Moran_HW1

September 14, 2017

Shannon Moran Physics 514 Homework 1 Due: 9/14/2017

0.1 Import required packages

```
In [1]: %matplotlib inline
        import numpy as np
        import math
        import matplotlib.pyplot as plt
```

0.2 1: ODE Integrators

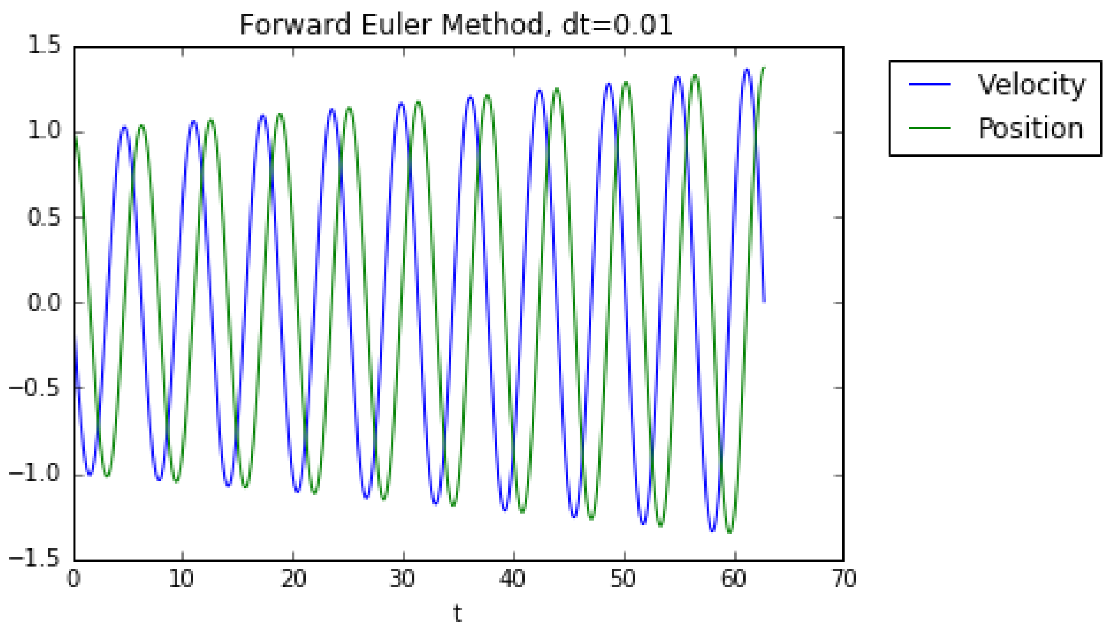
```
In [26]: # 1) Forward Euler Algorithm
         def forward_euler(settings,t_run,dt):
             (m,k,v_0,x_0) = settings
             x,v = [],[]
             v.append(v_0), x.append(x_0)

             n = 0
             for t in np.arange(dt,t_run,dt):
                 v.append(v[n] + (dt)*(-k/m)*(x[n]))
                 x.append(x[n] + (dt)*(v[n]))
                 n += 1

             plt.plot(np.arange(0,t_run,dt),v,label="Velocity")
             plt.plot(np.arange(0,t_run,dt),x,label="Position")
             plt.legend(bbox_to_anchor=(1.35, 1))
             plt.title("Forward Euler Method, dt=%r" %dt)
             plt.xlabel("t")
             plt.show()

             # Check error
             v_exact = -np.sin(np.arange(0,t_run,dt))
             x_exact = np.cos(np.arange(0,t_run,dt))

             plt.plot(np.arange(1/2,t_run+1/2,dt),v-v_exact,label="Velocity",color="red")
             plt.plot(np.arange(0,t_run,dt),x-x_exact,label="Position",color="red")
             plt.legend(bbox_to_anchor=(1.35, 1))
             plt.title("Error: Forward Euler Method, dt=%r" %dt)
             plt.xlabel("t")
             plt.show()


         # 2) Backward Euler Algorithm
```

