PHY 410
Final Assignment

Han Wen

Person No. 50096432

December 11, 2014

Abstract

Final project for PHY 410, 2014 fall semester

Contents

1 Problem 1 **2**
1.1 Description . . . 2
1.2 Solution . . . 2
1.2.1 PART a . . . 2
1.2.2 PART b . . . 2
1.2.3 PART c . . . 2

2 Problem 2 **4**
2.1 Description . . . 4
2.2 Solution . . . 4
2.2.1 PART a . . . 4
2.2.2 PART b . . . 4
2.2.3 PART c . . . 4

3 Problem 3 **7**
3.1 Description . . . 7
3.2 Solution . . . 7
3.2.1 PART a . . . 7
3.2.2 PART b . . . 7
3.2.3 PART c . . . 7

4 Problem 4 **7**
4.1 Description . . . 7
4.2 Solution . . . 10
4.2.1 PART a . . . 10
4.2.2 PART b . . . 10

A Appendix **12**
A.1 python code . . . 12

1 Problem 1

1.1 Description

Problem 1 (25 points). Consider a rocket of total mass m_0, with a mass of m_1 when it is empty of propellant. Assume the rocket expels propellent at a constant exhaust velocity v_e, and the mass linearly decreases with time with rate B until it is spent :

$$m(t) = max(m_1, m_0 - Bt)$$

Consider the Saturn V rocket with $m_0 = 2,970,000kg$ and $m_1 = 130,000kg$ after the first stage, which can apply 34,020 kN of force to move (thrust). The burn time is 165 s. This burn time corresponds to B = 17212 kg/s. `http://en.wikipedia.org/wiki/Saturn_V`

a. (5 points) In the absence of air resistance, calculate the escape velocity off the surface of the earth if a rocket is fired straight up.

b. (10 points) The balle program from Lecture 20 handles projectile motion in a gravitational field, close to the surface of the earth. Modify the balle program from Lecture 20 to account for the changing mass of a rocket, and the full gravitational potential from the earth for arbitrary radius. Plot the distance from the surface of the earth r(t), as well as the velocity v(t), as a function of time for the first stage of the Saturn V rocket. Does the rocket achieve escape velocity like this?

c. (10 points) Now consider air resistance in the same problem. Imagine that the cross-sectional area of the rocket is $25m^2$, and that the density of air (in kg/m^3) is equal to $\rho(h) = 1.2e^{-h/h_0}$

where h is the height off the earths surface in meters, and $h_0 = 10000m$. With the same initial parameters as in (b), compare the results from (b) to the case with air resistance.

1.2 Solution

1.2.1 PART a

The condition for the rocket to just escape will be the total energy is 0, therefore:

$$\frac{1}{2}mv^2 = \frac{GMm}{r} \longrightarrow v = \sqrt{\frac{2GM}{r}} \longrightarrow v = 11180.5m/s$$

The m, M, r, G are mass of rocket, mass of earth, radius of earth, gravitational constant.

1.2.2 PART b

The code is included in the appendix, the result plot is shown below: Fig. 1

The final speed is 4580.3m/s less than the escape velocity.

1.2.3 PART c

With the friction, the result is shown below:Fig. 2

The final speed is 4567.3 m/s . We can see the results are almost the same, that's because when speed is high, the drag coefficient is no longer a constant, and will increase [2]. This program is only to show I can perform the calculation.

