# PHY 410
# Homework Assignment 8

Han Wen

Person No. 50096432

November 9, 2014

**Abstract**

The goal of this assignment is to get more familiar with RK-4 method in ODE, as well as other methods, examples of celestial orbits, quantum harmonic oscillator and Kronig-Penney model will be presented.

## Contents

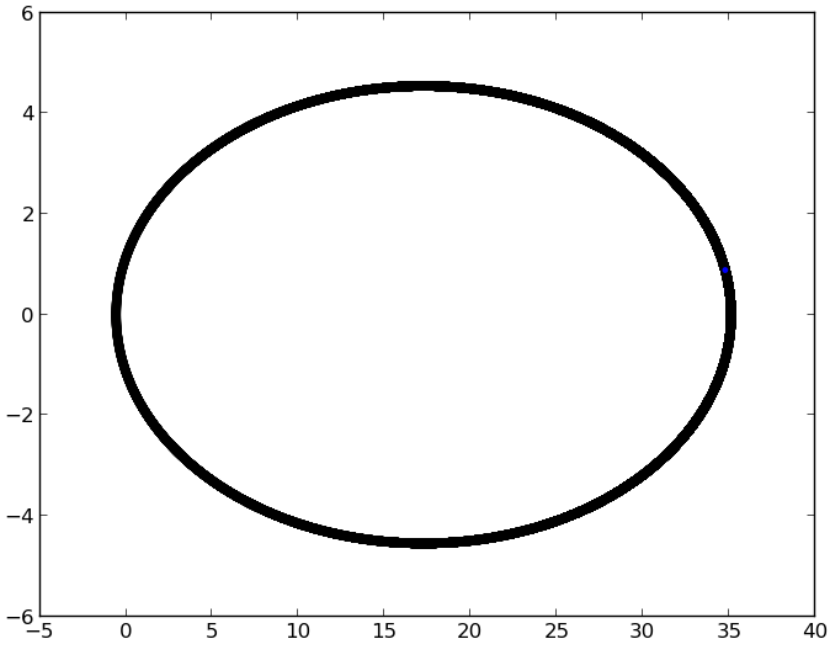


Figure 1: Halley Comet orbit with RK4 method

# 1 Problem 1

## 1.1 Description

Generate the orbits of (1) Halley's Comet and at least one planetary orbit using fixed-time step Runge-Kutta and adaptive time step Runge-Kutta and compare the efficiency of the two methods (you can estimate errors by measuring the period for example); and (2) a few interesting orbits near the Lagrangian points of the restricted 3-body problem, for example the Halo orbit selected by NASA for the James Webb telescope described by Dr. Mather earlier this year, or a Lissajous orbit.

## 1.2 Numerical Analysis

### 1.2.1 part a

Using the program, combined with the data from wikipedia [2], when using RK-4 method: Enter aphelion distance in AU: 35.1
Enter eccentricity: 0.967
Semimajor axis a = 17.8444331469 AU
Period T = 75.3796531592 yr
$v_y(0)$ = 0.192656328719 AU/yr
With the time step 0.001, the result is shown below: Fig. 1

When using RK-4 adaptive method: Enter aphelion distance in AU: 35.1
Enter eccentricity: 0.967

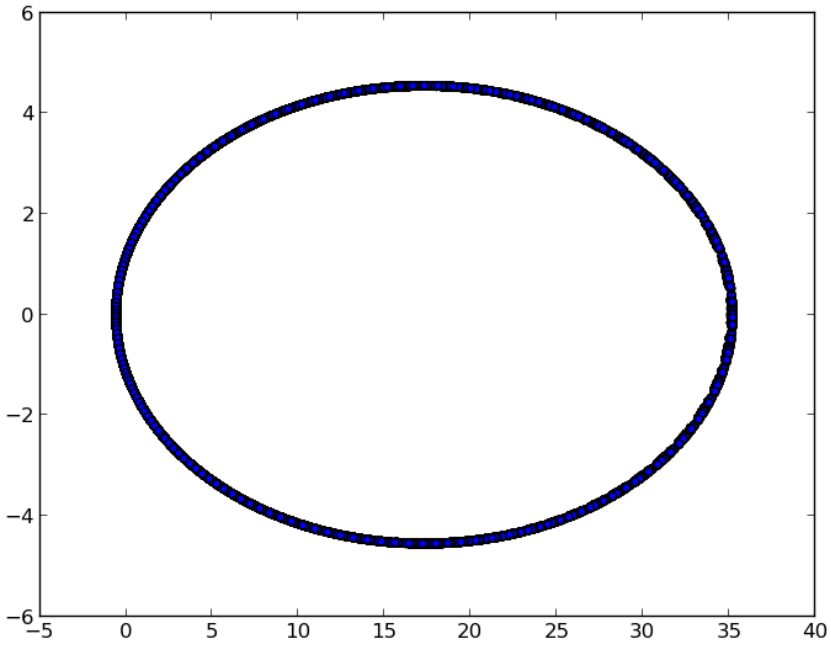


Figure 2: Halley Comet orbit with RK4-adaptive method

Semimajor axis a = 17.8444331469 AU
Period T = 75.3796531592 yr
$v_y(0)$ = 0.192656328719 AU/yr
Enter step size dt: 0.001
Enter desired accuracy for adaptive integration: 0.000001
The result is shown below: Fig 2

I found in this case the regular RK-4 method is more efficient.

### 1.2.2 part b

With the parameters included in here [3], around point L2, with the proper initial condition, I generate the halo orbit shown here: Fig. 3. We can see it's fairly a good plot, although since the data is not precise enough, a little "Lissajous orbit" style will be mixed into the simulation plot.

