CS 111 (S19): Homework 7

Due by 6:00 PM, Wednesday, June 5

NAME and PERM ID No.: Chen Li, 5468137 (replace with yours)

UCSB EMAIL: chenli@ucsb.edu (replace with yours)

1. (Compare NCM problem 1.38.) The moral of this problem is that, with floating-point arithmetic, sometimes two algorithms look equivalent but one is better than the other at getting an accurate answer. Suppose you have a number $\hat{x}$ that is an approximation to another number x. Define the *relative error* in $\hat{x}$ as $|(\hat{x} - x)/x|$. (This only makes sense if $x \neq 0$.)

1a. The classic quadratic formula says that the two roots of the quadratic equation

$$ax^2 + bx + c = 0$$

are

$$x_0, x_1 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Use this formula in `numpy` (show your input and output) to compute both roots for

$$a = 1, \quad b = 10{,}000{,}000{,}000, \quad c = 1.$$

Also compute the roots two other ways: first with `numpy`'s `np.roots()`, and then by hand. To at least one significant digit, what is the relative error of the approximation computed using the quadratic formula to x_0? To x_1? What are the relative errors of the approximations computed using `np.roots()`?

For by hand, $x^2 + 10^{10}x + 1 = 0$ is equivalent to $(x + 10^{10} - 10^{-10})(x + 10^{-10}) = -10^{-20}$, so solution is -10^{10} and -10^{-10}

```
x0=(-10000000000-math.sqrt(100000000000000000000-4))/2
x1=(-10000000000+math.sqrt(100000000000000000000-4))/2
p = [1, 10000000000,1]
sol = np.array([-10**(10),-10**(-10)])
root = np.roots(p)
res1 = npla.norm((root-sol)/sol)
quadraroot = np.array([x0,x1])
res2 = npla.norm((quadraroot-sol)/sol)
print("by hand:    ", sol)
print("quadraroot:",quadraroot, "with relative error:", res2)
print("np.roots:   ",root, "with relative error:", res1)
```

```
by hand:    [-1.e+10 -1.e-10]
quadraroot: [-1.e+10  0.e+00] with relative error: 1.0
np.roots:   [-1.e+10 -1.e-10] with relative error: 0.0
```

1b. You should have found in (1a) that the classic formula is good for computing one root but not the other. Explain in a sentence why in this case one root isn't computed accurately. Hint: The answer involves IEEE floating-point arithmetic!

the reason is that the exponent part are large because it need to store 10^{10}, this make the exponent of the last digit of mantissa(gap size), which is the deciding factor of accuracy large. Therefore, when such two very large number with relatively very small difference make subtraction, the result is relative too small comparing with gap size and lead to lack of accuracy.

1c. Use the classic formula to compute one root accurately, and then use the fact that

$$x_0 x_1 = \frac{c}{a}$$

to compute the other. What are the relative errors now?

$x_0 = -1.e + 10$, thus, $x_1 =$

```
x1 = 1/x0
solC = np.array([x0, x1])
res3 = npla.norm((solC-sol)/sol)
print("solution from Veda:", solC, "with relative error:", res3)
```

```
solution from Veda: [-1.e+10 -1.e-10] with relative error: 0.0
```

2. The standard form of a first-order ODE initial value problem is

$$\dot{y} = f(t, y), \quad y(t0) = y0,$$

where t is a scalar and y is a vector. Write each of the following ODEs as an equivalent first-order system of ODEs in standard form:

2a. Van der Pol equation:

$$\frac{d^2x}{dt^2} = (1 - x^2)\frac{dx}{dt} - x.$$

$$y = x'$$

$$y' = x'' = (1 - x^2)y - x$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 1 - x^2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

2b. Blasius equation:

$$\frac{d^3x}{dt^3} = -x\frac{dx}{dt}.$$

$$x_1 = x'$$

$$x_2 = x_1'$$

$$x_3 = x_2' = -xx_1$$

$$\begin{pmatrix} x' \\ x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -x & 0 \end{pmatrix} \begin{pmatrix} x \\ x_1 \\ x_2 \end{pmatrix}$$