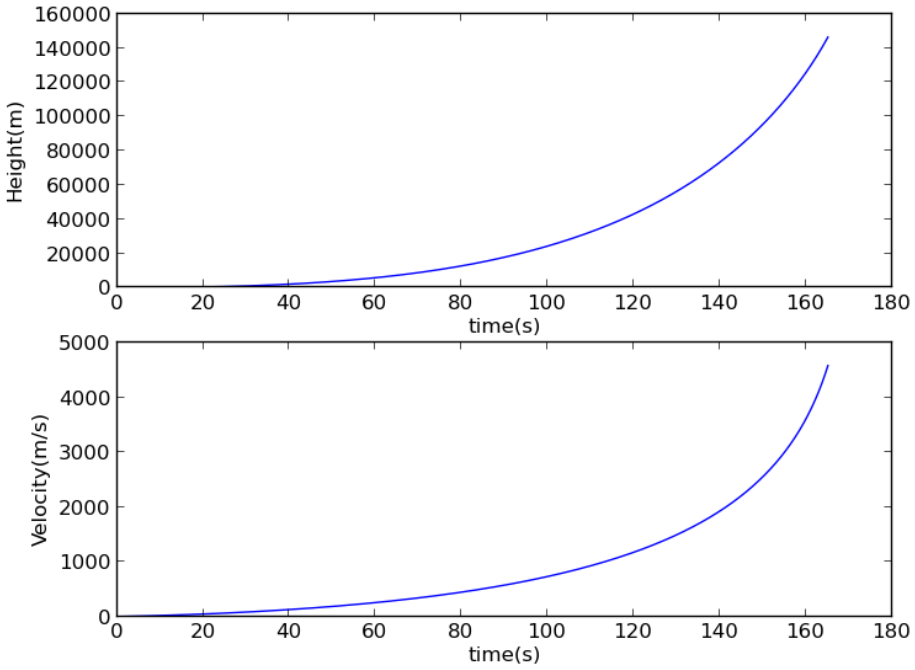

Figure 1: Height and velocity vs. time, no friction

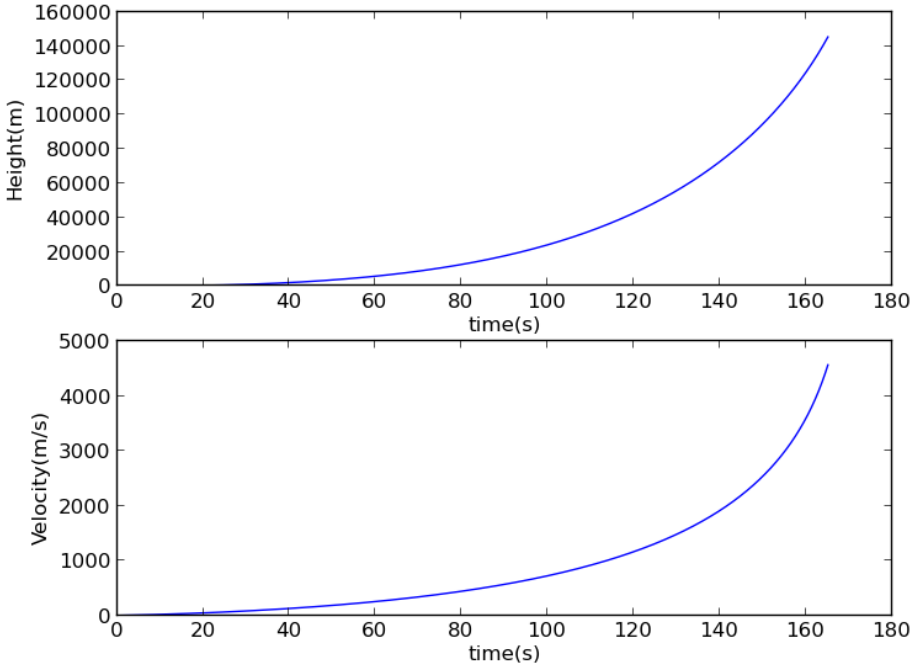

Figure 2: Height and velocity vs. time, no friction

2 Problem 2

2.1 Description

Problem 2 (25 Points). In Kerbal Space Program (KSP), orbital mechanics are simplified using a patched 2-body approach, rather than solving the full N-body problem. That is, the potential is a two-body gravitational potential, and only the LARGEST gravitational force on the object is considered. Assume you have a Saturn V (as in Problem 1) initially in a circular orbit with an altitude of 100 km (this would correspond to the third stage of the Saturn V rocket, with m = 13500 kg). Assume you can apply a maneuver in negligible time, such that the velocity is imparted as an impulse, with a final imparted velocity of 8000 m/s. (This is in addition to the orbital velocity of a rocket at an altitude of 100 km.) (Apologies to Kerbal Space Program https://kerbalspaceprogram.com. No Kerbals were harmed in the making of this final exam. I think.)

a.(5 points) Write an expression for V(r)/m, where V(r) is the potential acting on the rocket, and m is the mass of the rocket, for both the true and patched earth-moon-rocket systems. Compare the two graphically by plotting V(r)/m for both cases in the line connecting the earth and moon. b.(5 points) Calculate the orbital velocity of the Saturn V rocket at an altitude of 100 km. c.(15 points) Modify the planar3body code in Lecture 22 to handle BOTH the true earth-moon-rocket potential, and the KSP patched potential. Code a Moon encounter for Saturn V, such that your rocket is deflected by the Moons gravity (in real life, the astronauts would then burn retrograde to slow down and be captured by the moon). Do this for both the true and patched potentials. Plot the trajectory of your Moon shot. Consider the earth to be at rest for this purpose (also be sure to change the initial x3,y3,vx3,vy3 to be 0,0,0,0). HINT : It is easiest to work in coordinates with the earth at the origin, and use units such that the radii and orbital angular velocity of the moon are 1.0 and 2, respectively.

2.2 Solution

2.2.1 PART a

The code is included in the appendix, and the result diagram is shown here: Fig. 3

2.2.2 PART b

$$\frac{mv^2}{r} = \frac{GMm}{r^2} \longrightarrow v = \sqrt{\frac{GM}{r}} v = 7847.1 m/s$$

2.2.3 PART c

I modified my code so that the radii and angular velocity of moon is 1.0 and 2π. For simplicity, I chose the x coordinate of moon to be 0 and y to be 0.1592. The x and y coordinate for rocket to be -0.00268 , 0.0. For patched potential, the plot is shown here: Fig 4

For real potential, the plot is here: Fig. 5

We can see there is a little difference between two cases.