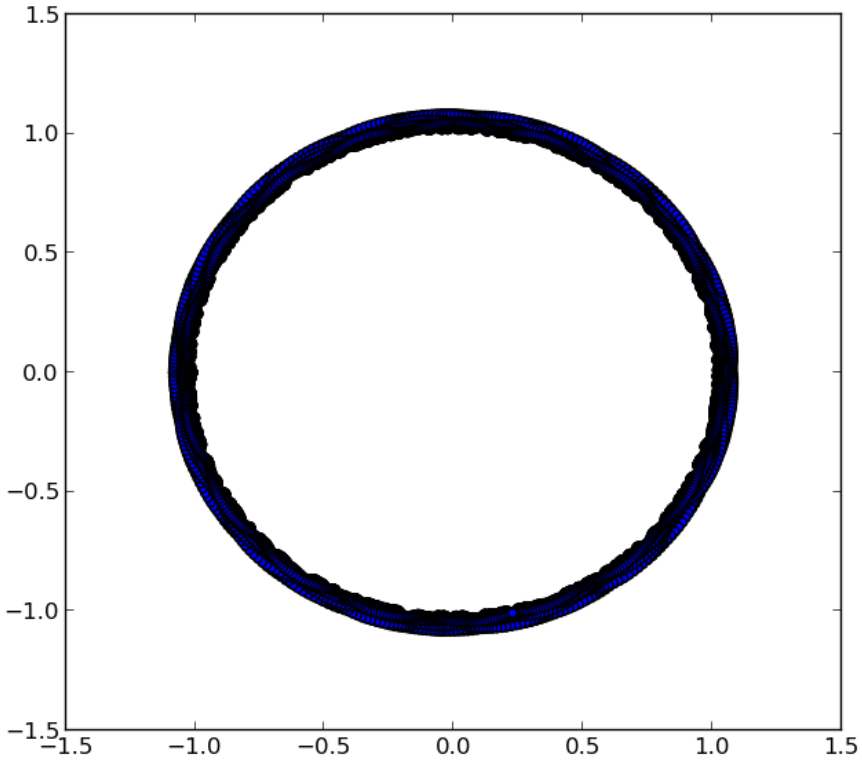


Figure 3: Halo orbit

# 2 Problem 2

## 2.1 Description

Modify the Schroedinger code to add (1) a cubic $x^3$ term and (2) a quartic $x^4$ term, to the harmonic oscillator potential. Discuss how each type of perturbation modifies the energy levels and eigenfunctions. Be sure to increase the left and right boundaries if needed!

## 2.2 Result

To be consistent with the perturbation theory, I added a parameter $\lambda$ before the cubic or the quartic term, and making $\lambda$ a rather small number. I chose E=5 in our case so that the number of eigenvalues is appropriate. Without the perturbation, the result is: Level Energy Simple Steps Secant Steps

—— ———————— ———————— ————————

-1.001250 0.500076
1.001250 0.500076

-1.732500 1.500095
1.732500 1.500095

-2.236250 2.500113
2.236250 2.500113

-2.646250 3.500115
2.646250 3.500115

-3.001250 4.500120
3.001250 4.500120

-3.317500 5.500139
3.317500 5.500139
With the diagram: Fig 4

When it comes to cubic perturbation, with $\lambda = 0.05$:
Level Energy Simple Steps Secant Steps

—— ———————— ———————— ————————

-9.903750 0.475473
0.933750 0.475473

-9.697500 1.422769
1.568750 1.422769

-9.482500 2.329516
1.973750 2.329516

-9.262500 3.166497
2.272500 3.166497

-9.041250 3.921575

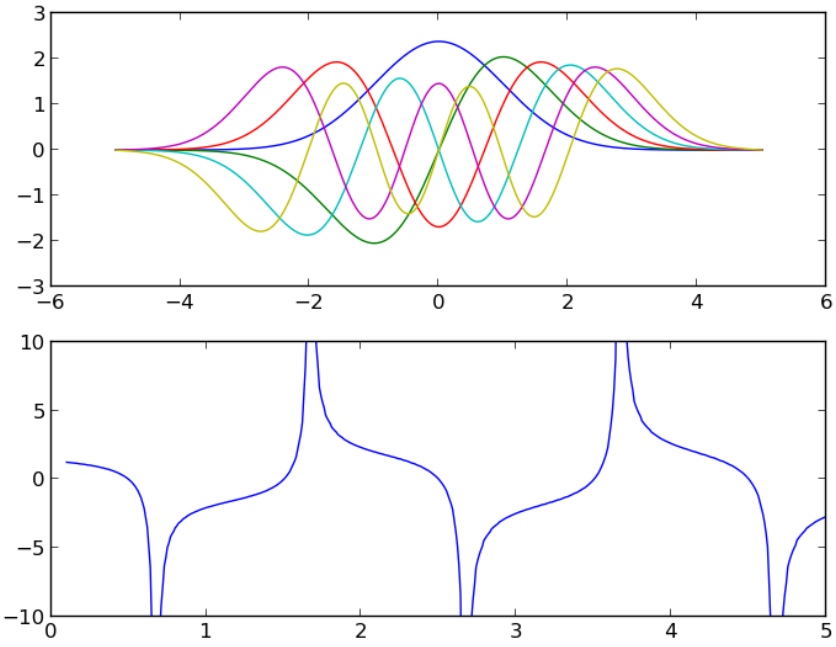


Figure 4: Harmonic oscillation without perturbation

2.505000 3.921575

-8.811250 4.618098
2.697500 4.618098

-8.198750 6.054249
3.047500 6.054249

And the plot: 5
When quadric perturbation, with $\lambda = 0.05$:
Level Energy Simple Steps Secant Steps
——— ———————— ———————— ————————