2c. Newton's second law of motion for a two-body problem in 2 dimensions (G and M are constants):

$$\frac{d^2x_0}{dt^2} = -GM\frac{x_0}{(x_0^2 + x_1^2)^{3/2}}, \tag{1}$$

$$\frac{d^2x_1}{dt^2} = -GM\frac{x_1}{(x_0^2 + x_1^2)^{3/2}}. \tag{2}$$

$$\begin{pmatrix} x_0' \\ x_0'' \\ x_1' \\ x_1'' \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ \frac{-GM}{(x_0^2+x_1^2)^{\frac{3}{2}}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{-GM}{(x_0^2+x_1^2)^{\frac{3}{2}}} & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_0' \\ x_1 \\ x_1' \end{pmatrix}$$

3. (Compare NCM problem 7.16.) This problem is partly about ODEs and partly about making nice plots with `matplotlib` (we import `matplotlib.pyplot` as `plt`).

Many modifications of the Lotka–Volterra predator-prey model that we saw in class have been proposed to more accurately reflect what happens in nature. For example, the number of rabbits can be prevented from growing indefinitely by fixing a maximum number R and changing the equations to

$$\frac{dr}{dt} = 2\Big(1 - \frac{r}{R}\Big)r - \alpha r f, \tag{3}$$

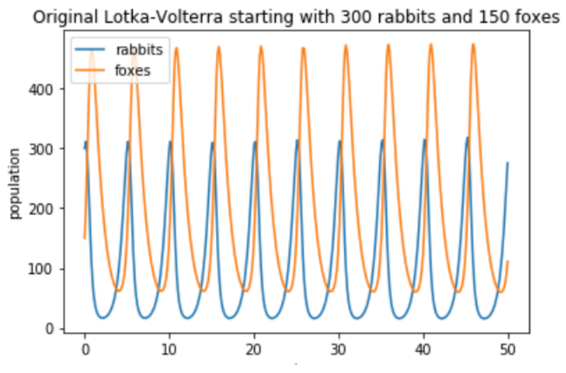
$$\frac{df}{dt} = -f + \alpha r f, \tag{4}$$

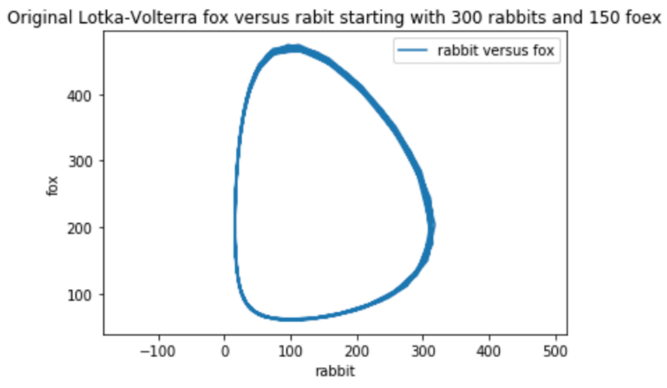
where t is time, $r(t)$ is the number of rabbits, $f(t)$ is the number of foxes, and $\alpha > 0$ is a constant. This makes dr/dt negative whenever $r > R$, which guarantees that the number of rabbits can never grow to exceed R.