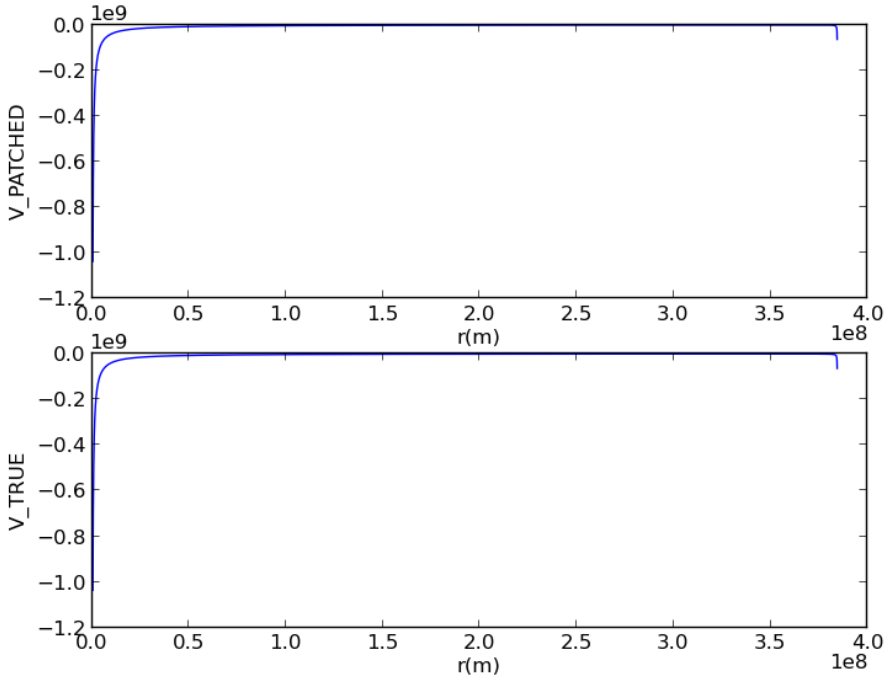

Figure 3: real and patched potential for moon-earth-rocket system

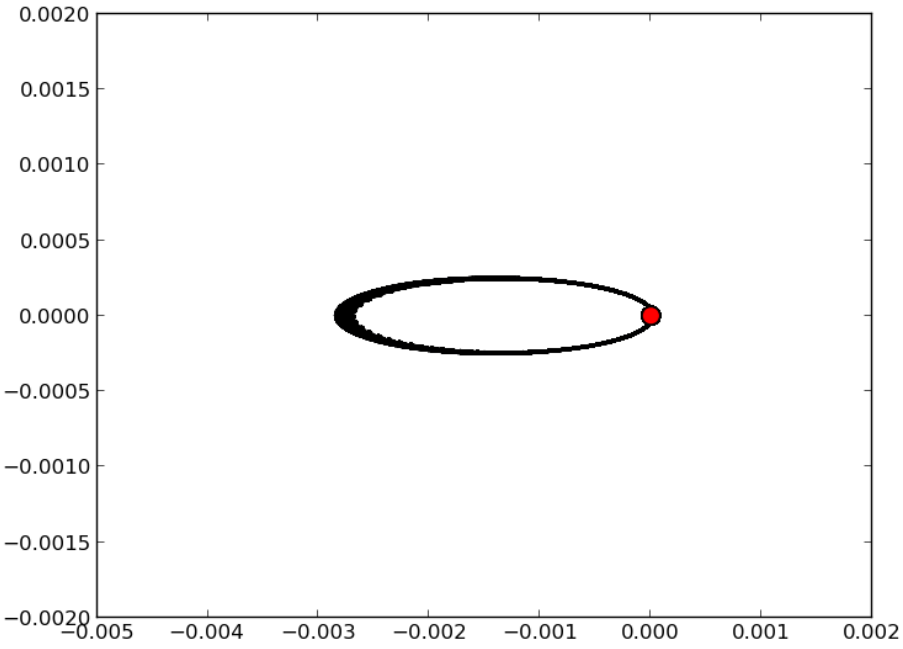

Figure 4: trajectory for patched potential for moon-earth-rocket system

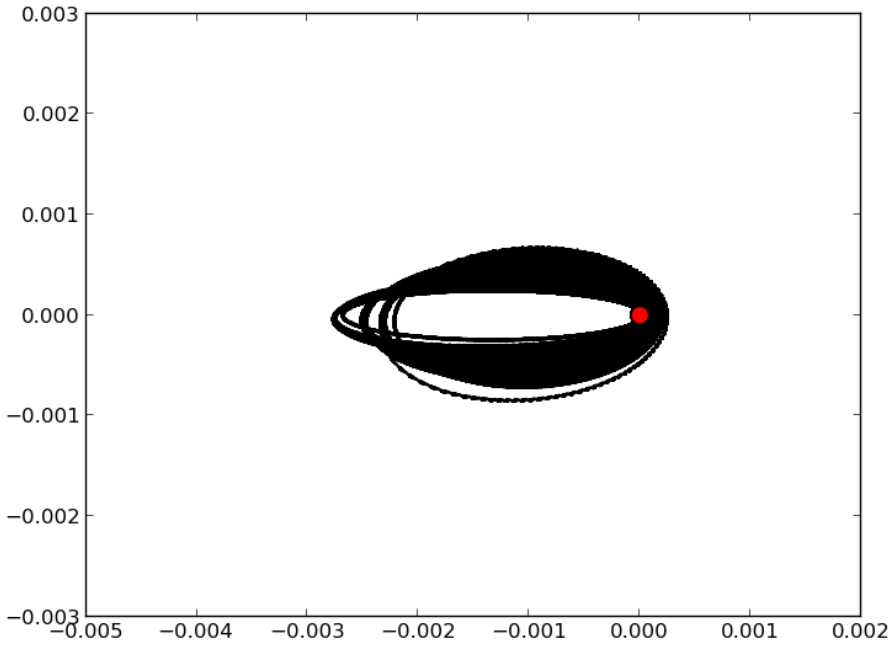

Figure 5: trajectory for real potential for moon-earth-rocket system

Problem 3 (25 points) : Consider a Gaussian wavepacket moving with initial wavenumber k_0 in 1 dimension :

$$\psi(x,0) = \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^{\frac{1}{4}} e^{ik_0 x - \frac{(x-x_0)^2}{4\sigma^2}}$$

Imagine that it evolves under the time-dependent Schrödinger equation :

$$i\hbar\frac{\partial\psi(x,t)}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi(x,t)}{\partial^2 x^2} + V(x)\psi(x,t) \equiv (\mathcal{T}+\mathcal{V})\psi(x,t) \ ,$$

Investigate the case where V(x) is a quantum harmonic oscillator :

$$V(x) = \frac{1}{2}m\omega^2\hat{x}^2$$

For this problem, use units such that $\hbar = m = 1$, and use the "wavepacket" code from Lecture 28 to examine the time evolution of the wavepacket.

a. **(10 points)** Modify the "wavepacket" code in Lecture 28 to solve the QHO potential. Place the minimum of the QHO potential at $L/2$, and set $\omega = \sqrt{2/L}$. Show the time evolution for $E = 1$ (a simple set a snapshots is best).

b. **(10 points)** Repeat (a), except now use the potential

$$V(x) = \frac{1}{2}m\omega^2(\hat{x}^2 + \hat{x}^4)$$

c. **(5 points)** Do you expect the initial state that you've constructed in (a) to remain coherent throughout the evolution? Why or why not? Did you observe what you expect?

3 Problem 3

3.1 Description

3.2 Solution

3.2.1 PART a

The snapshots for the quantum harmonic oscillation is shown below: Fig. 6

3.2.2 PART b

The snapshots for this potential is shown below: Fig. 7

3.2.3 PART c

They should remain coherent. Because for a coherent state, during the evolution in time, the only change will be in its coefficient $e^{-i\omega t}$, therefore should remain coherent. The simulation support this, showing a same width of the wavepacket through out the evolution.

4 Problem 4

4.1 Description

Problem 4 (25 points). Consider the MC integration of a Gaussian distribution with the Metropolis algorithm (metropolis code in Lecture 31). a. (15 points) Modify the metropolis program to perform Metropolis integrations with the same number (A) for both the number of trials (M) and the number of Metropolis steps (N). Check the values of A = 5, 10, 50, 100, 500, 1000, and report the accuracy of the answer (variable ans, compared to the true value). Perform this 5 times and report the average deviation for each choice of A. b. (10 points) At what point do you achieve accuracy within 10true answer? If you had to change either M or N (but not both) by a factor of two to achieve better accuracy, which would you do? Why? Support your case with examples from the code.