-0.986250 0.532749
0.986250 0.532749

-1.620000 1.653598
1.620000 1.653598

-2.021250 2.874194
2.021250 2.874194

-2.328750 4.176595

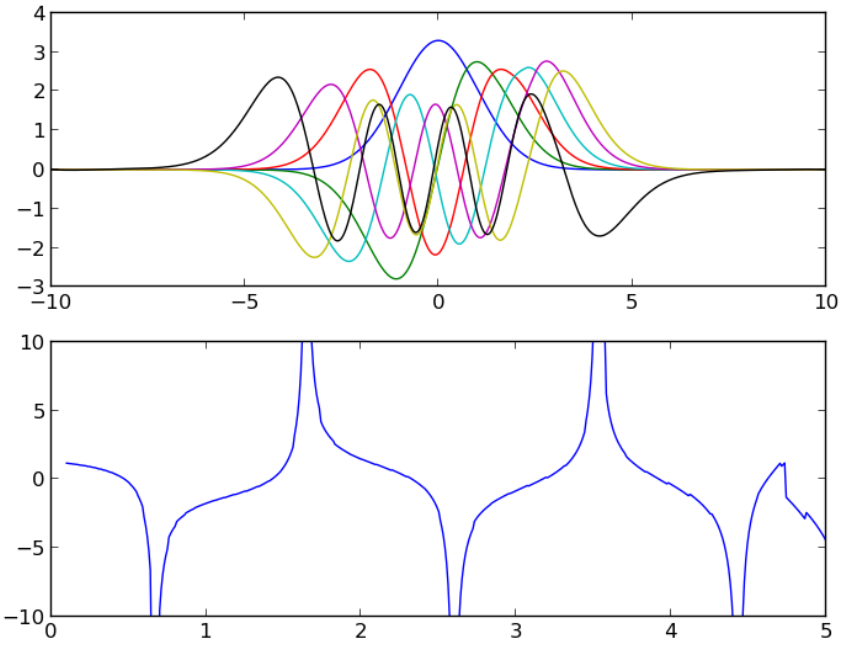


Figure 5: Harmonic oscillation with cubic perturbation

2.328750 4.176595

-2.581250 5.549612
2.581250 5.549612

And the plot: 6

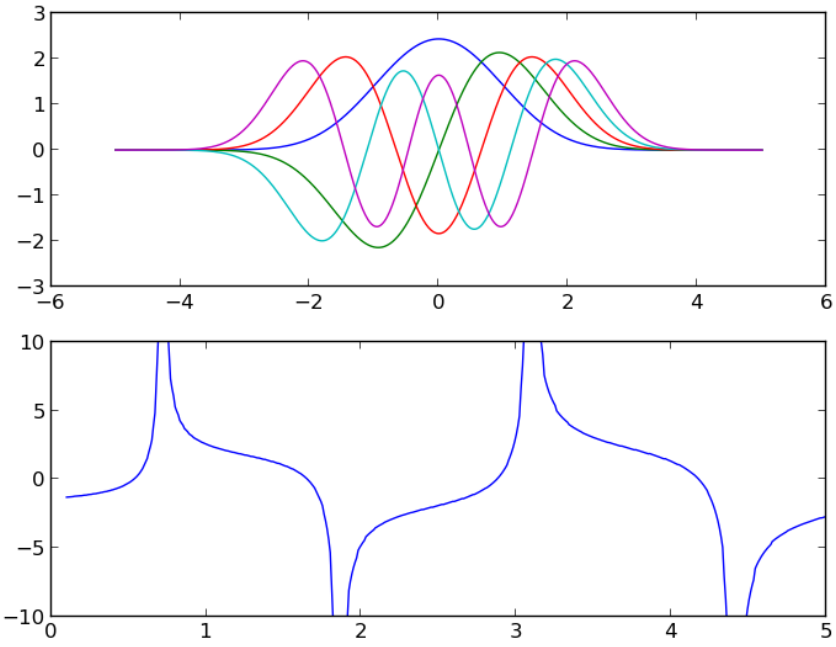


Figure 6: Harmonic oscillation with quadric perturbation

We can see, with the cubic perturbation, the number of energy level increase from 5 to 6, the whole numbers of level shifted and the values of energy for the first 5 levels decrease a little, while with the quadric perturbation, the number remains the same and the values of energy increase a little. Those property origin from the form of the perturbation, namely the cubic form is antisymmetric and the quadric one is symmetric.

# 3 Problem 3

## 3.1 Description

Study the dependence of the Kronig-Penney model band structure on the width and height of the potential step, and describe any systematic trends you notice.

## 3.2 Result

Guided by the theoretical result as well as the symmetric property of the system. Based on the result of different trials, I found when fix the width and increase the height, the band gap increase Fig 7. When fix the the height and change the width, the more close the width is to half of the size of the unit cell, the bigger the gap. When the width is approaching 0 or the size of the unit cell, the gap is vanishing. 8

