Assignment_1_Notebook - Noah Steinberg - Computational Physics

September 13, 2017

1) We are to solve the simple harmonic oscillator with differential equation:

$$\ddot{x} = -\omega^2 x$$

with $\omega^2 = \frac{k}{m}$

We will investigate four different methods, forward euler, backward euler, runge kutta, and leapfrog. The following function solves the S.H.O. It takes in the initial and final time, initial position and velocity, ω^2, and the integration method. It outputs 2 arrays: [t,x(t)], and [t,v(t)]. Each element of the previous outputs is also an array.

In [10]:
```
import matplotlib.pyplot as plt
import math as math
import numpy as np


def integrate_SHO(t_0, t_f, x_0, v_0, steps, omega_squared, method):

    # Create empty arrays for x(t), v(t), and t
    x,v,t = [], [], []
    t_shift = [] # For the leap frog method, v(t) is evaluated at t0/2 + del_t*(n+1/2)

    # Define the interval and time step
    interval = (t_f - t_0)
    del_t = interval/steps

    x_t = []
    v_t = []

    #Append initial conditions
    x.append(x_0)
    v.append(v_0)
    t.append(t_0)
    t_shift.append(t_0 + del_t/2)

    #Choose a method to solve differential equation
```

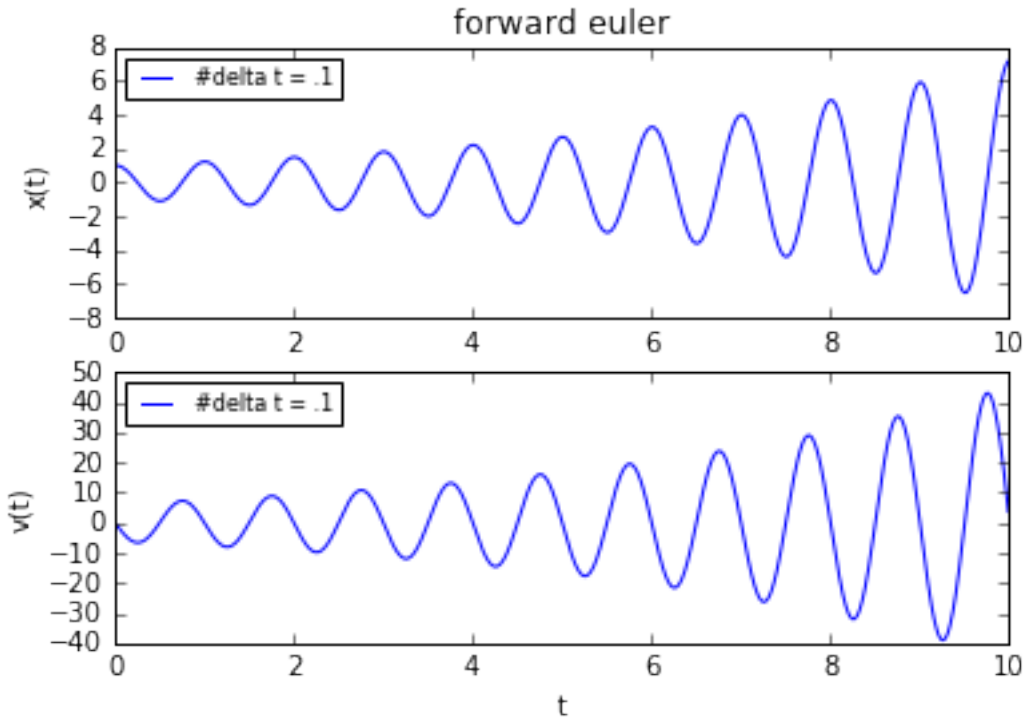
```
    if method == 'forward euler':
        for n in range(1,steps + 1):
            t.append(t_0 + n*del_t)
            x.append(x[n-1] + v[n-1]*del_t)
            v.append(v[n-1] + (-omega_squared*x[n-1])*del_t)

    if method == 'backward euler':
        for n in range(1,steps + 1):
            t.append(t_0 + n*del_t)
            x.append(x[n-1] + del_t*((v[n-1]
            - omega_squared*x[n-1]*del_t)/(1 + omega_squared*(del_t**2))))
            v.append((v[n-1]
            - (omega_squared)*x[n]*del_t)/(1 + (omega_squared)*(del_t**2)))

    if method == 'runge kutta':
        for n in range(1, steps + 1):
            t.append(t_0 + n*del_t)

            xk2 = del_t*(v[n-1] + (-omega_squared)*x[n-1]*del_t/2)
            vk2 = del_t*(-omega_squared)*(x[n-1] + v[n-1]*del_t/2)

            x.append(x[n-1] + xk2)
            v.append(v[n-1] + vk2)

    if method == 'leapfrog':

        v = [] #We will evaluate v half of a step ahead of x, so clear v for now
        v_half = v_0 + (1/2)*del_t*(-(x_0)*omega_squared)
        v.append(v_half)

        for n in range(1,steps + 1):
            t.append(t_0 + n*del_t)
            t_shift.append(t_shift[0] + n*del_t)
            v.append(v[n-1] + (-omega_squared*x[n-1])*del_t)
            x.append(x[n-1] + v[n]*del_t)

    # Make arrays of time vs position, and time vs velocity
    #(make sure to use offset time for velocity if using leapfrog method)
    x_t = [t,x]
    if method == 'leapfrog':
        v_t = [t_shift, v]
    else:
        v_t = [t, v]

    return x_t, v_t
```