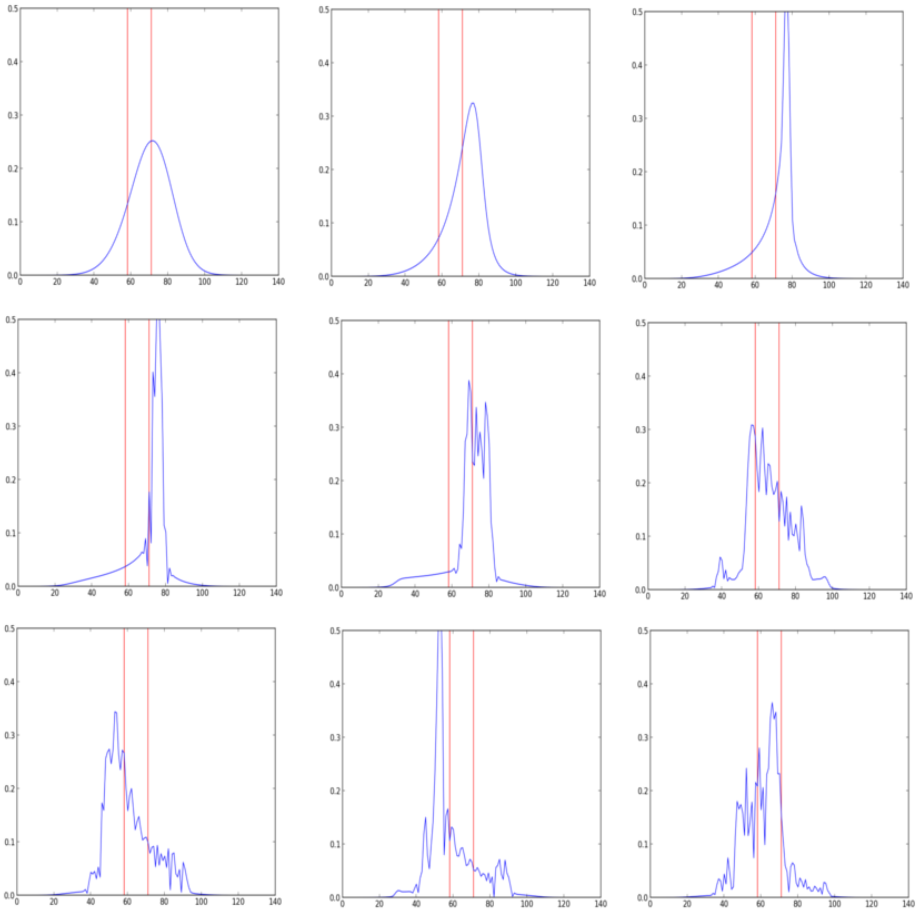

Figure 6: snapshots for quantum harmonic oscillation

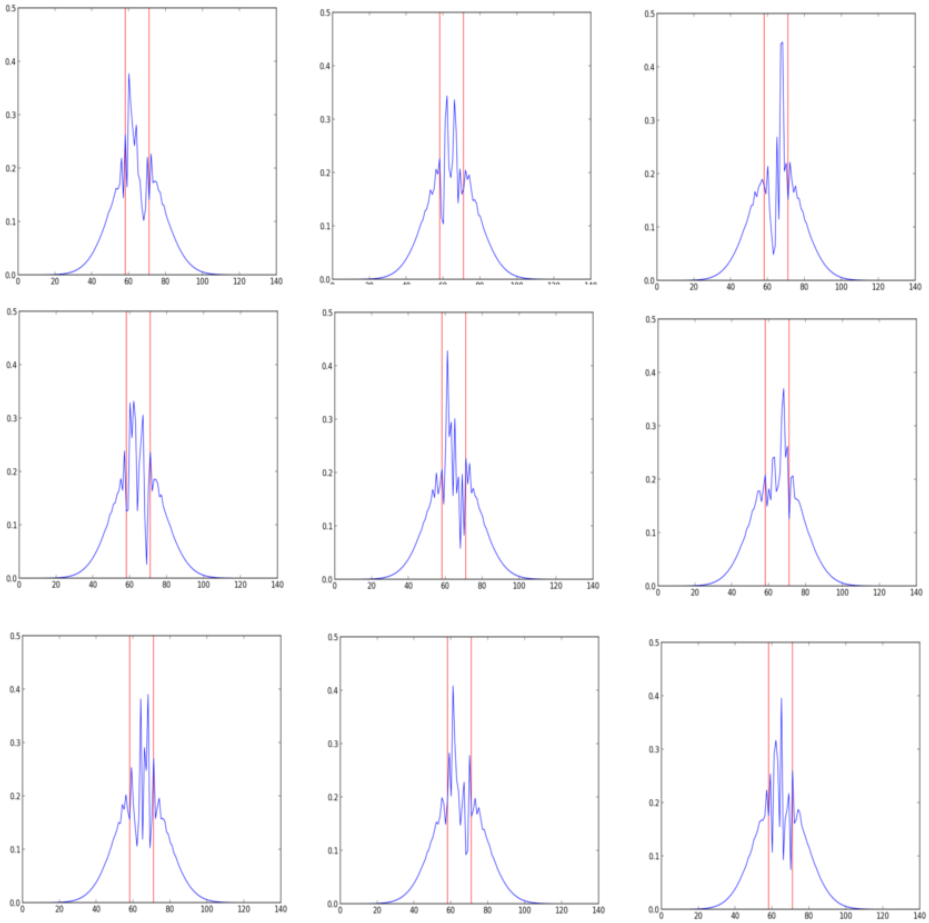

Figure 7: snapshots for specific potential

4.2 Solution

4.2.1 PART a

Here is the table of my result: 1

I used delta = 1

4.2.2 PART b

I modified the code(included in the appendix) to try 100 trials for each M and N, see if any of those trial have accuracy lower than 10-percent. Turned out, although the random nature make this result only a statistical one, A = 20 would be a ideal number.

I think changing N will be a better idea. Because N is the number for each round of M, if change M, the each individual cycle still gives a result with large error, the total average will not improve as efficient as changing N, which improve the individual result for M cycle.