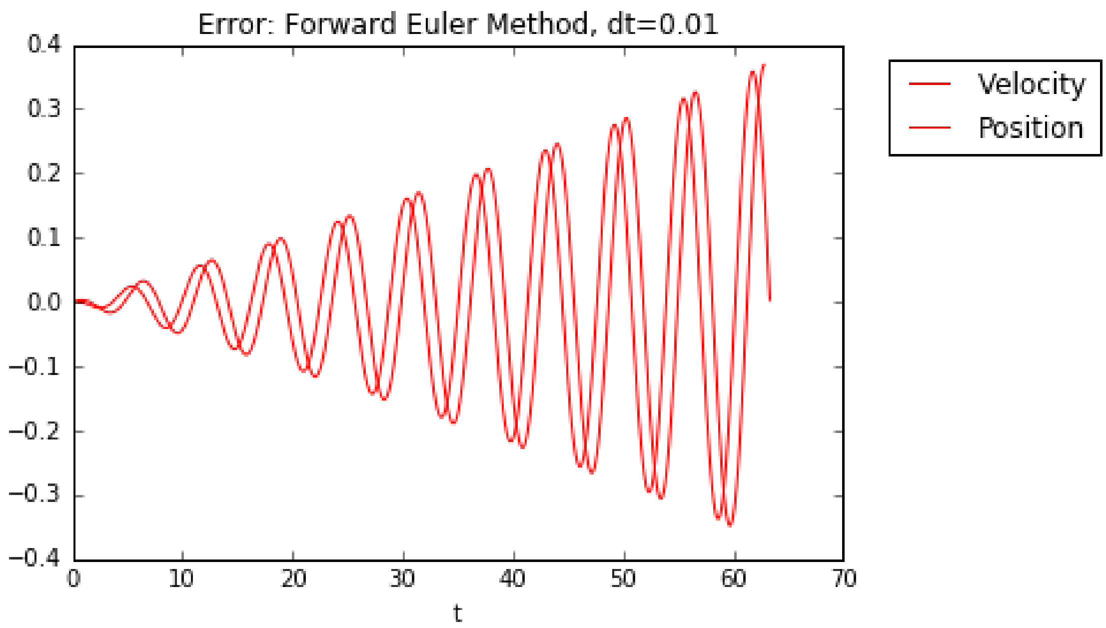
```
def backward_euler(settings,t_run,dt):
    (m,k,v_0,x_0) = settings
    x,v,x_est = [],[],[]
    v.append(v_0), x.append(x_0), x_est.append(0)

    n = 1
    for t in np.arange(dt,t_run,dt):
        x_est.append(x[n-1] + dt*v[n-1])
        v.append(v[n-1] + (dt)*(-k/m)*(x_est[n]))
        x.append(x[n-1] + (dt)*(v[n]))
        n += 1

    plt.plot(np.arange(0,t_run,dt),v,label="Velocity")
    plt.plot(np.arange(0,t_run,dt),x,label="Position")
    plt.legend(bbox_to_anchor=(1.35, 1))
    plt.title("Backward Euler Method, dt=%r" %dt)
    plt.xlabel("t")
    plt.show()

    # Check error
    v_exact = -np.sin(np.arange(0,t_run,dt))
    x_exact = np.cos(np.arange(0,t_run,dt))

    plt.plot(np.arange(1/2,t_run+1/2,dt),v-v_exact,label="Velocity",color="red")
    plt.plot(np.arange(0,t_run,dt),x-x_exact,label="Position",color="orange")
    plt.legend(bbox_to_anchor=(1.35, 1))
    plt.title("Error: Backward Euler Method, dt=%r" %dt)
    plt.xlabel("t")
    plt.show()


# 3) RK4 Runge Kutta
def runge_kutta(settings,t_run,dt):
    (m,k,v_0,x_0) = settings
    x,v,x_est = [],[],[]
    v.append(v_0), x.append(x_0)

    n = 0
    for t in np.arange(dt,t_run,dt):
        k1x = (dt)*(v[n])
        k1v = (dt)*(-k/m)*(x[n])
        k2x = (dt)*(v[n]+k1v/2)
        k2v = (dt)*(-k/m)*(x[n]+k2x/2)
        k3x = (dt)*(v[n]+k2v/2)
        k3v = (dt)*(-k/m)*(x[n]+k3x/2)
        k4x = (dt)*(v[n]+k3v)
        k4v = (dt)*(-k/m)*(x[n]+k4x)
        v.append(v[n] + (1/6)*(k1v+2*k2v+2*k3v+k4v))
        x.append(x[n] + (1/6)*(k1x+2*k2x+2*k3x+k4x))
        n += 1

    plt.plot(np.arange(0,t_run,dt),v,label="Velocity")
    plt.plot(np.arange(0,t_run,dt),x,label="Position")
    plt.legend(bbox_to_anchor=(1.35, 1))
```

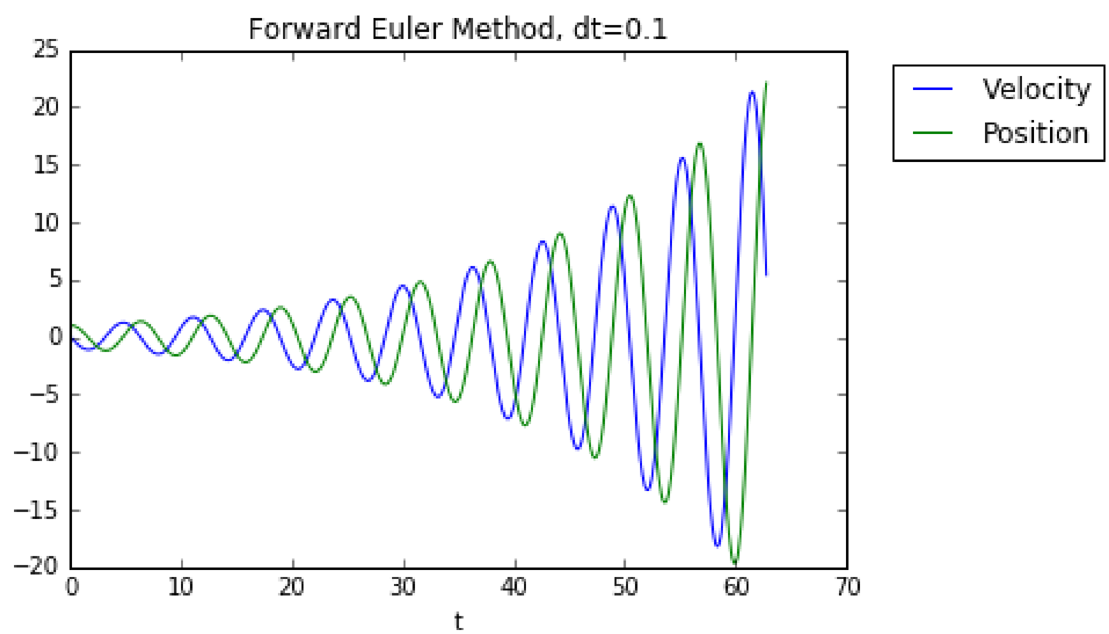
```
    plt.title("RK4 Runga Kutta Method, dt=%r" %dt)
    plt.xlabel("t")
    plt.show()

    # Check error
    v_exact = -np.sin(np.arange(0,t_run,dt))
    x_exact = np.cos(np.arange(0,t_run,dt))

    plt.plot(np.arange(1/2,t_run+1/2,dt),v-v_exact,label="Velocity",color="red")
    plt.plot(np.arange(0,t_run,dt),x-x_exact,label="Position",color="orange")
    plt.legend(bbox_to_anchor=(1.35, 1))
    plt.title("Error: RK4 Method, dt=%r" %dt)
    plt.xlabel("t")
    plt.show()

# 4) Leapfrog
def leapfrog(settings,t_run,dt):
    (m,k,v_0,x_0) = settings
    x,v = [],[]
    x.append(x_0)
    v.append(v_0 + (dt)*(1/2)*(-k/m)*x[0])

    n = 1
    for t in np.arange(dt,t_run,dt):
        # Indices for velocity are actually for each 1/2 dt; corrected in plot
        v.append(v[n-1] + (dt)*(-k/m)*x[n-1])
        x.append(x[n-1] + (dt)*(v[n]))
        n += 1

    plt.plot(np.arange(0,t_run,dt)+(1/2),v,label="Velocity")
    plt.plot(np.arange(0,t_run,dt),x,label="Position")
    plt.legend(bbox_to_anchor=(1.35, 1))
    plt.title("Leapfrog Method, dt=%r" %dt)
    plt.xlabel("t")
    plt.show()

    # Check error
    v_exact = -np.sin(np.arange(0,t_run,dt))
    x_exact = np.cos(np.arange(0,t_run,dt))

    plt.plot(np.arange(1/2,t_run+1/2,dt),v-v_exact,label="Velocity",color="red")
    plt.plot(np.arange(0,t_run,dt),x-x_exact,label="Position",color="orange")
    plt.legend(bbox_to_anchor=(1.35, 1))
    plt.title("Error: Leapfrog Method, dt=%r" %dt)
    plt.xlabel("t")
    plt.show()
```