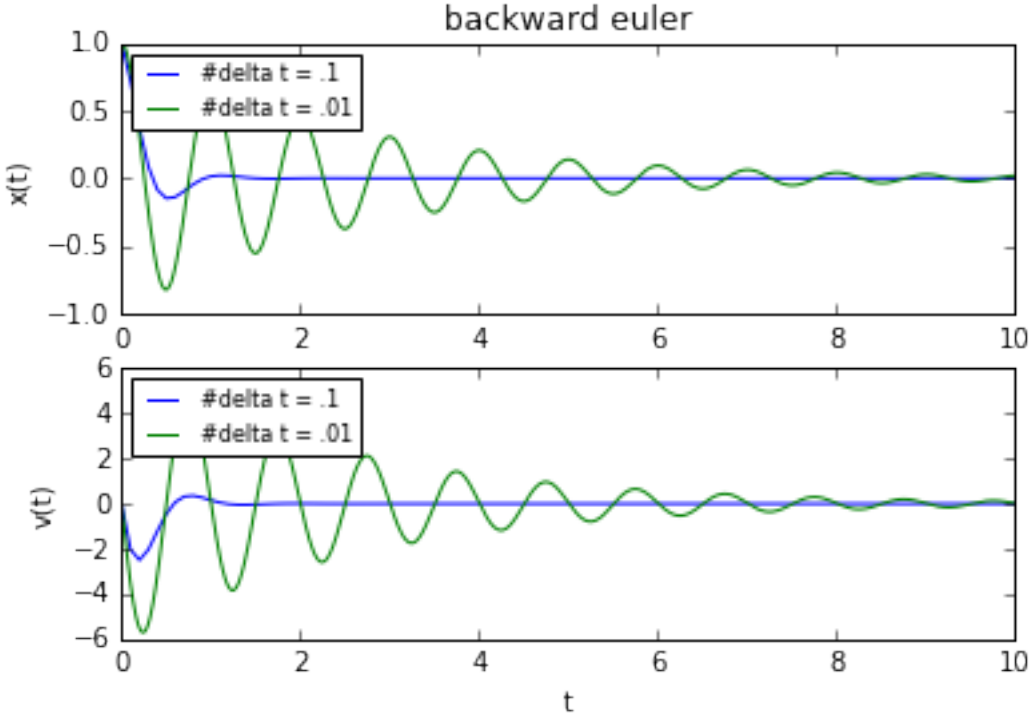
This function simply makes the plots from the output of integrate_SHO. It takes in an integration method and plots x(t) and v(t) for two different time steps, $\Delta t = .1$ and $\Delta t = .01$.