When using 500 for both M,N, the result is 1.0039 ± 0.0027 When changing M to 1000, the result is 1.0031 ± 0.0019 When changing N to 1000, the result is 1.0019 ± 0.0019

Apparently changing N is more effective. Although one trial won't be representative enough, choosing a big number should help improve the result.

Also I repeat the code for 10-percent with either M or N times 2, the result showed the biggest number for M times 2 is bigger. Indicating changing N is more effective

Acknowledgements

I discussed this assignment with my classmates and used material from the cited references, but this write-up is my own.

References

[1] PHY 410-505 Webpage, `http://www.physics.buffalo.edu/phy410-505`.

[2] Wiki for air resistance `http://en.wikipedia.org/wiki/Drag_%28physics%29`

A Appendix

A.1 python code

The following python code was used to obtain the results in this report:

```
# balle - Program to compute the trajectory of a baseball
#         using the Euler method.
# Adapted from Garcia, Numerical Methods for Physics, 2nd Edition

#* Set initial position and velocity of the baseball


#Modified by Han Wen for final project
#2014-12-09
from math import *




#y1 = 0.0
speed = 0.0
#theta = 0.0
r1 = [0.0] * 2
v1 = [0.0] * 2
r =  [0.0] * 2
v =  [0.0] * 2
accel = [0.0] * 2
radius_e = 6371000.0                    #radius of the earth


euler = 0
#y1 = input( "Enter initial height (meters): ")
r1[0] = 0
r1[1] = radius_e            #y1      # Initial vector position
#speed = input( "Enter initial speed (m/s): ")
#theta = input("Enter initial angle (degrees): ")

v1[0] = 0           #speed*cos(theta*pi/180.)   # Initial velocity (x)
v1[1] = 0           #speed*sin(theta*pi/180.)   # Initial velocity (y)
r[0] = r1[0]
r[1] = r1[1]        # Set initial position and velocity
v[0] = v1[0]
v[1] = v1[1]


def Gravitational_acceleration(r):
    return (6.67*5.97*10**13)/r**2
```

```
def rho(r):
    a = 1.2*exp(-r/10000.0)
    return a



#* Set physical parameters (mass, Cd, etc.)
F = 34020000.0   # force of thrust
B = 17212.0      #coefficient of mass decreasing
Cd = 0.35       # Drag coefficient (dimensionless)
area = 25.0  # Cross-sectional area of projectile (m^2)
#grav = 9.81     # Gravitational acceleration (m/s^2)
mass = 2970000.0   # Mass of projectile (kg)
airFlag = 0
#rho = 0.0
airFlag = input( "Air resistance? (Yes:1, No:0): ")
if airFlag == 0 :
    rho_c = 0       # No air resistance
else :
    rho_c = 1     # Density of air (kg/m^3)
#air_const = -0.5*Cd*rho*area/mass  # Air resistance constant

euler = input("Use Euler (0), Euler-Cromer (1), or Midpoint(2) ? ")

#* Loop until ball hits ground or max steps completed
tau = 0.0
tau = input("Enter timestep, tau (sec): ")
#iStep = 0
maxStep = int(165/tau)   # Maximum number of steps
tplot = []
yplot = []
accplot = []
vplot = []
#tNoAir = []
#yNoAir = []
t = 0.0
for iStep in xrange(maxStep) :

    #print iStep
    #* Record position (computed and theoretical) for plotting
    tplot.append( t )   # Record trajectory for plot
    yplot.append( r[1]-radius_e )
    vplot.append(v[1])
    t += tau      # Current time, +save more time than *
#     xNoAir.append( r1[0] + v1[0]*t )
#     yNoAir.append( r1[1] + v1[1]*t - 0.5*grav*t*t )
```

```
    #* Calculate the acceleration of the ball
#     normV = sqrt( v[0]*v[0] + v[1]*v[1] )



    air_const = -0.5*Cd*rho_c*rho(r[1]-radius_e)*area/mass
#     accel[0] = air_const*normV*v[0]   # Air resistance
    accel[1] = air_const*v[1]*v[1]  # Air resistance
    accel[1] -= Gravitational_acceleration(r[1])                # Gravity
    accel[1] += F/mass
    accplot.append(accel[1])
    mass -= B*tau


    #* Calculate the new position and velocity using Euler method
    if ( euler == 0 ) :          # Euler step
#         r[0] += tau*v[0]
        r[1] += tau*v[1]
#         v[0] += tau*accel[0]
        v[1] += tau*accel[1]
    elif ( euler == 1 ) : # Euler-Cromer step
#         v[0] += tau*accel[0]
        v[1] += tau*accel[1]
#         r[0] += tau*v[0]
        r[1] += tau*v[1]
    else :                       # Midpoint step
#         vx_last = v[0]
        vy_last = v[1]
#         v[0] += tau*accel[0]
        v[1] += tau*accel[1]
#         r[0] += tau*0.5*(v[0] + vx_last)
        r[1] += tau*0.5*(v[1] + vy_last)


        #* If ball reaches ground (y<0), break out of the loop
#         if r[1] < 0 :
#             xplot.append( r[0] )  # Record last values computed
#             yplot.append( r[1] )
#             break                  # Break out of the for loop


#     print "a"
#     print r[1]
#     print v[1]
#     print accel[1]
#* Print maximum range and time of flight
print "Maximum range is " + str( r[1] ) + " meters"
```

```
print "the final speed is" + str(v[1])
#print "Time of flight is " +str( iStep*tau ) + " seconds"
print accel[1]
print air_const*v[1]*v[1]
import matplotlib
matplotlib.rcParams['legend.fancybox'] = True
import matplotlib.pyplot as plt
from matplotlib import legend


ax1 = plt.subplot(2,1,1)

p1, = ax1.plot(tplot,yplot)
#p2, = ax1.plot(xNoAir,yNoAir)
plt.xlabel("time(s)")
plt.ylabel("Height(m)")
plt.subplot(2,1,2)
plt.plot(tplot,vplot)
#ax1.legend([p1, p2], ["Numerical", "Exact (no air)"])
plt.xlabel("time(s)")
plt.ylabel("Velocity(m/s)")
plt.show()

# pendul - Program to compute the motion of a simple pendulum
# using the Euler or Verlet method
from cpt import *
from math import *
from numpy import array


class Planar3Body :
    m1 = 0.00           # mass of rockst is negligible
    m2 = 0.08127        # mass of moon
    m3 = 10.0           # mass of earth
    # use units such that G*(m1+m2+m3) = 4*pi**2
    G = 4 * math.pi**2 / (m1 + m2 + m3)#248363527.3          #4 * math.pi**2
#/ (m1 + m2 + m3)       set

    def __init__( self, m1, m2, m3 ):
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3
        self.G = 4 * math.pi**2 / (m1 + m2 + m3)#248363527.3#4 * math.pi**2 #/ (m1

    def __call__(self, trv ) :

        t = trv[0]
        x1 = trv[1] ; y1 = trv[2]  ;  vx1 = trv[3]  ;  vy1 = trv[4]
```

```
        x2 = trv[5] ; y2 = trv[6]  ;  vx2 = trv[7]  ;  vy2 = trv[8]
        x3 = trv[9] ; y3 = trv[10] ;  vx3 = trv[11] ;  vy3 = trv[12]

        r12 = math.sqrt( (x1 - x2)**2 + (y1 - y2)**2 )
        r13 = math.sqrt( (x1 - x3)**2 + (y1 - y3)**2 )
        r23 = math.sqrt( (x2 - x3)**2 + (y2 - y3)**2 )

        #ax1 = - self.G * self.m2 * (x1 - x2) / r12**3 - self.G * self.m3 * (x1 - x
#real
        #ay1 = - self.G * self.m2 * (y1 - y2) / r12**3 - self.G * self.m3 * (y1 - y
#real


        if self.G * self.m2/r12 >  self.G * self.m3/r13:

            ax1 = - self.G * self.m2 * (x1 - x2) / r12**3          #patched
            ay1 = - self.G * self.m2 * (y1 - y2) / r12**3          #patched
        else:
            ax1 = - self.G * self.m3 * (x1 - x3) / r13**3         #patched