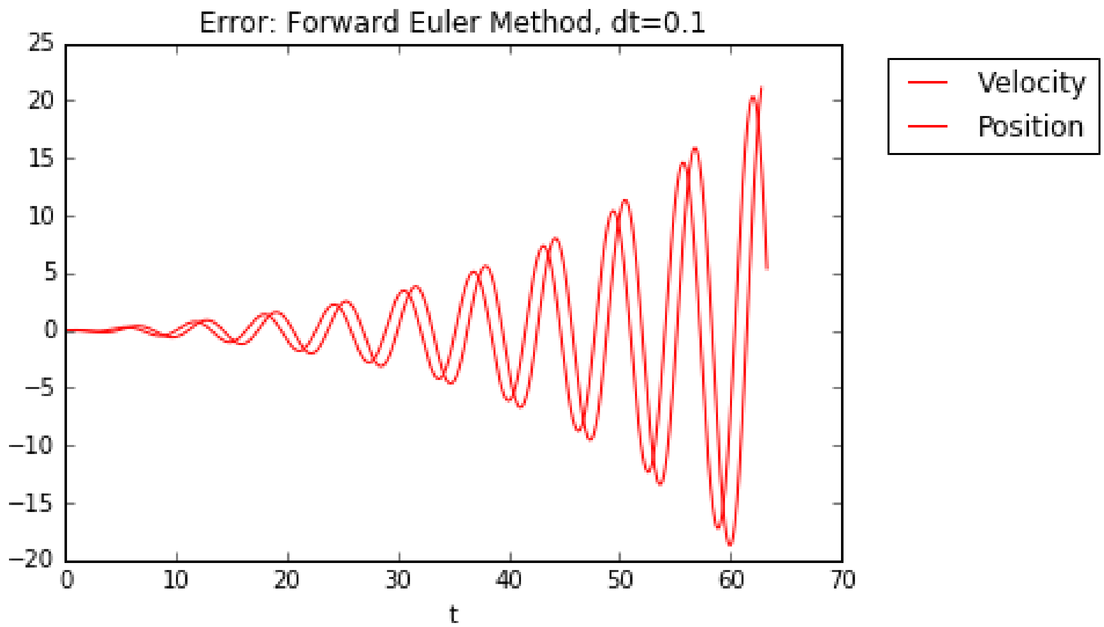
In [27]:
```
# Assumptions given in problem statement.
m = 1
k = 1
v_0 = 0
x_0 = 1
assumptions = (m,k,v_0,x_0)
```

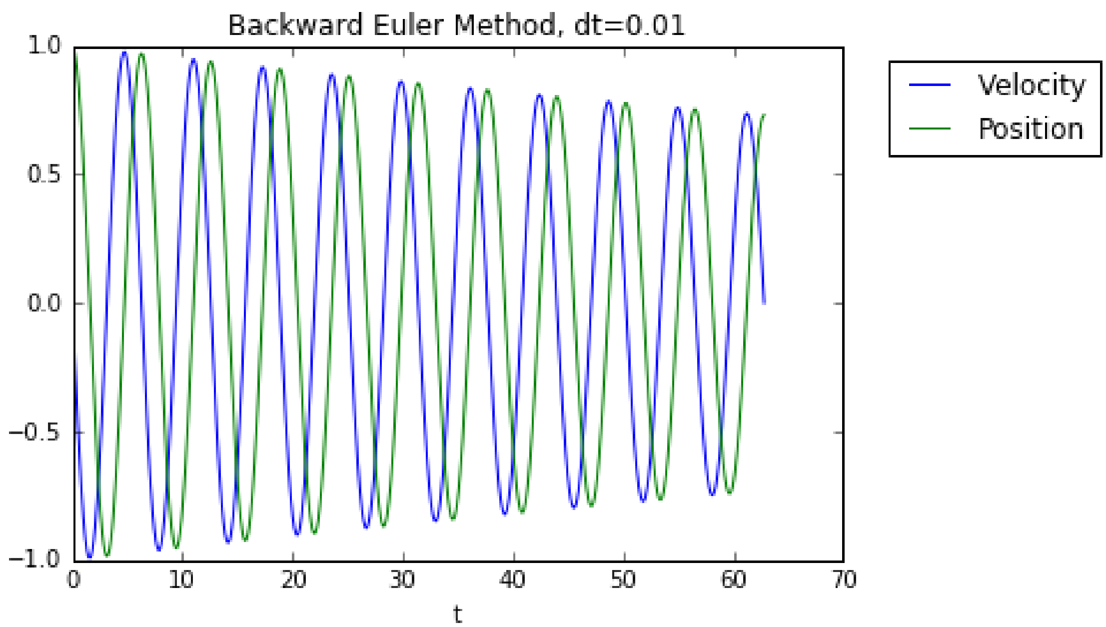
```
# Run time for 10 periods, as prescribed in the problem statement.
t_run = 20*math.pi*math.sqrt(m/k)

# Call all methods and output results.
forward_euler(assumptions,t_run,0.01)
forward_euler(assumptions,t_run,0.1)
backward_euler(assumptions,t_run,0.01)
backward_euler(assumptions,t_run,0.1)
runge_kutta(assumptions,t_run,0.01)
runge_kutta(assumptions,t_run,0.1)
leapfrog(assumptions,t_run,0.01)
leapfrog(assumptions,t_run,0.1)
```

