Jacob Dennis

21 March 2016

Physics 411

Prof. Gull

Homework 6 – Code and Results

**Harmonic Oscillator**

*Code*

import numpy as np

import matplotlib.pyplot as plt

def HarmonicOscillator(x, t):

return -x

def VectorizedF(r, t):

return np.array([r[1], -r[0]])

def Verlet(x0, v0, f, a, b, N):

h = (b - a) / N

T = np.arange(a, b, h)

X = np.zeros(N)

V = np.zeros(N)

vHalfways = np.zeros(N)

X[0] = x0

V[0] = v0

vHalfways[0] = V[0] + 0.5 \* h \* f(X[0], T[0])

for i in range(N - 1):

X[i + 1] = X[i] + h \* vHalfways[i]

k = h \* f(X[i + 1], T[i] + h)

V[i + 1] = vHalfways[i] + 0.5 \* k

vHalfways[i + 1] = vHalfways[i] + k

return [X, V]

def Leapfrog(r0, f, a, b, N): #R = [positions, velocities]

h = (b - a) / N

T = np.arange(a, b, h)

R = np.zeros([N, len(r0)])

rHalfways = np.zeros([N, len(r0)])

R[0] = r0

rHalfways[0] = R[0] + 0.5 \* h \* f(R[0], T[0])

for i in range(N - 1):

R[i + 1] = R[i] + h \* f(rHalfways[i], T[i] + 0.5 \* h)

rHalfways[i + 1] = rHalfways[i] + h \* f(R[i + 1], T[i] + h)

return np.fliplr(np.rot90(R, k = 3))

def EulerMethod(r0, f, a, b, N): #R = [positions, velocities]

h = (b - a) / N

T = np.arange(a, b, h)

R = np.zeros([N, len(r0)])

R[0] = r0

for i in range(N - 1):

R[i + 1] = R[i] + h \* f(R[i], T[i])

return np.fliplr(np.rot90(R, k = 3))

def BackwardsEulerMethod(r0, f, a, b, N): #R = [positions, velocities]

h = (b - a) / N

T = np.arange(a, b, h)

R = np.zeros([N, len(r0)])

R[0] = r0

for i in range(N - 1):

R[i + 1] = 1.0/(1.0 + h\*\*2) \* R[i] + h \* f(R[i], T[i])

return np.fliplr(np.rot90(R, k = 3))

def Energy(xRange):

xMax = np.amax(xRange)

return xMax

def PlotSetsOfApproximations(tRange, ResultsList, LabelsList, XLabel, YLabel, PlotTitle, SaveName, integer):

plt.clf()

fig, axes = plt.subplots(1, 1)

for i in range(len(ResultsList)):

axes.plot(tRange, ResultsList[i][integer], label = LabelsList[i])

plt.xlabel(XLabel)

plt.ylabel(YLabel)

plt.title(PlotTitle)

axes.set\_ylim(-3.0, 3.0)

plt.legend(loc = 0)

plt.savefig(SaveName)

return None

R0 = [0.0, 1.0]

tRange = [np.linspace(0.0, 10 \* 2.0 \* np.pi, 500), np.linspace(0.0, 10 \* 2.0 \* np.pi, 1000)]

VerletResults1 = Verlet(R0[0], R0[1], HarmonicOscillator, 0.0, 10 \* 2.0 \* np.pi, 500)

LeapfrogResults1 = Leapfrog(R0, VectorizedF, 0.0, 10 \* 2.0 \* np.pi, 500)

EulerResults1 = EulerMethod(R0, VectorizedF, 0.0, 10 \* 2.0 \* np.pi, 500)

BackwardsEulerResults1 = BackwardsEulerMethod(R0, VectorizedF, 0.0, 10 \* 2.0 \* np.pi, 500)

VerletResults2 = Verlet(R0[0], R0[1], HarmonicOscillator, 0.0, 10 \* 2.0 \* np.pi, 1000)

LeapfrogResults2 = Leapfrog(R0, VectorizedF, 0.0, 10 \* 2.0 \* np.pi, 1000)

EulerResults2 = EulerMethod(R0, VectorizedF, 0.0, 10 \* 2.0 \* np.pi, 1000)

BackwardsEulerResults2 = BackwardsEulerMethod(R0, VectorizedF, 0.0, 10 \* 2.0 \* np.pi, 1000)

ResultsList1 = [VerletResults1, LeapfrogResults1, EulerResults1, BackwardsEulerResults1]

ResultsList2 = [VerletResults2, LeapfrogResults2, EulerResults2, BackwardsEulerResults2]

LabelsList = ['Verlet', 'Leapfrog', 'Euler', 'Backwards Euler']

PlotSetsOfApproximations(tRange[0], ResultsList1, LabelsList, 't', 'x', 'Position versus Time - Steps = 500',

'Homework 6 - All Methods Positions 1.png', 0)

PlotSetsOfApproximations(tRange[1], ResultsList2, LabelsList, 't', 'x', 'Position versus Time - Steps = 1000',