            ay1 = - self.G * self.m3 * (y1 - y3) / r13**3         #patched

        ax2 = - self.G * self.m1 * (x2 - x1) / r12**3 - self.G * self.m3 * (x2 - x3
        ay2 = - self.G * self.m1 * (y2 - y1) / r12**3 - self.G * self.m3 * (y2 - y3

        ax3 = 0#- self.G * self.m1 * (x3 - x1) / r13**3 - self.G * self.m2 * (x3 -
        ay3 = 0#- self.G * self.m1 * (y3 - y1) / r13**3 - self.G * self.m2 * (y3 -

        #print [1.0, vx1, vy1, ax1, ay1, vx2, vy2, ax2, ay2, vx3, vy3, ax3, ay3]

        return [1.0, vx1, vy1, ax1, ay1, vx2, vy2, ax2, ay2, vx3, vy3, ax3, ay3]


def main () :

    method = input( "Choose a numerical method : RK4 (0), or Adaptive RK4 (1): ")
    #m1, m2, m3 = input (" Enter m1, m2, m3: ")
    m1 = 0.00            # mass of rockst is negligible
    m2 = 0.08127         # mass of moon
    m3 = 10.0            # mass of earth
    #x1, y1, vx1, vy1 = input(" Enter x1, y1, vx1, vy1: ")
    x1 = -0.00268
    y1 = 0.0
    vx1 = 0.0
    vy1 = 15.49
    #x2, y2, vx2, vy2 = input(" Enter x2, y2, vx2, vy2: ")
    x2 = 0.0
    y2 = 0.1592
    vx2 = -1.0
```

```
vy2 = 0.0

tau = input("␣Enter␣time␣step␣dt␣=␣")
t_max = input("␣Enter␣total␣time␣:␣")


planar3Body = Planar3Body( m1, m2, m3  )

# compute position and velocity of m3 assuming center of mass at origin
x3 = 0#- (planar3Body.m1 * x1 + planar3Body.m2 * x2) / planar3Body.m3
y3 = 0#- (planar3Body.m1 * y1 + planar3Body.m2 * y2) / planar3Body.m3
vx3 = 0#- (planar3Body.m1 * vx1 + planar3Body.m2 * vx2) / planar3Body.m3
vy3 = 0#- (planar3Body.m1 * vy1 + planar3Body.m2 * vy2) / planar3Body.m3
#print " x3 = ", x3, " y3 = ", y3, " vx3 = ", vx3, " vy3 = ", vy3

# compute net angular velocity
I = planar3Body.m1 * (x1**2 + y1**2) + planar3Body.m2 * (x2**2 + y2**2) + plana
L = planar3Body.m1 * (x1*vy1 - y1*vx1) + planar3Body.m2 * (x2*vy2 - y2*vx2) + p
omega = L / I
print "␣Net␣angular␣velocity␣=␣", omega


accuracy = 1e-6
# Initialize vector
t = 0.0
xv = [ t, x1, y1, vx1, vy1, x2, y2, vx2, vy2, x3, y3, vx3, vy3 ]


t_plot = []
x1_plot = []
y1_plot = []
x2_plot = []
y2_plot = []
x3_plot = []
y3_plot = []
dt_min = tau
dt_max = tau

print ''
print ''
while xv[0] < t_max :

    #* Record angle and time for plotting
    t_plot.append( xv[0] )
    x1_plot.append( xv[1] )
    y1_plot.append( xv[2] )
    x2_plot.append( xv[5] )
    y2_plot.append( xv[6] )
```

```
            x3_plot.append( xv[9] )
            y3_plot.append( xv[10] )
        # for ival in xv :
        #     print '{0:12.6f}'.format( ival),
        # print ''


            if  method == 0  :
                RK4_step( xv, tau, planar3Body )
            elif method == 1 :
                tau = RK4_adaptive_step(xv, tau, planar3Body, accuracy)
                dt_min = min(tau, dt_min)
                dt_max = max(tau, dt_max)


    import matplotlib
    matplotlib.rcParams['legend.fancybox'] = True
    import matplotlib.pyplot as plt

    plt.scatter( x1_plot, y1_plot, c='blue', s=1 )
#   plt.scatter( x2_plot, y2_plot, c='green', s=11*1 )
    plt.scatter( x3_plot, y3_plot, c='red', s=109*1 )

    plt.show()



if __name__ == "__main__" :
    main()

#PHY 505 final project, problem 2 part a
#This program is going to plot the gravitational potential for both cases
#Han Wen 2014-12-10


import matplotlib.pyplot as plt
from matplotlib import legend

Gme = 6.67*5.972e13                    #Gravitational constant x mass of earth
Gmm = 6.67*7.348e11                    #Gravitational constant x mass of moon
rem = 384400000.0                        #distance from earth to moon

r_out = []
v_real = []
v_patched = []

N = 10000
```

```
dr = rem/N                          #step length
r = 10*dr                              #distance to earth

for i in range(N-11):
    Ve = -Gme/r
    Vm = -Gmm/(rem-r)
    r_out.append(r)
    if abs(Ve)>abs(Vm):
        v_patched.append(Ve)
    elif abs(Ve)==abs(Vm):
        v_patched.append(Ve)
    else:
        v_patched.append(Vm)
    v_real.append(Ve+Vm)
    r += dr

plt.subplot(2,1,1)
plt.plot(r_out,v_patched)
plt.xlabel("r(m)")
plt.ylabel("V_PATCHED")

plt.subplot(2,1,2)
plt.plot(r_out,v_real)
plt.xlabel("r(m)")
plt.ylabel("V_TRUE")

plt.show()

import math
import cmath
import time
import matplotlib
matplotlib.use('TkAgg')
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation


class Wavepacket :

    def __init__(self, N=600, L=100., dt=0.1, periodic=True):

        self.h_bar = 1.0              # Planck's constant / 2pi in natural units
        self.mass = 1.0               # particle mass in natural units

        # The spatial grid
        self.N = N                    # number of interior grid points
        self.L = L                    # system extends from x = 0 to x = L
```

```
        self.dx = L / float(N + 1)    # grid spacing
        self.dt = dt                  # time step
        self.x = []                   # vector of grid points
        self.periodic = periodic      # True = periodic, False = Dirichlet boundary

        # The potential V(x)
        self.V_0 = 1.0                # height of potential barrier
        self.V_width = 10.0           # width of potential barrier
        self.V_center = 0.5 * L       # center of potential barrier
        self.gaussian = True          # True = Gaussian potential, False = step pote


        # Initial wave packet
        self.x_0 = L / 2.0            # location of center
        self.E = 1.0                  # average energy
        self.sigma_0 = L / 10.0       # initial width of wave packet
        self.psi_norm = 1.0           # norm of psi
        self.k_0 = 0.0                # average wavenumber
        self.velocity = 0.0           # average velocity

        self.t = 0.0                  # time
        self.psi = []                 # complex wavefunction
        self.chi = []                 # wavefunction for simplified Crank-Nicholson
        self.a = []
        self.b = []                   # to represent tridiagonal elements of matrix
        self.c = []
        self.alpha = 0.0
        self.beta = 0.0               # corner elements of matrix Q

        # reset global vectors
        self.psi = [0 + 1j*0 for j in range(N)]
        self.chi = [0 + 1j*0 for j in range(N)]

        # reset time and the lattice
        self.t = 0.0
        self.dx = L / float(self.N + 1)
        self.x = [ float(j * self.dx) for j in range(self.N) ]

        # initialize the packet
        self.k_0 = math.sqrt(2*self.mass*self.E - self.h_bar**2 / 2 / self.sigma_0
        self.velocity = self.k_0 / self.mass
        self.psi_norm = 1 / math.sqrt(self.sigma_0 * math.sqrt(math.pi))
        for j in range(self.N):
            exp_factor = math.exp( - (self.x[j] - self.x_0)**2 / (2 * self.sigma_0
            self.psi[j] = (math.cos(self.k_0 * self.x[j]) + 1j * math.sin(self.k_0