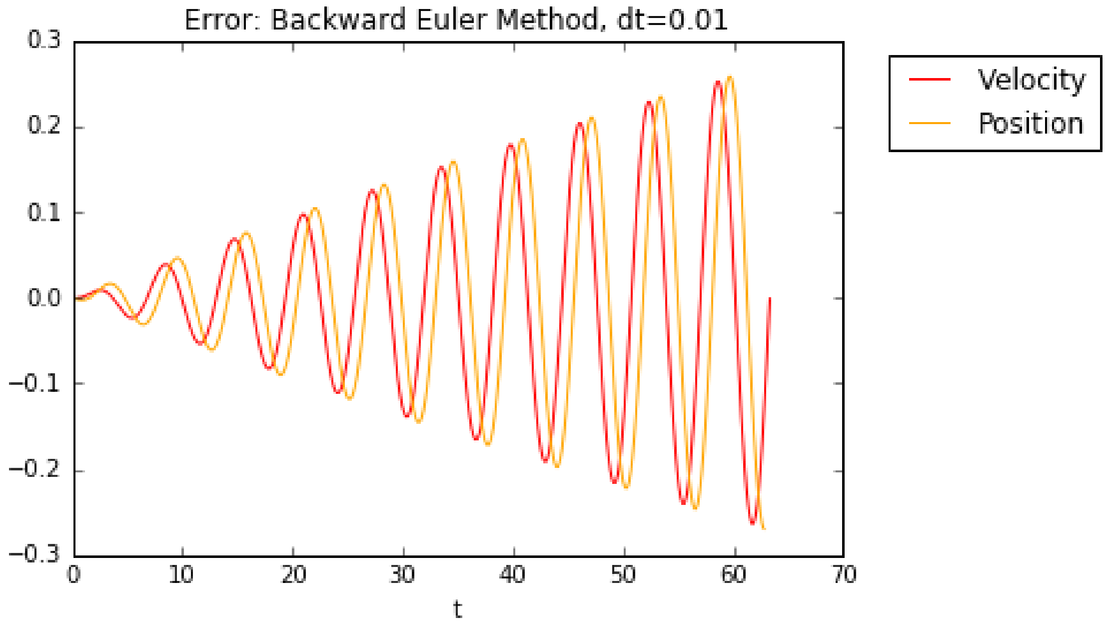


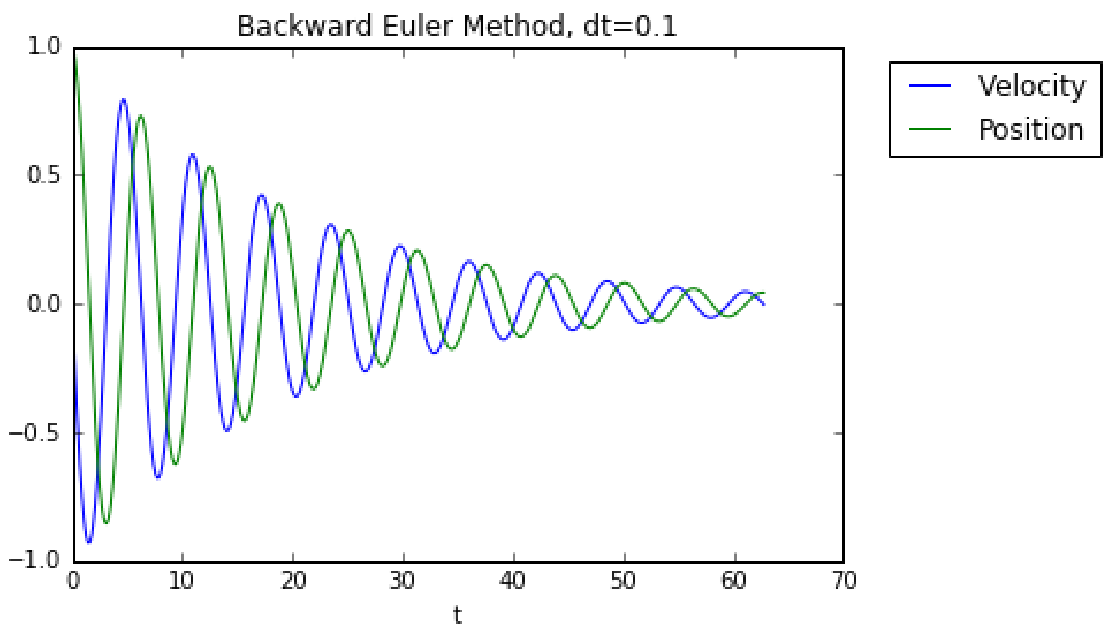

Error: Forward Euler Method, dt=0.01
Velocity
Position
t

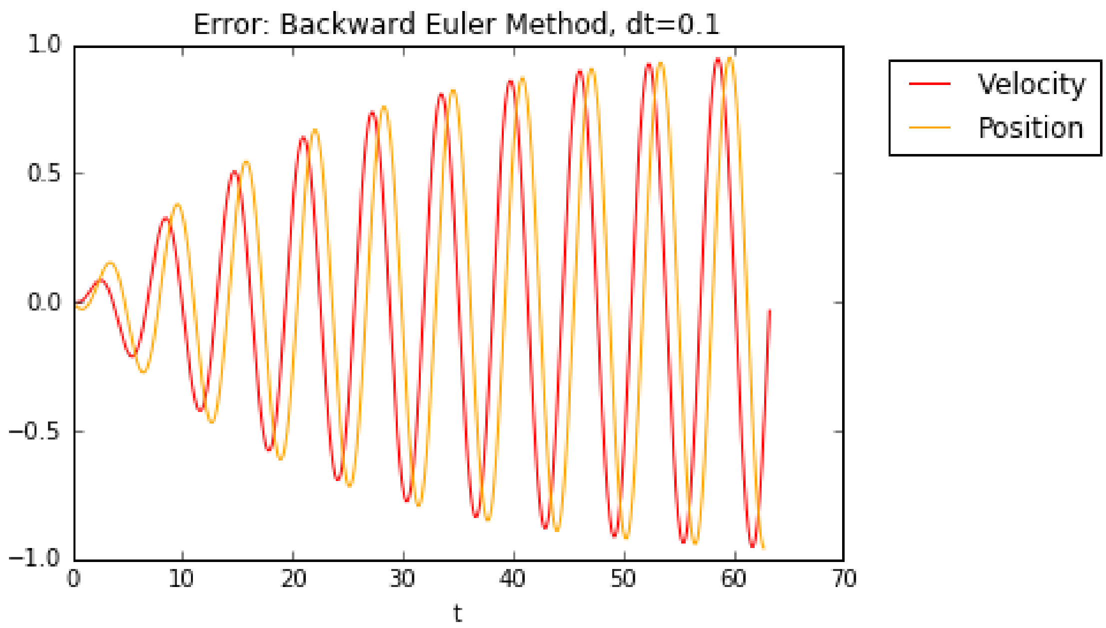

Forward Euler Method, dt=0.1
Velocity
Position
t

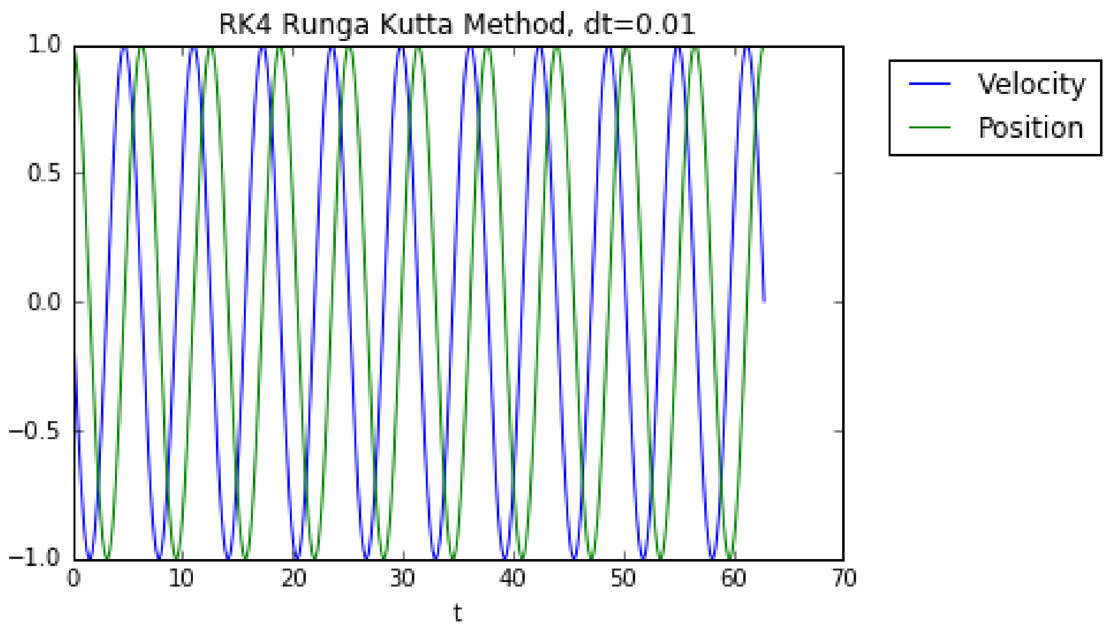

Error: Forward Euler Method, dt=0.1