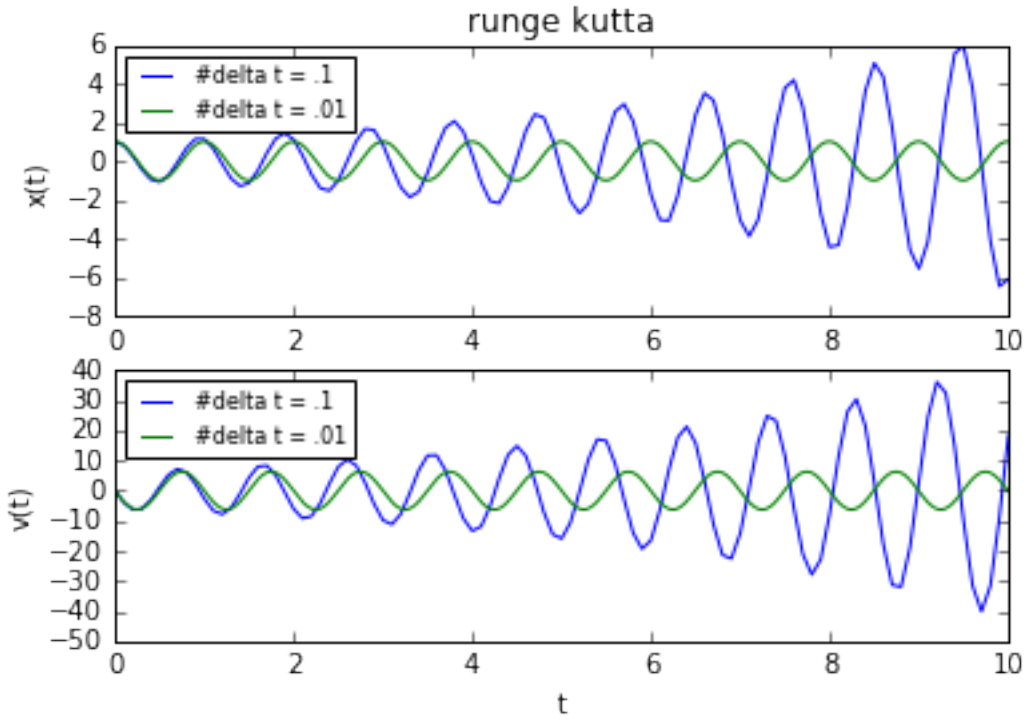
```
In [12]: def make_plots(method):

             # Set $\omega^2$ = (2*pi)^2 so that the period T = (2pi/omega) = 1
             # and run the given integration method for del_t = .1, and del_t = .01
             result_small_steps = integrate_SHO(0, 10, 1, 0, 100, (2*math.pi)**2, method)
             result_large_steps = integrate_SHO(0, 10, 1, 0, 1000, (2*math.pi)**2, method)

             plt.subplot(2,1,1)
             if (method == 'forward euler'):
                 plt.plot(result_large_steps[0][0],
                          result_large_steps[0][1], label="#delta t = .01")
             else:
                 plt.plot(result_small_steps[0][0],
                          result_small_steps[0][1], label="#delta t = .1")
                 plt.plot(result_large_steps[0][0],
                          result_large_steps[0][1], label="#delta t = .01")
             plt.ylabel('x(t)')
             plt.title(method)
             plt.legend(['#delta t = .1', '#delta t = .01'], loc="upper left", prop={'size': 8})

             plt.subplot(2,1,2)
             if (method == 'forward euler'):
                 plt.plot(result_large_steps[1][0],
                          result_large_steps[1][1], label="#delta t = .01")
             else:
                 plt.plot(result_small_steps[1][0],
                          result_small_steps[1][1], label="#delta t = .1")
                 plt.plot(result_large_steps[1][0],
                          result_large_steps[1][1], label="#delta t = .01")
             plt.ylabel('v(t)')
             plt.legend(['#delta t = .1', '#delta t = .01'], loc="upper left", prop={'size': 8})

             plt.xlabel('t')
             plt.show()
```