Here is the plot with width 0.2 height -5.0 Fig 9

Here is the plot with width 0.5 height -5.0 Fig 10

Here is the plot with width 0.2 height -10.0 Fig 11

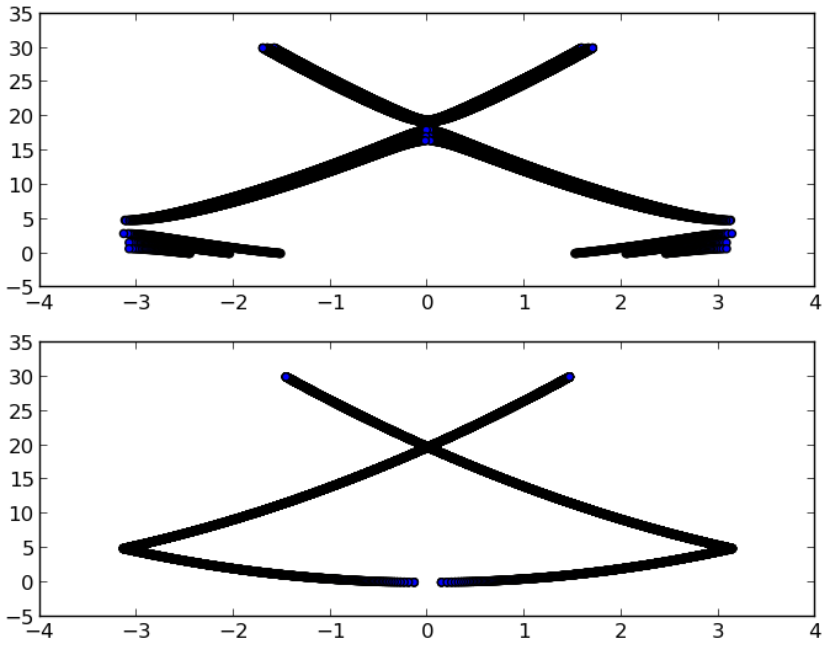


Figure 7: Kronig Penney with height -5 -8 -10 and width 0.2

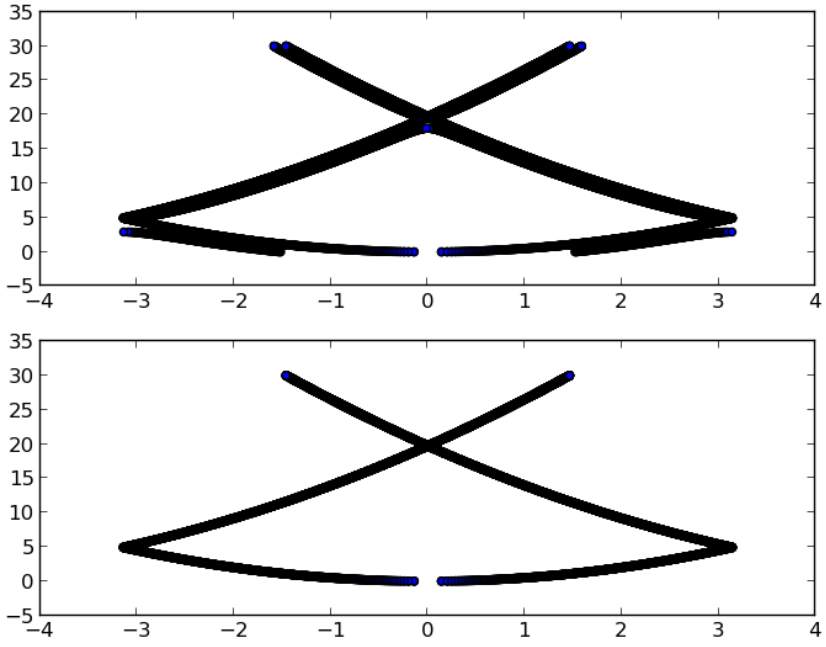


Figure 8: Kronig Penney with width 0.2 0.5 1.0 and height -5.0

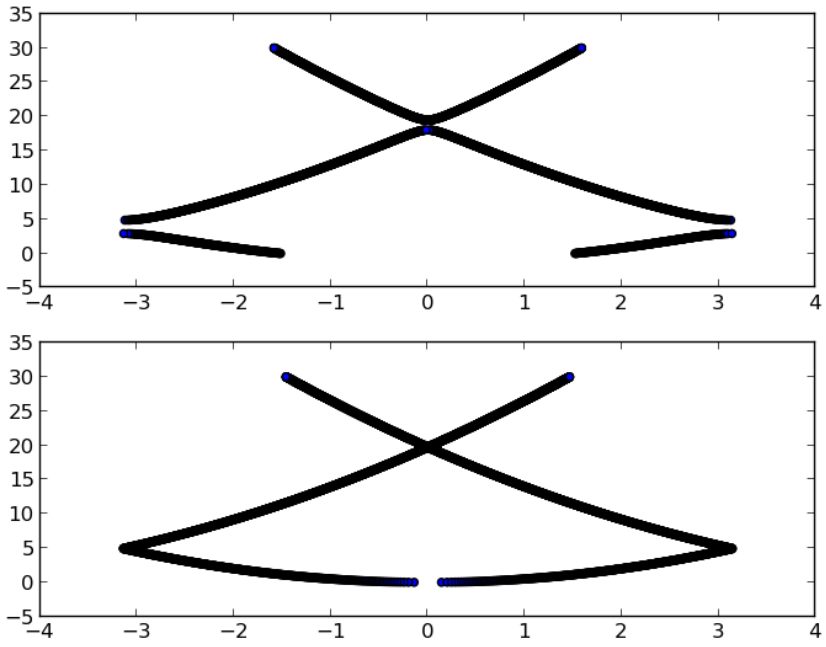