Velocity
Position
t

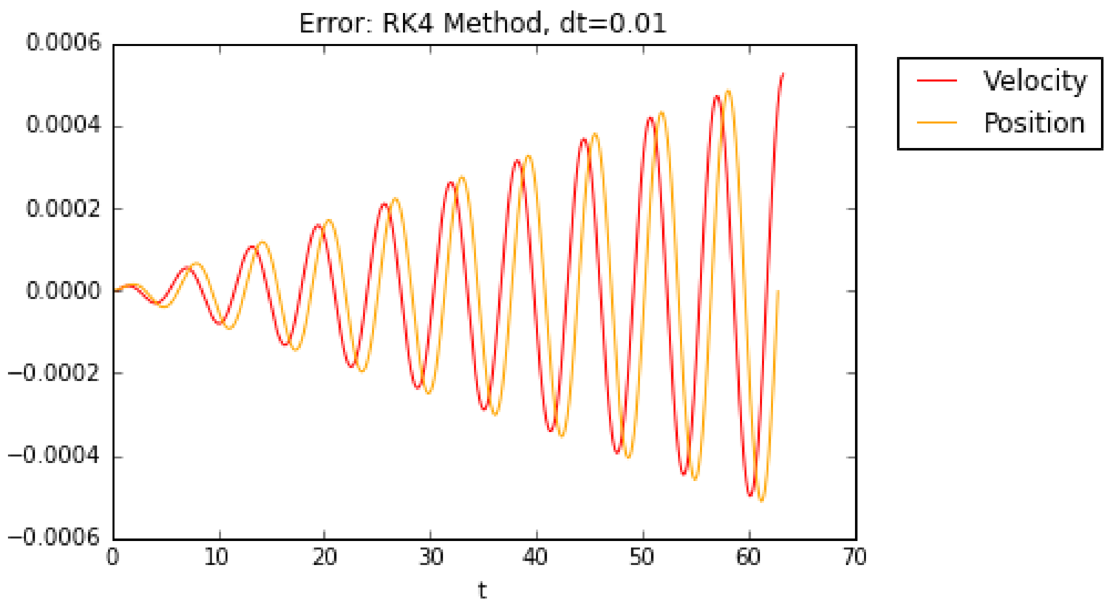

Backward Euler Method, dt=0.01
Velocity
Position
t

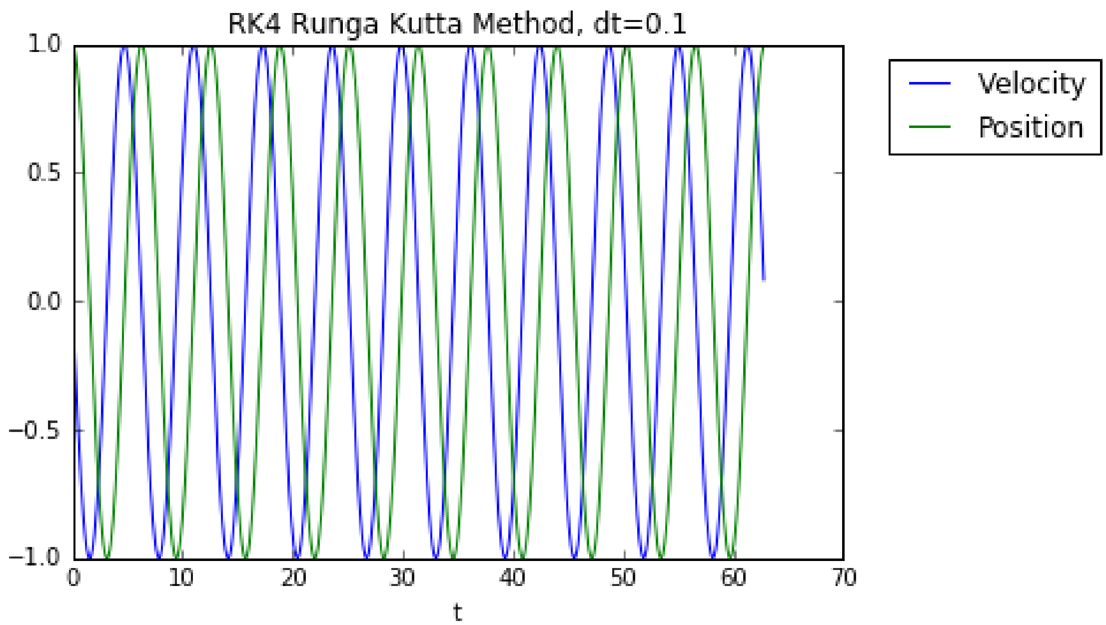

Error: Backward Euler Method, dt=0.01
Velocity
Position
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
0
10
20
30
40
50
60
70
t