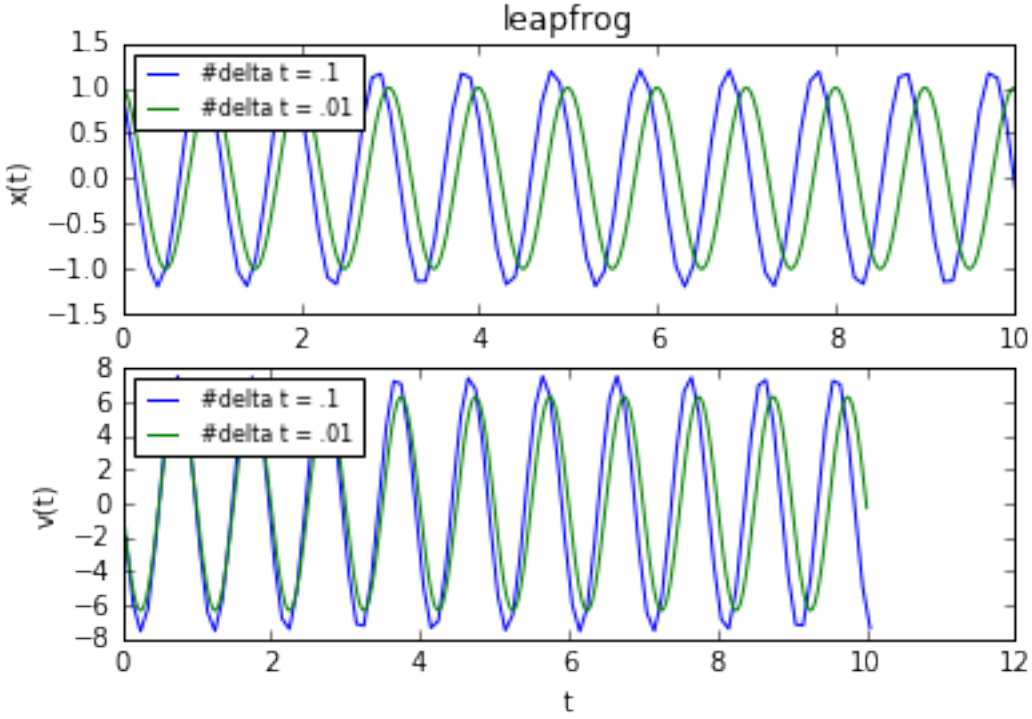
We then loop over all four methods and plot the outputs.

```
In [13]: methods = ('forward euler', 'backward euler', 'runge kutta', 'leapfrog')

         for method in methods:
             make_plots(method)
```


forward euler
#delta t = .1
x(t)
#delta t = .1
v(t)
t


backward euler
#delta t = .1
#delta t = .01
x(t)
#delta t = .1
#delta t = .01
v(t)
t


runge kutta
#delta t = .1
#delta t = .01
x(t)
v(t)
t


leapfrog
#delta t = .1
#delta t = .01
x(t)
v(t)
t

We see that the forward euler is unstable and does not converge, this is typical for application of the forward euler method to an oscillation problem. I did not plot the forward euler method with a step size of .1s as the curve diverged extremely quickly and the curve with a step size of .01s could not be seen. The backwards euler method is stable, in that it does not blow up to infinity, but it does not converge to the correct solution. Both of these methods behavior improves with smaller step size, as expected.

Runge kutta and leapfrog both have solutions which converge and are stable when the step size is $\Delta t = .01$. It is interesting to note that the runge kutta method is not stable and diverges with too large a step size, in contrast to the leapfrog method whose solution is stable for the same step size.

For each method one could plot the error versus the step size to verify the behavior of each method. This was not done.

2) We are to now solve the cannonball trajectory problem. Given a cannoball located at $(x_0, y_0) = (0, 0)$, with initial velocity $v_0 = 150$ m/s, and a target located at (1500 m, 0), at what angle α must the cannon shoot?

We solve this problem by writing down x(t) and y(t) for a particle moving under only the force of gravity.

$x(t) = x_0 + v_0 cos(\alpha)t = 150 cos(\alpha)t$

$y(t) = y_0 + v_0 sin(\alpha)t - \frac{g}{2}t^2 = 150 sin(\alpha)t - 5t^2$

When the cannonball lands at time t_f, we have that $y(t_f) = 0 = 150 sin(\alpha)t - 5{t_f}^2$. Solving gives $t_f = 30 sin(\alpha)$. We then plug this back into $x(t_f) = 1500 = 4500 cos(\alpha) sin(\alpha)$.

Thus, we just have to solve the equation $3 cos(\alpha) sin(\alpha) - 1 = 0$. To do this we use the bisection method shown below.

```
In [15]: #This is the function which we need to find the root of
         def f(x):
             return 3*math.cos(x)*math.sin(x) - 1

         #The bisection method, we iteratively use the method until the difference
         #between the last guess and the new one is less than 1e-10
         def BisectionMethod(start, end, error):
             diff = 10000
             x = start
             y = end
             old_midpoint = 3.141

             while (diff > error):
                 midpoint = (x+y)/2

                 diff = abs(midpoint - old_midpoint)

                 if (np.sign(f(midpoint)) == np.sign(f(x))):
                     x = (x+y)/2
                 if (np.sign(f(midpoint)) == np.sign(f(y))):
                     y = (x+y)/2
```