Figure 9: Kronig Penney with width 0.2 height -5.0

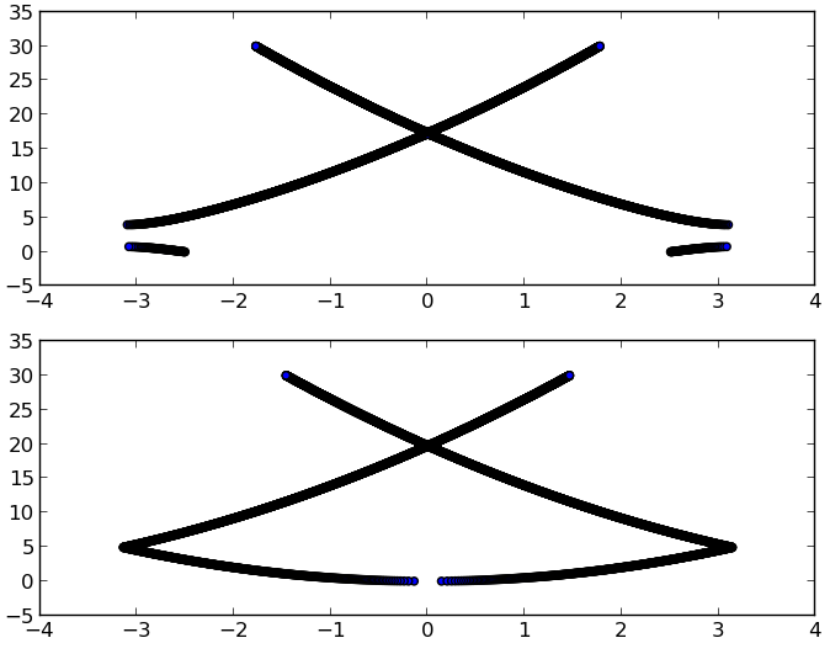


Figure 10: Kronig Penney with width 0.5 height -5.0

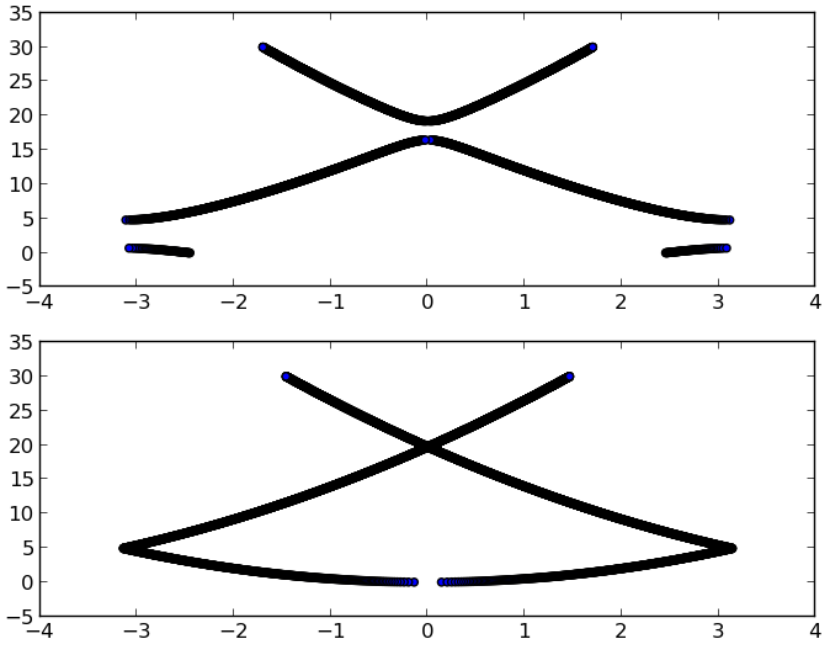


Figure 11: Kronig Penney with width 0.2 height -10.0

## 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] Halley Comet Wikipedia `http://en.wikipedia.org/wiki/Halley%27s_Comet`