'Homework 6 - All Methods Positions 2.png', 0)

PlotSetsOfApproximations(tRange[0], ResultsList1, LabelsList, 't', 'v', 'Velocity versus Time - Steps = 500',

'Homework 6 - All Methods Velocities 1.png', 1)

PlotSetsOfApproximations(tRange[1], ResultsList2, LabelsList, 't', 'v', 'Velocity versus Time - Steps = 1000',

'Homework 6 - All Methods Velocities 2.png', 1)

stepsizeRange = np.linspace(50, 1000, 100)

VerletRange = []

EnergyRange = []

for step in stepsizeRange:

tempVerlet = Verlet(R0[0], R0[1], HarmonicOscillator, 0.0, 10 \* 2.0 \* np.pi, int(step))

VerletRange.append(tempVerlet)

EnergyRange.append(Energy(tempVerlet[0]))

plt.clf()

plt.plot(stepsizeRange, EnergyRange)

plt.xlabel('Stepsize')

plt.ylabel('Energy')

plt.title('Energy as a function of stepsize')

plt.savefig('Homework 6 - Energy per stepsize.png')

*Results*

(See attached graphs)

**Bound States in a Potential Well**

*Code*

import numpy as np

import matplotlib.pyplot as plt

#Constants

m = 1.0

hbar = 1.0

e = 1.0

L = 1.0

def V(c, x):

return c \* ((x/L)\*\*2 - (x/L))

def FVector(r, x, c, E):

psi = r[0]

phi = r[1]

fpsi = phi

fphi = (2 \* m / hbar\*\*2) \* (V(c, x) - E) \* psi

return np.array([fpsi, fphi])

def RungeKutta(r0, x0, xf, N, f, c, E):

h = (xf - x0) / N

R = np.zeros((N + 1, 2))

R[0] = r0

X = np.arange(x0, xf, h)

for i in range(len(X)):

k1 = h \* f(R[i], X[i], c, E)

k2 = h \* f(R[i] + 0.5 \* k1, X[i] + 0.5 \* h, c, E)

k3 = h \* f(R[i] + 0.5 \* k2, X[i] + 0.5 \* h, c, E)

k4 = h \* f(R[i] + k3, X[i] + h, c, E)

R[i + 1] = R[i] + (k1 + 2 \* k2 + 2 \* k3 + k4) / 6.0

return R

def RKforanE(E, c):

return RungeKutta(np.array([0.0, 1.0]), 0.0, L, 1000, FVector, c, E)[-1][0]

def SecantMethod(a, b, f, tol, c):

x1 = 0.0

x2 = f(a, c)

counter = 0

while (abs(a - b) > tol) & (counter < 1000):

x1, x2 = x2, f(b, c)

a, b = b, b - x2 \* (b - a) / (x2 - x1)

counter += 1

if counter == 1000:

print 'Divergence, no more nodes.'

return None

return b

groundStateEnergy = SecantMethod(0.0, 1.0, RKforanE, 1.0/1000, 1.0)

groundEnergy2 = SecantMethod(10.0, 11.0, RKforanE, 1.0/1000, 1.0)

groundEnergy3 = SecantMethod(1000.0, 1001.0, RKforanE, 1.0/1000, 1.0)

print 'Ground State Energy:', groundStateEnergy

print 'groundEnergy2:', groundEnergy2

print 'groundEnergy3:', groundEnergy3

groundWavefunction = np.flipud(np.rot90(RungeKutta(np.array([0.0, 1.0]), 0.0, 1.0, 1000, FVector, 1.0, groundStateEnergy)))[0][:-1]

wavefunction2 = np.flipud(np.rot90(RungeKutta(np.array([0.0, 1.0]), 0.0, 1.0, 1000, FVector, 1.0, groundEnergy2)))[0][:-1]

wavefunction3 = np.flipud(np.rot90(RungeKutta(np.array([0.0, 1.0]), 0.0, 1.0, 1000, FVector, 1.0, groundEnergy3)))[0][:-1]

plt.clf()

plt.plot(np.linspace(0.0, 1.0, 1000), groundWavefunction, label = 'c = 1 - Ground State')

plt.plot(np.linspace(0.0, 1.0, 1000), wavefunction2, label = 'c = 1 - First Excited Level')

plt.plot(np.linspace(0.0, 1.0, 1000), wavefunction3, label = 'c = 1 - Another Excited Level')

plt.xlabel('X')

plt.ylabel('Energy')

plt.legend(loc = 1)

plt.title('Wavefunction in a Box')

plt.savefig('Homework 6 Problem 2 - Ground State, c = 1.png')

*Results*

Ground State Energy: 4.71743779239

groundEnergy2: 44.2409307445

groundEnergy3: 967.054367524

It is evident that in this situation, for c = 1, the wavefunction I have has an infinite number of excited levels, all just harmonics of the bas function.

I was unfortunately unable to figure out how to generalize the example problem in the book to the problem we were given – and by the time I realized this it was too late to come to office hours.