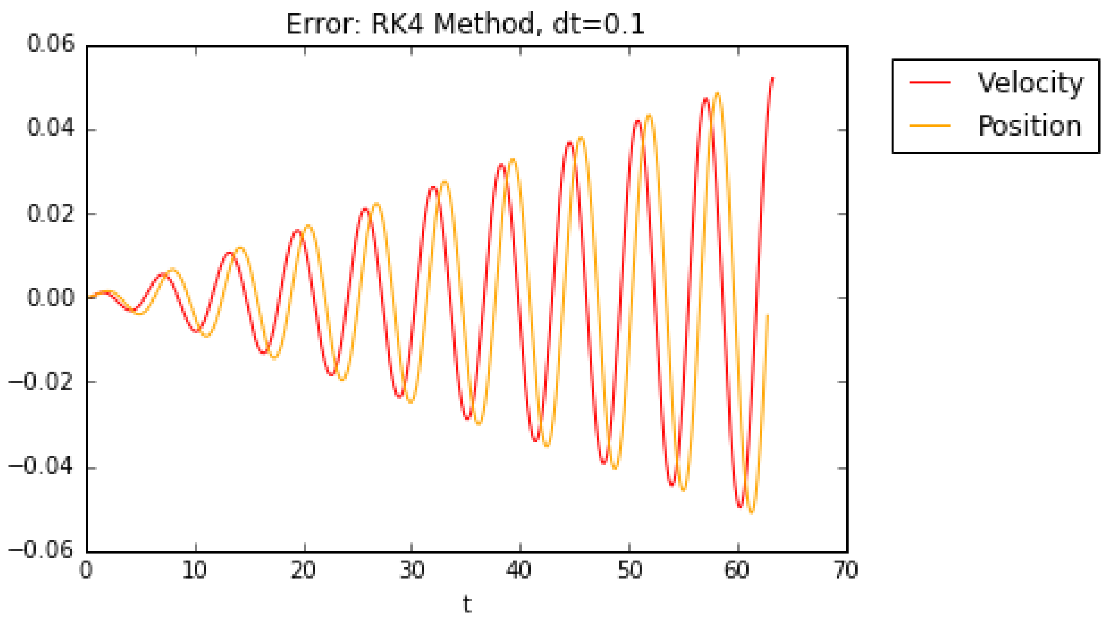

Backward Euler Method, dt=0.1
Velocity
Position
1.0
0.5
0.0
−0.5
−1.0
0
10
20
30
40
50
60
70
t

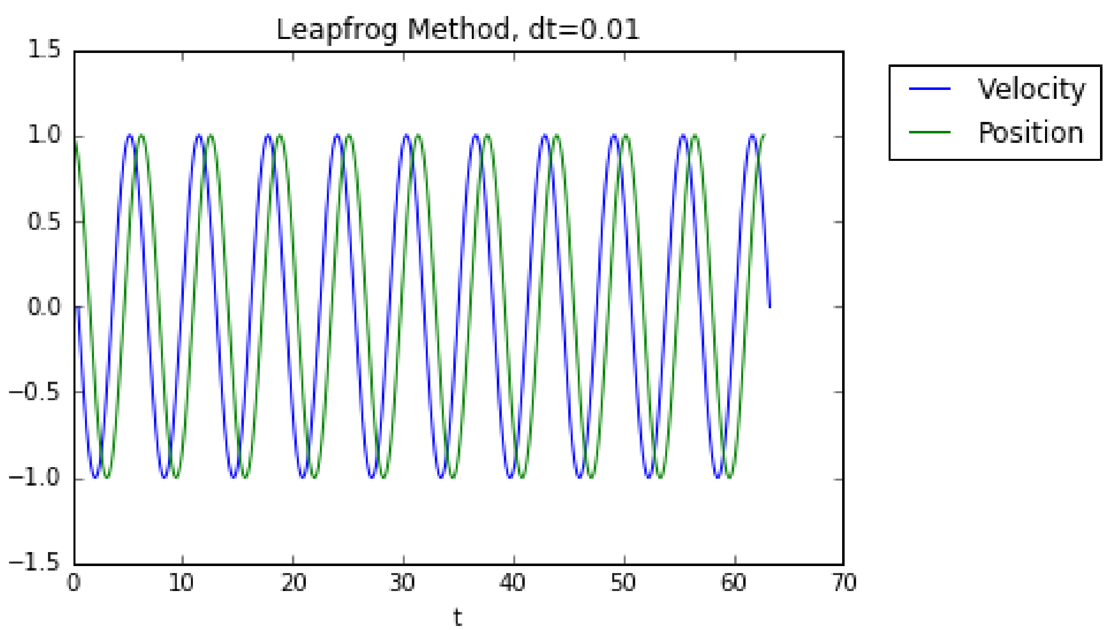

Error: Backward Euler Method, dt=0.1
1.0
0.5
0.0
−0.5
−1.0
0
10
20
30
40
50
60
70
t
Velocity
Position