            self.psi[j] *= exp_factor * self.psi_norm

        # elements of tridiagonal matrix Q = (1/2)(1 + i dt H / (2 hbar))
```

```
        for j in range(self.N):
            self.a.append( - 1j * self.dt * self.h_bar / (8 * self.mass * self.dx**
            self.b.append( 0.5 + 1j * self.dt / (4 * self.h_bar) *
                      (self.V(self.x[j]) + self.h_bar**2 / (self.mass * self.dx**2)
            self.c.append( - 1j * self.dt * self.h_bar / (8 * self.mass * self.dx**
        self.alpha = self.c[N-1]
        self.beta = self.a[0]




    def V(self, x):
        half_width = abs(0.5 * self.V_width)
        if self.gaussian:
            #return self.V_0 * math.exp(-(x - self.V_center)**2 / (2 * self.half_w
            #return self.V_0*(x-self.V_center)**(-1)
            return self.mass*(x-self.L/2)**2/self.L
        else:
            if abs(x - self.V_center) <= half_width:
                return self.V_0
            else:
                return 0.0


    def solve_tridiagonal(self, a, b, c, r, u):
        n = len(r)
        gamma = [ 0 + 1j*0 for j in range(n) ]
        beta = b[0]
        u[0] = r[0] / beta
        for j in range(1, n):
            gamma[j] = c[j-1] / beta
            beta = b[j] - a[j] * gamma[j]
            u[j] = (r[j] - a[j] * u[j-1]) / beta
        for j in range(n-2, -1, -1):
            u[j] -= gamma[j+1] * u[j+1]

    def solve_tridiagonal_cyclic(self, a, b, c, alpha, beta, r, x):
        n = len(r)
        bb = [0 + 1j*0 for j in range(self.N)]
        u = [0 + 1j*0 for j in range(self.N)]
        z = [0 + 1j*0 for j in range(self.N)]
        gamma = -b[0]
        bb[0] = b[0] - gamma
        bb[n-1] = b[n-1] - alpha * beta / gamma
        for i in range(1, n-1):
            bb[i] = b[i]
        self.solve_tridiagonal(a, bb, c, r, x)
        u[0] = gamma
```

```
        u[n-1] = alpha
        for i in range(1, n-1):
            u[i] = 0
        self.solve_tridiagonal(a, bb, c, u, z)
        fact = x[0] + beta * x[n-1] / gamma
        fact /= 1.0 + z[0] + beta * z[n-1] / gamma
        for i in range(n):
            x[i] -= fact * z[i]

#    T = 5.0                    # time to travel length L


class Animator :


    def __init__(self, periodic=True,wavepacket=None):
        self.avg_times = []
        self.periodic = periodic
        self.wavepacket = wavepacket
        self.t = 0.
        self.fig, self.ax = plt.subplots()

        self.myline = plt.axvline( x=(self.wavepacket.V_center - 0.5 * self.wavepa
                                   color='r'
            )
        self.myline = plt.axvline( x=(self.wavepacket.V_center + 0.5 * self.wavepa
                                   color='r'
            )
        self.ax.set_ylim(0,0.5)
        initvals = [ abs(ix) for ix in self.wavepacket.psi]
        self.line, = self.ax.plot(initvals)


    def update(self, data) :
        self.line.set_ydata(data)
        return self.line,

    def time_step(self):
        while True :
            start_time = time.clock()
            if self.periodic:
                self.wavepacket.solve_tridiagonal_cyclic(self.wavepacket.a, self.w
                                                         self.wavepacket.c, self.w
                                                         self.wavepacket.psi, self.
            else:
                self.wavepacket.solve_tridiagonal(self.wavepacket.a, self.wavepacke
                                                  self.wavepacket.c, self.wavepacke
            for j in range(self.wavepacket.N):
```

```
                self.wavepacket.psi[j] = self.wavepacket.chi[j] - self.wavepacket.p
            self.t += self.wavepacket.dt;
            end_time = time.clock()
            print 'Tridiagnonal step in ' + str(end_time - start_time)
            yield [abs(ix) for ix in self.wavepacket.psi]

    def create_widgets(self):
        self.QUIT = Button(self, text="QUIT", command=self.quit)
        self.QUIT.pack(side=BOTTOM)

        self.draw = Canvas(self, width="600", height="400")
        self.draw.pack(side=TOP)

    def animate(self) :
        self.ani = animation.FuncAnimation( self.fig,        # Animate our figure
                                            self.update,     # Update function drau
                                            self.time_step,  # "frames" function d
                                            interval=50,     # 50 ms between iterat
                                            blit=False       # don't blit anything
                                            )


wavepacket = Wavepacket(N=128)
animator = Animator(periodic=True,wavepacket=wavepacket)
animator.animate()
plt.show()

import math
import random

class Metropolis :
    def __init__(self, delta=1.0) :
        self.x = 0.0             # initial position of walker
        self.delta = delta       # step size
        self.accepts = 0         # number of steps accepted

    def step(self):
        x_trial = self.x + self.delta * random.uniform(-1, 1)
        ratio = self.P(x_trial) / self.P(self.x)
        if (ratio > random.random()):
            self.x = x_trial
            self.accepts += 1

    def P(self, x):
        # normalized Gaussian function
        return math.exp(-x**2 / 2.0) / math.sqrt(2 * math.pi)

    def f_over_w(self):
```

```
        # integrand divided by weight function
        return float(self.x**2)


print " Monte Carlo Quadrature using Metropolis et al. Algorithm"
print " ================================================================="
delta = 1.0#float(input(" Enter step size delta: "))
M = 4#int(input(" Enter number of trials M: "))
N = 2#int(input(" Enter number of Metropolis steps per trial N: "))


#I modified the code so that we can find the value to achieve more than 10% accurac
for iwho in range (40):
    for iwh in range(100):
        f_sum = 0.0          # accumulator for f(x) values
        f2_sum = 0.0         # [f(x)]**2 values
        err_sum = 0.0        # error estimates

        metropolis = Metropolis( delta=delta )

        for i in range(M):
            avg = 0.0
            var = 0.0
            for j in range(N):
                metropolis.step()
                fx = metropolis.f_over_w()
                avg += fx
                var += fx**2
            avg /= float(N)
            var /= float(N)
            var = var - avg**2
            err = math.sqrt(var / N)
            f_sum += avg
            f2_sum += avg**2
            err_sum += err
        ans = f_sum / float(M)
        std_dev = math.sqrt(f2_sum / M - ans * ans)
        std_dev /= math.sqrt(M-1.0)
        err = err_sum / float(M)
        err /= math.sqrt(M)
        if abs(ans-1) > 0.9:
            break
    if iwh == 99:
        break
    N +=1
    M = 2*N
    #N = M*2
```