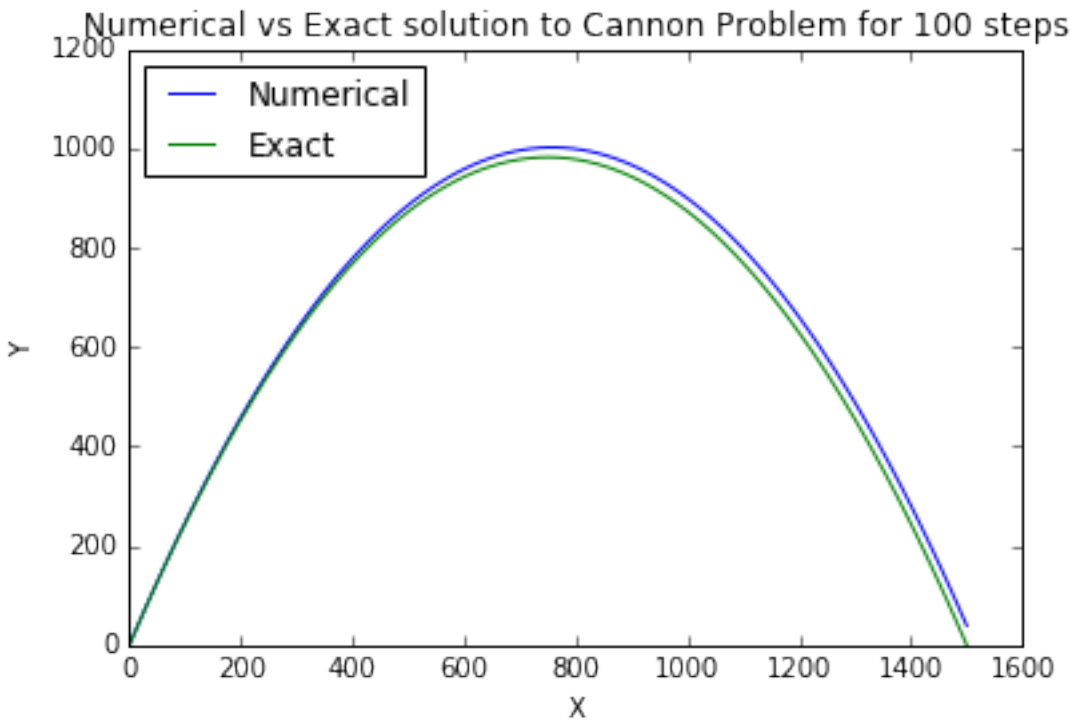
```
            old_midpoint = midpoint

        return old_midpoint

    alpha = BisectionMethod((math.pi)/2, (math.pi)/4, 1e-10)

    print('alpha = ' , alpha, 'radians, or', alpha*180/math.pi, ' degrees')
```

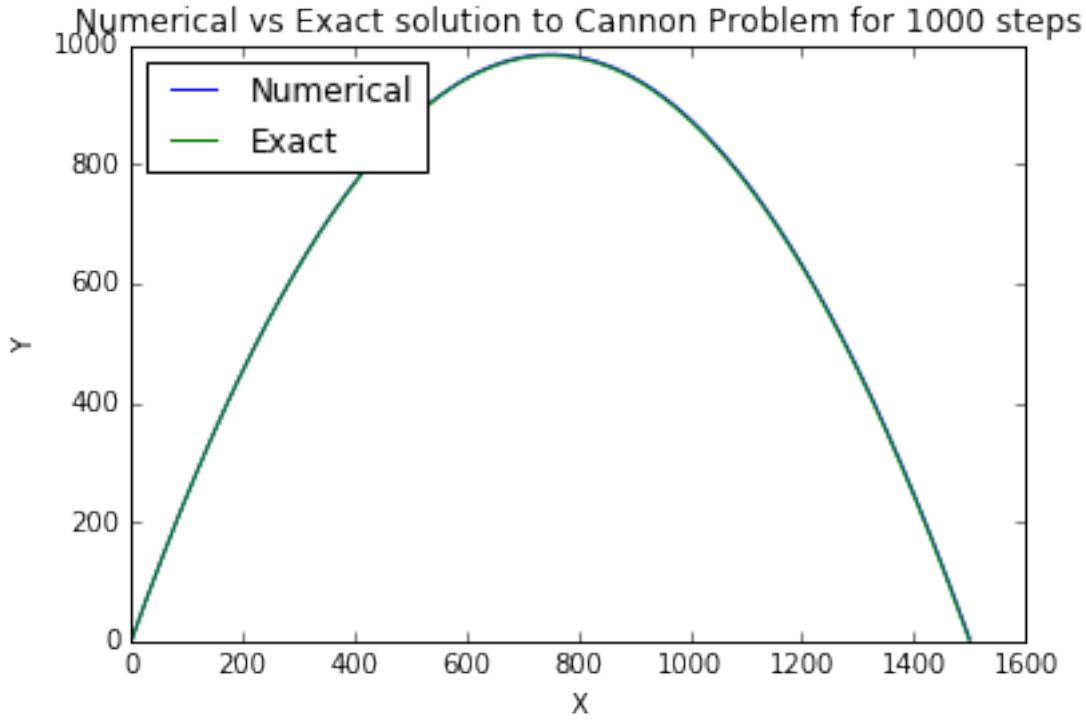
```
alpha =  1.2059324987176425 radians, or 69.09484255418649  degrees
```