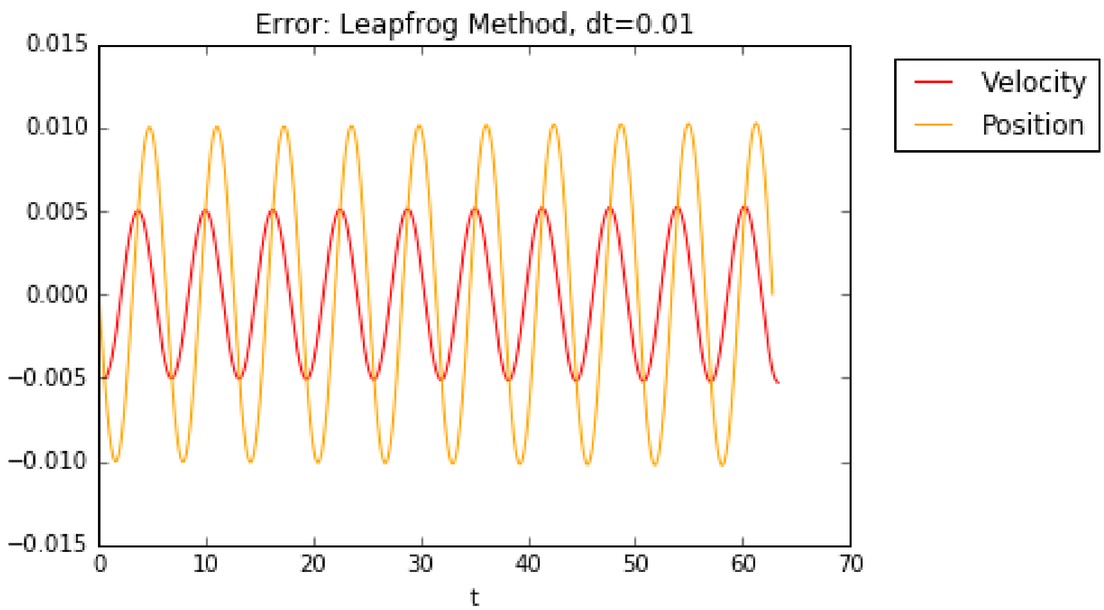

RK4 Runga Kutta Method, dt=0.01
1.0
0.5
0.0
−0.5
−1.0
0
10
20
30
40
50
60
70
t
Velocity
Position

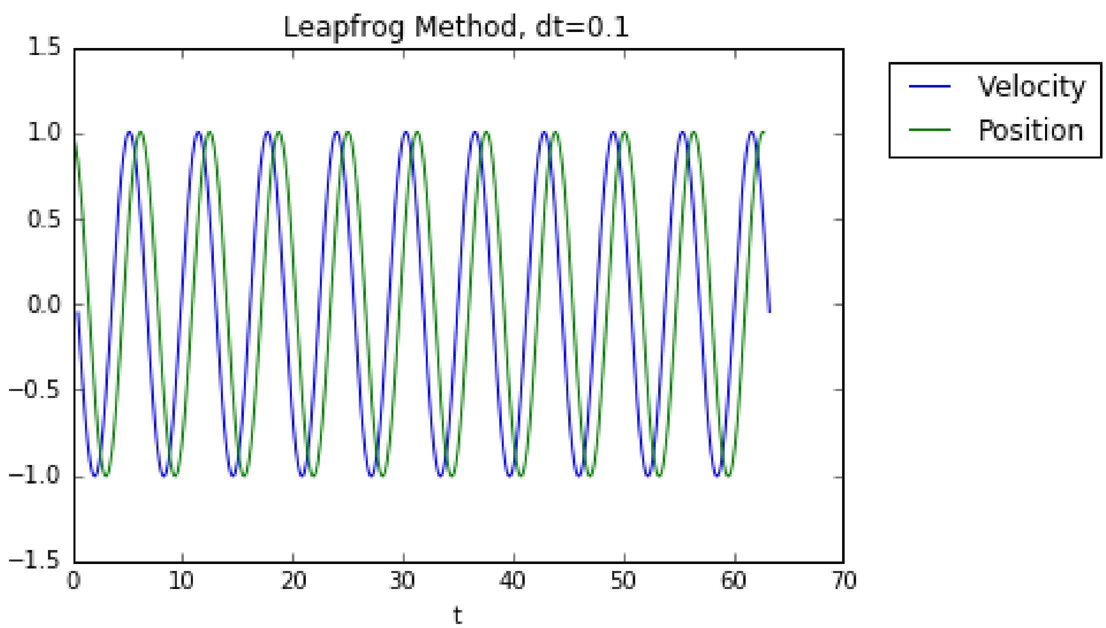

Error: RK4 Method, dt=0.01
0.0006
0.0004
0.0002
0.0000
−0.0002
−0.0004
−0.0006
0
10
20
30
40
50
60
70
t
Velocity
Position

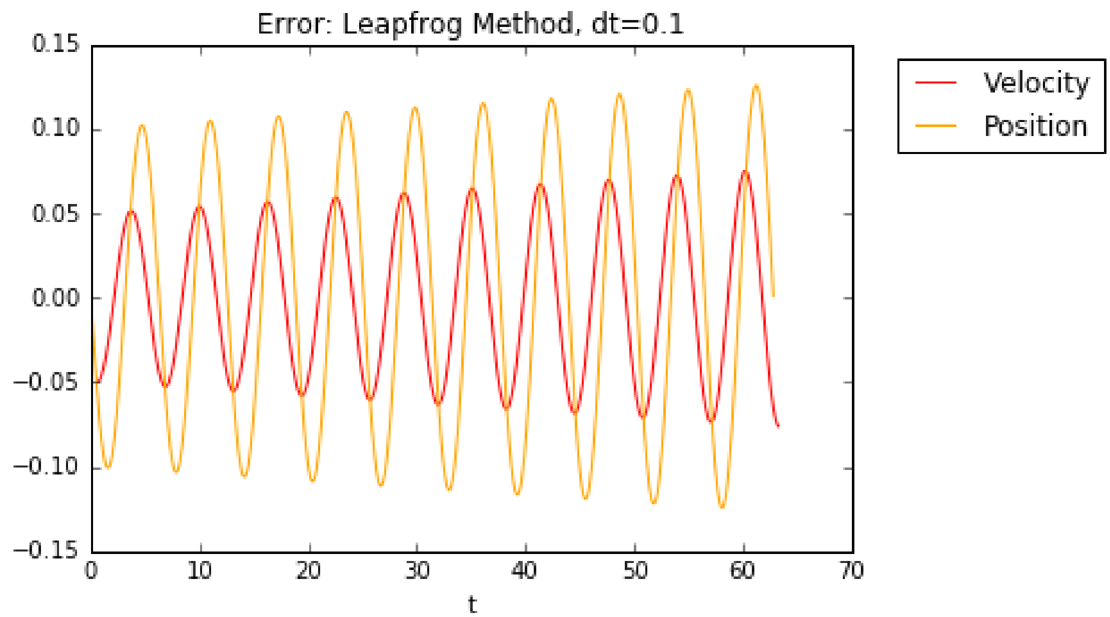

RK4 Runga Kutta Method, dt=0.1
1.0
0.5
0.0
−0.5
−1.0
0
10
20
30
40
50
60
70
t
Velocity
Position