[3] Lagrange point `http://www.physics.montana.edu/faculty/cornish/lagrange.pdf`

# A Appendix

## A.1 python code

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

```
# 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 Kepler :
    G_m1_plus_m2 = 0.
    step_using_y = False
    def __init__( self, G_m1_plus_m2, step_using_y ):
        self.G_m1_plus_m2 = G_m1_plus_m2
        self.step_using_y = step_using_y

    def __call__(self, p ) :

        t = p[0]
        x = p[1]
        y = p[2]
        vx = p[3]
        vy = p[4]
        r = math.sqrt(x**2 + y**2)
        ax = - self.G_m1_plus_m2 * x / r**3
        ay = - self.G_m1_plus_m2 * y / r**3
        flow = [ 1, vx, vy, ax, ay ]
        if self.step_using_y:              # change independent variable from t to y
            for i in range(5):
                flow[i] /= vy
        return flow


def main () :


    kepler = Kepler( G_m1_plus_m2 = 4 * math.pi**2, step_using_y=False
)

    #* Select the numerical method to use: Euler or Verlet
    method = input( "Choose a numerical method : RK4 (0), or Adaptive RK4 (1): ")
    print " Kepler orbit using fixed and then adaptive Runge-Kutta"
    r_aphelion = float(input(" Enter aphelion distance in AU: "))
    eccentricity = float(input(" Enter eccentricity: "))
    a = r_aphelion / (1 + eccentricity)
```

```
T = a**1.5
vy0 = math.sqrt(kepler.G_m1_plus_m2 * (2 / r_aphelion - 1 / a))
print " Semimajor axis a = ", a, " AU"
print " Period T = ", T, " yr"
print " v_y(0) = ", vy0, " AU/yr"
tau = float(input(" Enter step size dt: "))
accuracy = float(input(" Enter desired accuracy for adaptive integration: "))



# Initialize vector
xv = [0.0] * 5
xv[0] = 0.0
xv[1] = r_aphelion
xv[2] = 0.0
xv[3] = 0.0
xv[4] = vy0

#* Loop over desired number of steps with given time step
#    and numerical method
nStep = input( "Enter number of time steps: " )

t_plot = []
x_plot = []
y_plot = []
dt_min = tau
dt_max = tau

for  iStep in xrange(nStep) :

    #if ( xv[0] > T ) :
    #    break

    #* Record angle and time for plotting
    t_plot.append( xv[0] )
    x_plot.append( xv[1] )
    y_plot.append( xv[2] )

    if  method == 0  :
        RK4_step( xv, tau, kepler )
    elif method == 1 :
        tau = RK4_adaptive_step(xv, tau, kepler, 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( x_plot, y_plot )

    plt.show()



if __name__ == "__main__" :
    main()

import math
from cpt import *

# the restricted circular planar 3-body problem has one parameter


# represent a point in the extended phase space by a 5-component vector
# trv = [ t, r, v ] = [ t, x, y, vx, vy ]


class Rcp3Body :
    # alpha = m2/(m1+m2) in the webnotes
    a = 0.0

    # switch to zero in on Poincare section point
    # use y instead of t as independent variable
    step_using_y = False
    def __init__(self, a, step_using_y ) :
        self.a = a
        self.step_using_y=step_using_y

    def set_step_using_y( self, step_using_y ) :
        self.step_using_y = step_using_y
    def __call__(self, trv):          # equations in co-rotating frame

        t = trv[0]
        x = trv[1] ; y = trv[2] ; vx = trv[3] ; vy = trv[4]

        d1 = math.pow( (x - self.a)**2 + y**2, 1.5 )
        d2 = math.pow( (x + 1 - self.a)**2 + y**2, 1.5 )

        ax = - (1 - self.a) * (x - self.a) / d1 - self.a * (x + 1 -self.a) / d2 + x
        ay = - (1 - self.a) * y / d1 - self.a * y / d2 + y - 2 * vx

        flow = [ 1.0, vx, vy, ax, ay ]

        if self.step_using_y:         # change integration variable from t to y
```

```
            for i in range(len(flow)):
                flow[i] /= vy

        return flow

    def Jacobi(self, trv):     # Jacobi Integral

        t = trv[0]
        x = trv[1] ; y = trv[2] ; vx = trv[3] ; vy = trv[4]

        r1 = math.sqrt( (x - self.a)**2 + y**2 )
        r2 = math.sqrt( (x + 1 - self.a)**2 + y**2 )

        return x**2 + y**2 + 2 * (1 - self.a) / r1 + 2 * self.a / r2 - vx**2 - vy**

    def f_x(self, x):      # effective x component of force on the x-axis
        return ( x - (1 - self.a) * (x - self.a) / abs(x - self.a) / (x - self.a)**
                   - self.a * (x + 1 - self.a) / abs(x + 1 - self.a) / (x + 1 - se

    def zero(self, f, x_lower, x_upper, accuracy=1.0e-6, max_steps=1000):
        # use bisection search to solve f(x) = 0 in interval [x_lower, x_upper]
        assert x_lower < x_upper, " zero(f, x_lower, x_upper) not bracketed"
        x_mid = (x_upper + x_lower) / 2
        dx = x_upper - x_lower
        f_lower = f(x_lower)
        step = 0
        while abs(dx) > accuracy:
            f_mid = f(x_mid)
            if f_mid == 0:
                dx = 0
            else:
                if f_lower * f_mid > 0:
                    x_lower = x_mid
                    f_lower = f_mid
                else:
                    x_upper = x_mid
                x_mid = (x_upper + x_lower) / 2
                dx = x_upper - x_lower
            step += 1
            assert step < max_steps, " zero(f) too many steps " + str(max_steps)
        return x_mid

# get parameters from user
print " Restricted circular planar 3-body problem"

while True:
    a = input(" Enter alpha = m2/(m1+m2) > 0 and < 0.5: ")
    if a <= 0 or a > 0.5:
```

```
        print " Bad alpha , please try again"
        continue
    break

rcp3Body = Rcp3Body (a, False)

# find Lagrangian points for this alpha
eps = 1e-6
print " Lagrangian points:"
print " L1 : x =", rcp3Body.zero(rcp3Body.f_x, rcp3Body.a - 1 + eps, rcp3Body.a -
print " L2 : x =", rcp3Body.zero(rcp3Body.f_x, -1.5,rcp3Body. a - 1 - eps), " y =
print " L3 : x =", rcp3Body.zero(rcp3Body.f_x, rcp3Body.a + eps, 1.5), " y =", 0.
print " L4 : x =", rcp3Body.a - 0.5, " y =", math.sqrt(3.0) / 2
print " L5 : x =", rcp3Body.a - 0.5, " y =", math.sqrt(3.0) / 2

# get initial values
one = input(" Enter 0 to specify [x,y,vx,vy] or 1 to specify C and [x,y,vx]: ")
if one:
    while True:
        C = input(" Enter value of the Jacobi integral C: ")