Now we plot the exact solution vs a numerical solution using the forward euler method.

```
In [16]: def integrate_cannon(t_0, x_0, y_0, v_0, Alpha, steps):
             # Create empty arrays for x(t), v(t), and t
             x,y,vy = [], [], []

             t_f = math.sin(Alpha)*v_0/5

             # Define the interval and time step
             interval = (t_f - t_0)
             del_t = interval/steps

             #Append initial conditions
             x.append(x_0)
             y.append(y_0)
             vy.append(v_0*math.sin(Alpha))

             #Use forward Euler
             for n in range(1,steps + 1):
                 x.append(x[n-1] + v_0*math.cos(Alpha)*del_t)
                 vy.append(vy[n-1] + (-10)*del_t)
                 y.append(y[n-1] + vy[n-1]*del_t)

             return x,y

         def exact_cannon(t_0, x_0, y_0, v_0, Alpha, steps):
             # Create empty arrays for x(t), v(t), and t
             x,y,t = [],[],[]
             t_f = math.sin(Alpha)*v_0/5

             # Define the interval and time step
             interval = (t_f - t_0)
             del_t = interval/steps

             #Append initial conditions
             x.append(x_0)
             y.append(y_0)
```

```
    #Put exact result using same times steps as approximate method
    for n in range(1, steps + 1):
        t.append(t_0 + n*del_t)
        x.append(x[0] + v_0*math.cos(Alpha)*del_t*n)
        y.append(y[0] + v_0*math.sin(Alpha)*del_t*n - 5*((n*del_t)**2))

    return x,y

approx_result = integrate_cannon(0, 0, 0, 150, alpha, 100)
exact_result = exact_cannon(0, 0, 0, 150, alpha, 1000)
plt.plot(approx_result[0], approx_result[1], label="Numerical")
plt.plot(exact_result[0], exact_result[1], label="Exact")
plt.ylabel('Y')
plt.xlabel('X')
plt.legend(['Numerical', 'Exact'], loc="upper left")
plt.title("Numerical vs Exact solution to Cannon Problem for 100 steps")
plt.show()

approx_result_better = integrate_cannon(0, 0, 0, 150, alpha, 1000)
exact_result = exact_cannon(0, 0, 0, 150, alpha, 1000)
plt.plot(approx_result_better[0], approx_result_better[1], label="Numerical")
plt.plot(exact_result[0], exact_result[1], label="Exact")
plt.ylabel('Y')
plt.xlabel('X')
plt.legend(['Numerical', 'Exact'], loc="upper left")
plt.title("Numerical vs Exact solution to Cannon Problem for 1000 steps")
plt.show()
```


Numerical vs Exact solution to Cannon Problem for 100 steps
Numerical
Exact
Y
X
0
200
400
600
800
1000
1200
1400
1600


Numerical vs Exact solution to Cannon Problem for 1000 steps
Numerical
Exact
Y
X
0
200
400
600
800
1000
1200
1400
1600

We can see that the numerical solution tends to overshoot the target, but gets more and more accurate as the number of steps increases, i.e. the step size decreases.

In [ ]: