fg_e}{m} - \frac{1}{2}\frac{\rho v_{rel}^2 A C_D}{m} - g_e sin(\theta), -\frac{g_e cos(\theta)}{v}]$$

This is defined as a function below to integrate via solve_ivp() routine.

We are using Runge-Kutta approximation to solve as specified by method = 'RK45' in the solve_ivp() routine.

On solving, the routine ouputs the values of input u at time values specified in t_eval as an array.

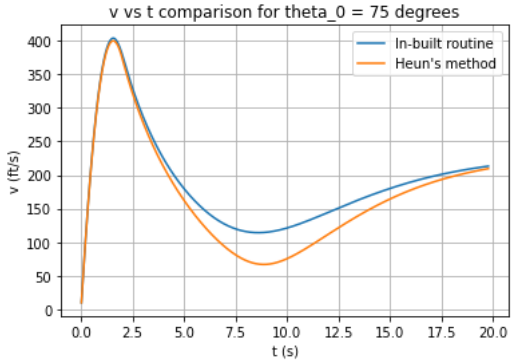
```
def RHS_Du_by_dt_2(t,u):

  t = t
  g = 32.2

  if t <= 2:
     F = (18-9*t)*g
     m =  1.1 + ((4.5*(t**2))-(18*t))/165

  else :
    F = 0
    m = 1.1 - 18/165

  I_sp = Trajectory_75_drag.I_sp
  C_d = Trajectory_75_drag.C_d    #  0.3 since we kept C_d = 0.3
  C_L = Trajectory_75_drag.C_L    #  0 since we kept C_L = 0
  D_ref = Trajectory_75_drag.D_ref
  rho = Trajectory_75_drag.rho
  v_rel = u[0] + 14.667*np.cos(u[1])
  A = (pi*D_ref**2)/4

  Dvdt = F/m - 0.5*rho*v_rel*np.abs(v_rel)*A*C_d/m\
                    - g*np.sin(u[1])

  D_thetadt = -g*np.cos(u[1])/u[0]


  if u[0] == 0:
    return np.array([Dvdt, 0])
  else:
    return np.array([Dvdt, D_thetadt])


sol_2 = solve_ivp(RHS_Du_by_dt_2, \
                  t_span = np.array([0, Trajectory_75_drag.tt[-1]]),\
                  y0 = [0, 75*pi/180], method = 'RK45', \
                  t_eval =  Trajectory_75_drag.tt)
```

```
    <ipython-input-32-c9cdb73eebb6>:25: RuntimeWarning: divide by zero encountered in double_scalars
      D_thetadt = -g*np.cos(u[1])/u[0]
```

> Comparing values:

```
plt.plot(sol_2.t, sol_2.y[0], label = "In-built routine")
Trajectory_75_drag.plot_v_vs_t(label = "Heun's method")
plt.legend()
plt.title('v vs t comparison for theta_0 = 75 degrees' )
```

```
Text(0.5, 1.0, 'v vs t comparison for theta_0 = 75 degrees')
```

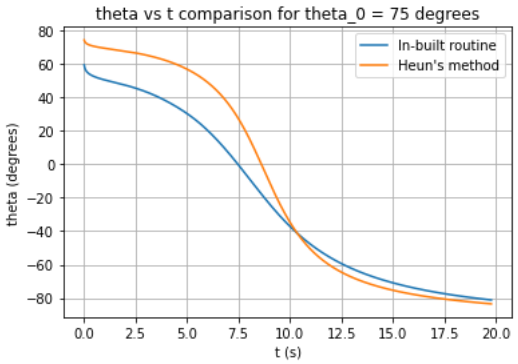



```
print(180*sol_2.y[1]/pi)
```

```
[ 59.35925095  58.58223296  58.01402159 ... -81.11478771 -81.12814557
 -81.14148086]
```

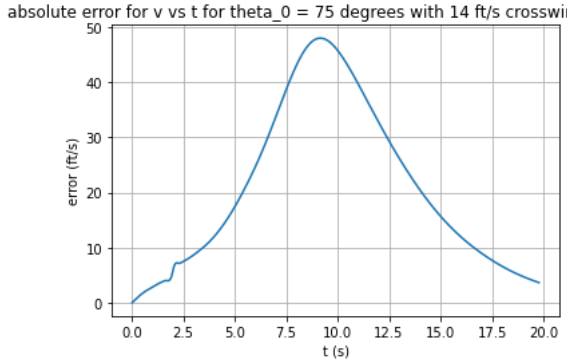
```
plt.plot(sol_2.t, 180*sol_2.y[1]/pi, label = "In-built routine")
Trajectory_75_drag.plot_theta_vs_t(label = "Heun's method")
plt.legend()
plt.title('theta vs t comparison for theta_0 = 75 degrees' )
```

```
Text(0.5, 1.0, 'theta vs t comparison for theta_0 = 75 degrees')
```



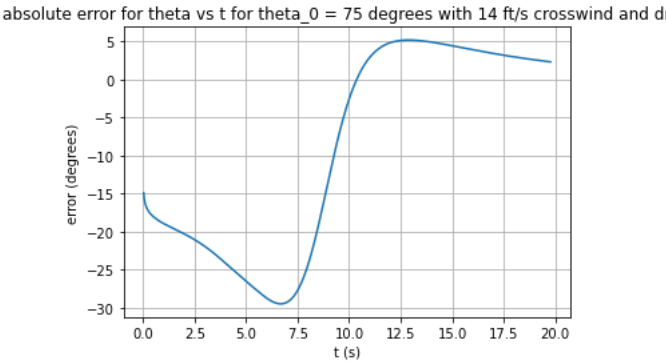

```
error_v_75 = (sol_2.y[0] - Trajectory_75_drag.uu[0])

plt.plot(sol_2.t, error_v_75)
plt.xlabel("t (s)")
plt.ylabel("error (ft/s)")
plt.grid()
plt.title('absolute error for v vs t for theta_0 = 75 degrees with 14 \
ft/s crosswind and drag')
```

```
Text(0.5, 1.0, 'absolute error for v vs t for theta_0 = 75 degrees with 14 ft/s crosswind and drag')
```



```
error_theta_75 = 180*(sol_2.y[1] - Trajectory_75_drag.uu[1])/pi

plt.plot(sol_2.t, error_theta_75)
plt.xlabel("t (s)")
plt.ylabel("error (degrees)")
plt.grid()
plt.title('absolute error for theta vs t for theta_0 = 75 degrees with \
14 ft/s crosswind and drag')
```

```
Text(0.5, 1.0, 'absolute error for theta vs t for theta_0 = 75 degrees with 14 ft/s crosswind and drag')
```



> We observe that the error for both v and θ seem to be maximum at the peak height due to the fact that velocity becomes zero and our Huen's iteration needs to be corrected. We still achieve reasonable accuracy with errors becoming small towards the terminal phase.

Conclusion:

> We can conclude from the above comparison that Huen's iteration method is reasonably accurate in solving the system of differential equations. But one can also see that Huen's iteration might diverge slightly from expected values if we have to introduce a correction during iteration procedure.

> The link to the Google colab listing of the code is given here