        x, y, vx = input(" Enter x, y, vx: ")
        r1 = math.sqrt((x - a)**2 + y**2)
        r2 = math.sqrt((x + 1 - rcp3Body.a)**2 + y**2)
        vy_sqd = - C + x**2 + y**2 + 2 * (1 - rcp3Body.a) / r1 + 2 * rcp3Body.a / r
        if vy_sqd < 0:
            print " Sorry C too large , cannot solve for vy"
        else:
            vy = math.sqrt(vy_sqd)
            print " vy =", vy
            break
else:
    x, y, vx, vy = input(" Enter x, y, vx, vy: ")
    print " Jacobi Integral C =", rcp3Body.Jacobi([0, x, y, vx, vy])

t_max = input(" Enter maximum integration time t_max: ")


crossing = 0
dt = 0.01
t = 0
trv = [ t, x, y, vx, vy ]

x_plot = []
y_plot = []

while t < t_max:
```

```
    # write trajectory point
    t = trv[0]
    x = trv[1] ; y = trv[2] ; vx = trv[3] ; vy = trv[4]
    y_save = y      # remember y to check section crossing
    x_plot.append( x )
    y_plot.append( y )

    # use adaptive Runge-Kutta with default accuracy
    dt = RK4_adaptive_step(trv, dt, rcp3Body)

    # Poincare section at y = 0 and vy positive
    x = trv[1] ; y = trv[2] ; vx = trv[3] ; vy = trv[4]
    if y_save < 0 and y >= 0 and vy >= 0:
        rcp3Body.set_step_using_y(True)
        dy = -y
        RK4_step(trv, dy, rcp3Body)
        t = trv[0] ; x = trv[1] ; y = trv[2] ; vx = trv[3] ; vy = trv[4]
        crossing += 1
        print " Crossing No.", crossing, " at t = ", t, " C = ", rcp3Body.Jacobi(tr
        rcp3Body.set_step_using_y(False)



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

plt.scatter( x_plot, y_plot, c='blue' )

plt.show()

import math
import cpt
import matplotlib.pyplot as plt

class Schroedinger :

    def __init__(self) :
        self.hbar = 1.0                    # Planck's constant / 2pi
        self.m = 1.0                       # particle mass
        self.omega = 1.0                   # oscillator frequency
        self.E = 0.0                       # current energy in search
        self.N = 500                       # number of lattice points = N+1
        self.x_left = -5.0                 # left boundary
        self.x_right = 5.0                 # right boundary
        self.h = (self.x_right - self.x_left) / self.N  # grid spacing

        self.phi_left = [0.0]*(self.N+1)       # wave function integrating from left
```

```
        self.phi_right = [0.0]*(self.N+1)      # wave function integrating from righ
        self.phi = [0.0]*(self.N+1)            # whole wave function

        self.sign = 1                          # current sign used to make F(E) continuou
        self.nodes = 0                         # current number of nodes in wavefunction


    def V(self, x):                            # harmonic oscillator potential
        return 0.5 * self.m * self.omega**2 * x**2 + 0.05*x**4


    def q(self, x):                            # Sturm-Liouville q function
        return 2 * self.m / self.hbar**2 * (self.E - self.V(x))


    def F(self, energy):                       # eigenvalue at F(E) = 0

        # set energy needed by the q(x) function
        self.E = energy

        # find the right turning point
        i_match = self.N
        x = self.x_right                       # start at right boundary
        while self.V(x) > self.E:              # in forbidden region
            i_match -= 1
            x -= self.h
            if i_match < 0:
                raise Exception("can't find right turning point")

        # integrate self.phi_left using Numerov algorithm
        self.phi_left[0] = 0.0
        self.phi_left[1] = 1.0e-10
        c = self.h**2 / 12.0                   # constant in Numerov formula
        for i in range(1, i_match+1):
            x = self.x_left + i * self.h
            self.phi_left[i+1]  = 2 * (1 - 5 * c * self.q(x)) * self.phi_left[i]
            self.phi_left[i+1] -= (1 + c * self.q(x - self.h)) * self.phi_left[i-1]
            self.phi_left[i+1] /= 1 + c * self.q(x + self.h)

        # integrate self.phi_right
        self.phi[self.N] = self.phi_right[self.N] = 0.0
        self.phi[self.N-1] = self.phi_right[self.N-1] = 1.0e-10
        for i in range(self.N - 1, i_match - 1, -1):
            x = self.x_right - i * self.h
            self.phi_right[i-1]  = 2 * (1 - 5 * c * self.q(x)) * self.phi_right[i]
            self.phi_right[i-1] -= (1 + c * self.q(x + self.h)) * self.phi_right[i-
            self.phi_right[i-1] /= 1 + c * self.q(x - self.h)
            self.phi[i-1] = self.phi_right[i-1]
```

```
        # rescale self.phi_left
        scale = self.phi_right[i_match] / self.phi_left[i_match]
        for i in range(i_match + 2):
            self.phi_left[i] *= scale
            self.phi[i] = self.phi_left[i]

        # make F(E) continuous
        # count number of nodes in self.phi_left
        n = 0
        for i in range(1, i_match+1):
            if self.phi_left[i-1] * self.phi_left[i] < 0.0:
                n += 1

        # flip its sign when a new node develops

        if n != self.nodes:
            self.nodes = n
            self.sign = -self.sign

        return ( self.sign *
                 ( self.phi_right[i_match-1] - self.phi_right[i_match+1] -
                   self.phi_left [i_match-1] + self.phi_left[i_match+1] ) /
                (2 * self.h * self.phi_right[i_match]) )

    def normalize(self):
        norm = 0.0
        for i in range(self.N):
            norm += self.phi[i]**2
        norm /= self.N
        norm = math.sqrt(norm)
        for i in range(self.N):
            self.phi[i] /= norm

print " Eigenvalues of the Schroedinger equation"
print " for the harmonic oscillator V(x) = 0.5 x^2"
print " ------------------------------------------"
E_max = input(" Enter maximum energy E: ")

phi_file = open("phi.data", "w")


level = 0                   # level number
E_old = 0.0                 # previous energy eigenvalue

schroedinger = Schroedinger()
schroedinger.E = 0.1 # guess and E below the ground state
```