print M,N

```
#print ""
#print " Exact answer =", 1.0
#print "      Integral =", ans, "+-", err
#print "     Std. Dev. =", std_dev
#print " Accept ratio =", metropolis.accepts / float(N*M)
#print "percentage=", ans*100
```

Table 1: Result for different A

number of A	**Result**	**Deviation**
5	0.771 ± 0.112	0.315
5	0.532 ± 0.072	0.173
5	0.985 ± 0.073	0.421
5	0.296 ± 0.051	0.11
5	0.44 ± 0.067	0.119
average of 5	0.6048 ± 0.075	0.2276
10	0.699 ± 0.051	0.195
10	1.879 ± 0.123	0.482
10	0.93 ± 0.074	0.384
10	1.003 ± 0.083	0.224
10	0.816 ± 0.073	0.204
average of 10	1.0654 ± 0.0808	0.2978
50	0.994 ± 0.023	0.066
50	0.93 ± 0.02	0.075
50	0.939 ± 0.021	0.065
50	1.082 ± 0.024	0.072
50	0.875 ± 0.021	0.056
average of 50	0.964 ± 0.0218	0.0668
100	1.069 ± 0.013	0.044
100	0.975 ± 0.012	0.039
100	0.958 ± 0.012	0.037
100	0.919 ± 0.011	0.041
100	1.018 ± 0.013	0.047
average of 100	0.9878 ± 0.0122	0.0416
500	1.002 ± 0.003	0.009
500	1.003 ± 0.003	0.009
500	0.992 ± 0.003	0.01
500	0.985 ± 0.003	0.009
500	1.002 ± 0.003	0.01
average of 500	0.9968 ± 0.003	0.0094
1000	1.003 ± 0.001	0.005
1000	1.006 ± 0.001	0.005
1000	1.005 ± 0.001	0.005
1000	1.004 ± 0.001	0.005
1000	1.004 ± 0.001	0.004
average of 1000	1.0044 ± 0.001	0.0048