Error: RK4 Method, dt=0.1
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
0
10
20
30
40
50
60
70
t
Velocity
Position


Leapfrog Method, dt=0.01
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
10
20
30
40
50
60
70
t
Velocity
Position


Error: Leapfrog Method, dt=0.01
Velocity
Position
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
0
10
20
30
40
50
60
70
t


Leapfrog Method, dt=0.1
Velocity
Position
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
10
20
30
40
50
60
70
t



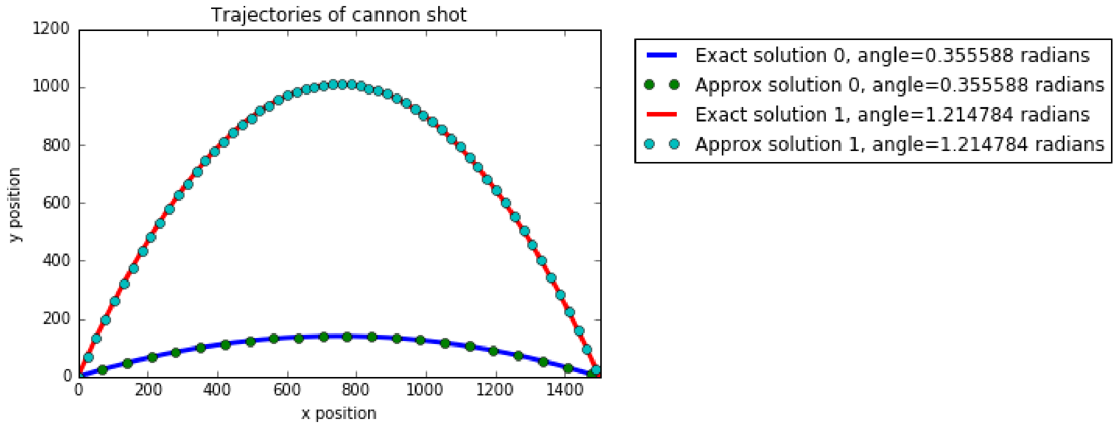
Analysis of convergence and stability: As seen in the above, neither the Forward nor the Backward Euler successfully converge on the correct solution; furthermore, their error grows over time. While Runge-Kutta appears to converge successfully, very small error does accrue over time (i.e. unstable). For practical purposes, though, this would likely converge sufficiently to be useful. Leapfrog also converges (for practical applciations), and is also slightly unstable, though its error grows at a slower rate than RK4.

0.3 2: Shooting

In [24]:
```
# Regula Falsi Method: Finding root of position function y between angle guesses g0 and g1
def regula_falsi(g0,g1,settings):
    yapprox = 100 # initial seed for y appro
    while abs(yapprox)>2:
        _,y0,_ = shoot(g0,settings)
        _,y1,_ = shoot(g1,settings)
        # Linearize the function
        angle_approx = (g0*y1[-2]-g1*y0[-2])/(y1[-2]-y0[-2])
        _,approx,_ = shoot(angle_approx,settings)
        yapprox = approx[-2]
        if np.sign(yapprox)==np.sign(y0[-2]): g0 = angle_approx
        elif np.sign(yapprox)==np.sign(y1[-2]): g1 = angle_approx
        else: pass
    return angle_approx

# Forward Euler: Iterate y values forward until x value is xf
def shoot(angle,settings):
    (dt,d_target,v_0) = settings
    y,vy,x = [],[],[]
    y.append(0), x.append(0)

    n,t = 0,dt
    vx = v_0*math.cos(angle)
```

```
    vy.append(v_0*math.sin(angle))
    while (x[n] < d_target):
        x.append(x[n] + dt*vx)
        vy.append(vy[n] + (-9.8)*dt)
        y.append(y[n] + dt*vy[n])
        t += dt
        n += 1
    return x,y,t

# Analytical solution for the parabolic trajectory.
def parabola(angle,settings):
    (dt,d_target,v_0) = settings
    x,y,t = [0],[0],[0]
    while x[-1]<d_target:
        t.append(t[-1]+dt)
        x.append(v_0*math.cos(angle)*t[-1])
        y.append(v_0*math.sin(angle)*t[-1]+(-9.8/2)*t[-1]**2)
    return x,y

# Iterates over space of guessed angles to find roots
def root_finder(guesses,settings):
    x_values, y_values = [],[]
    n,c = 0,0
    for angle in guesses:
        x,y,t = shoot(angle,settings)
        x_values.append(x[-2])
        y_values.append(y[-2])
        # If the final y value changes signs, look for a root and plot the resulting trajector
        if (n>1) and (np.sign(y_values[-1]) != np.sign(y_values[-2])):
            angle_approx = regula_falsi(guesses[n-1],guesses[n],settings)
            x,y,t = shoot(angle_approx,settings)
            x_exact, y_exact = parabola(angle_approx,settings)
            plt.plot(x_exact,y_exact,label="Exact solution %d, angle=%f radians" %(c,angle_appr
            plt.plot(x[0::50],y[0::50],label="Approx solution %d, angle=%f radians" %(c,angle_a
            c += 1
        else: pass
        n += 1
    plt.legend(bbox_to_anchor=(2, 1))
    plt.title("Trajectories of cannon shot")
    plt.axis([0, 1500, 0, 1200])
    plt.xlabel("x position")
    plt.ylabel("y position")
    plt.show()
```

In [25]:
```
# Given values in problem statement.
dt = 0.01
d_target = 1500 #m
v_0 = 150 #m/s
settings = (dt,d_target,v_0)

# Check all angles from 0-90 degrees for roots.
guesses = np.linspace(0,math.pi/2.1,10)
root_finder(guesses, settings)
```


Trajectories of cannon shot
1200
1000
800
600
400
200
0
y position
0
200
400
600
800
1000
1200
1400
x position
Exact solution 0, angle=0.355588 radians
Approx solution 0, angle=0.355588 radians
Exact solution 1, angle=1.214784 radians
Approx solution 1, angle=1.214784 radians