```
# draw the potential
for i in range(schroedinger.N+1):
        x = schroedinger.x_left + i * schroedinger.h
swrite = '{0:12.6f} {1:12.6f}'.format( x, schroedinger.V(x) )
print swrite
print ''

# find the energy levels
print ""
print " Level        Energy             Simple Steps    Secant Steps"
print " -----    --------------------    -----------    -------------"

x_data = []
phi_data = []

while True:                    # loop over levels

        # estimate next E and dE
        dE = 0.5 * (schroedinger.E - E_old)
        E_old = schroedinger.E
        schroedinger.E += dE

        # use simple search to locate root with relatively low accuracy
        accuracy = 0.01
        schroedinger.E = cpt.root_simple(schroedinger.F, schroedinger.E, dE, accura
        #simple_steps = cpt.root_steps()

        # use secant search with relatively high accuracy
        accuracy = 1.0e-6
        E1 = schroedinger.E + 100 * accuracy # guess second point required
        schroedinger.E = cpt.root_secant(schroedinger.F, schroedinger.E, E1, accura
        #secant_steps = cpt.root_steps()

        #ans = " " + repr(level).rjust(3) + " "*5 + repr(E).ljust(20) + " "*6
        #ans += repr(simple_steps).rjust(3) + " "*11 + repr(secant_steps).rjust(3)
        #print ans
        level += 1

        accuracy = 0.001
        x = cpt.root_simple(schroedinger.q, schroedinger.x_left, schroedinger.h, ac
        swrite = '{0:12.6f} {1:12.6f}'.format( x, schroedinger.E )
        print swrite

        x = cpt.root_simple(schroedinger.q, schroedinger.x_right, -schroedinger.h,
        swrite = '{0:12.6f} {1:12.6f}'.format( x, schroedinger.E )
        print swrite

        print ''
```

```
        schroedinger.normalize()
        x_data.append( [] )
        phi_data.append( [] )
        iphi = len(x_data) - 1
        for i in range(schroedinger.N+1):
                x = schroedinger.x_left + i * schroedinger.h
                x_data[iphi].append(x)
                phi_data[iphi].append( schroedinger.phi[i] )


        if schroedinger.E >= E_max:             # we are done
                break



# print the search function
schroedinger.E = 0.1
dE = 0.01
E_data = []
F_data = []
while schroedinger.E < E_max:
        E_data.append( schroedinger.E )
        F_data.append(  schroedinger.F(schroedinger.E) )
        schroedinger.E += dE


s1 = plt.subplot(2,1,1)
for i in range(len(x_data)) :
        plt.plot( x_data[i], phi_data[i] )

s2 = plt.subplot(2,1,2)
plt.plot( E_data, F_data )
plt.ylim( [-10, 10] )

plt.show()

import math
import cmath
import matplotlib.pyplot as plt

class KronigPenney :
        def __init__(self) :
                self.a = 1.0                                    # size of unit cell - latt
                self.V_0 = -5.0                                 # height of potential bar
                self.Delta = 0.2                                # width of potential barr
```

```
        def solve_for_E(self, E, k):                # to solve 2x2 eigenvalue problem
            # E is the desired enegy (input)
            # k is a list of the two solutions
            q = math.sqrt(2 * E)
            kappa = math.sqrt(2 * (E - self.V_0))
            i = 1.0j
            T11 = ( cmath.exp(i * q * (self.a - self.Delta)) / (4 * q * kappa) *
                    ( cmath.exp(i * kappa * self.Delta)  * (q + kappa)**2 -
                      cmath.exp(-i * kappa * self.Delta) * (q - kappa)**2
) )
            T22 = T11.conjugate()
            T12 = ( -i * cmath.exp(i * q * (self.a - self.Delta)) / (2 * q * kappa)
                    (q**2 - kappa**2) * math.sin(kappa * self.Delta)  )
            T21 = T12.conjugate()

            # solve quadratic determinantal equation
            b = - (T11 + T22)
            c = (T11 * T22 - T12 * T21)
            k[0] = (- b + cmath.sqrt(b**2 - 4*c)) / 2.0
            k[1] = (- b - cmath.sqrt(b**2 - 4*c)) / 2.0
            for j in range(2):
                k[j] = cmath.log(k[j]) / (i * self.a)

        def compute_bands(self, dE, steps, band_file_name):
            # dE = step size in E for search
            # steps = number of steps
            E_values = []
            rq_values = []
            file = open(band_file_name, "w")
            E = dE
            for step in range(steps):
                q = [ 0.0 + 0.0j, 0.0 + 0.0j ]
                self.solve_for_E(E, q)
                for j in range(2):
                    rq = q[j].real
                    if rq > 0.0 and rq < math.pi / self.a:
                        rq_values.append( rq )
                        rq_values.append( -rq )
                        E_values.append( E )
                        E_values.append( E )
                E += dE
            return rq_values, E_values
```

```
kp = KronigPenney()
print "␣Kronig-Penney␣Model"
dE = 0.01
steps = 3000
ax1 = plt.subplot(2,1,1)
print "␣V_0␣=", kp.V_0, "␣Delta␣=", kp.Delta
rq_values1, E_values1 = kp.compute_bands(dE, steps, "band.data")
kp.V_0 = 0.0
print "␣V_0␣=", kp.V_0, "␣Delta␣=", kp.Delta
rq_values2, E_values2 = kp.compute_bands(dE, steps, "band0.data")
plt.scatter( rq_values1, E_values1 )

kp.Delta=0.5
print "␣V_0␣=", kp.V_0, "␣Delta␣=", kp.Delta
rq_values1, E_values1 = kp.compute_bands(dE, steps, "band2.data")
kp.V_0 = 0.0
print "␣V_0␣=", kp.V_0, "␣Delta␣=", kp.Delta
rq2_values2, E2_values2 = kp.compute_bands(dE, steps, "band20.data")
plt.scatter( rq_values1, E_values1 )

kp.Delta= 1.0
print "␣V_0␣=", kp.V_0, "␣Delta␣=", kp.Delta
rq_values1, E_values1 = kp.compute_bands(dE, steps, "band3.data")
kp.V_0 = 0.0
print "␣V_0␣=", kp.V_0, "␣Delta␣=", kp.Delta
rq3_values2, E3_values2 = kp.compute_bands(dE, steps, "band30.data")

#ax1 = plt.subplot(2,1,1)
plt.scatter( rq_values1, E_values1 )
#plt.scatter( rq3_values1, E2_values1 )
#plt.scatter( rq2_values1, E3_values1 )

ax2 = plt.subplot(2,1,2)
plt.scatter( rq_values2, E_values2 )
plt.scatter( rq2_values2, E2_values2 )
plt.scatter( rq3_values2, E3_values2 )

plt.show()
```