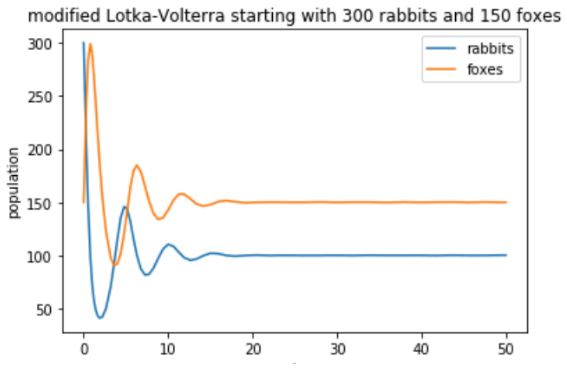
For $\alpha = 0.01$, compare the behavior of the original model with the behavior of this modified model with $R = 400$. Solve the equations (using `integrate.solve_ivp()` as we did in class) over 50 units of time, assuming that there are initially 300 rabbits and 150 foxes. Make four different plots to show the solutions and the phase space diagrams for both models as follows:

- number of foxes and rabbits (on the same plot) versus time for the original model,
- number of foxes and rabbits (on the same plot) versus time for the modified model,
- number of foxes versus number of rabbits (phase space) for the original model,
- number of foxes versus number of rabbits (phase space) for the modified model.

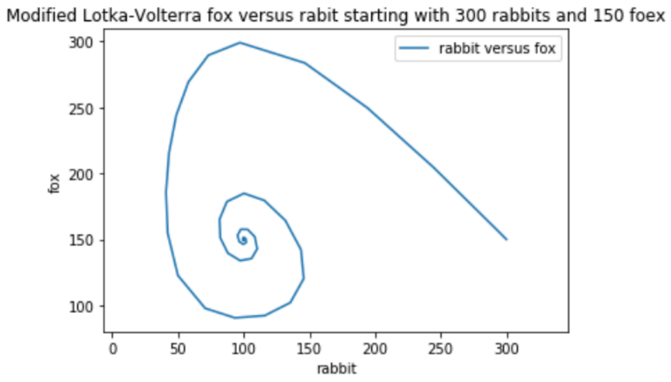
For all plots, label all curves (with `plt.legend()`) and all axes, and put a title on each plot that identifies it clearly. For the phase space plots, set the aspect ratio so that equal increments on the x- and y-axes are equal in size. (You may find the `matplotlib` tutorial linked under the "help" menu in Jupyter useful.)







modified Lotka-Volterra starting with 300 rabbits and 150 foxes
rabbits
foxes
population



Modified Lotka-Volterra fox versus rabit starting with 300 rabbits and 150 foex
rabbit versus fox
fox
rabbit

4. An important problem in classical mechanics is to determine the motion of two bodies under mutual gravitational attraction. Suppose that a body of mass m is orbiting a second body of much larger mass M, such as the earth orbiting the sun. From Newton's laws of motion and gravitation, the orbital trajectory $(x_0(t), x_1(t))$ is described by the system of second-order ODEs

$$\ddot{x}_0 = -GMx_0/r^3, \tag{5}$$
$$\ddot{x}_1 = -GMx_1/r^3, \tag{6}$$

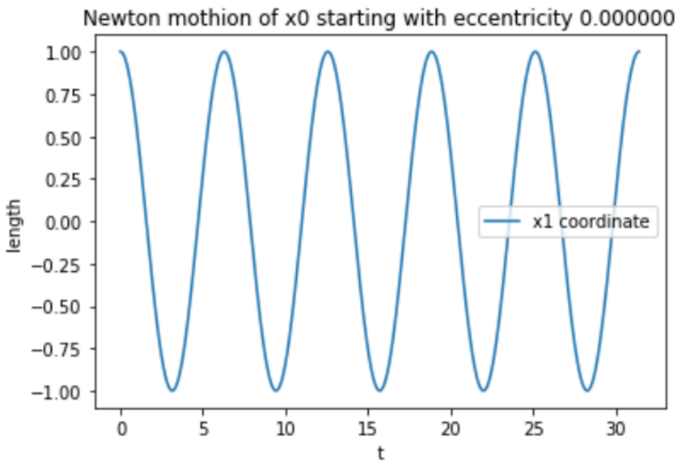
where G is the gravitational constant and $r = (x_0^2 + x_1^2)^{1/2} = ||x||$ is the distance of the orbiting body from the center of mass of the two bodies. For this exercise, we choose units such that $GM = 1$.

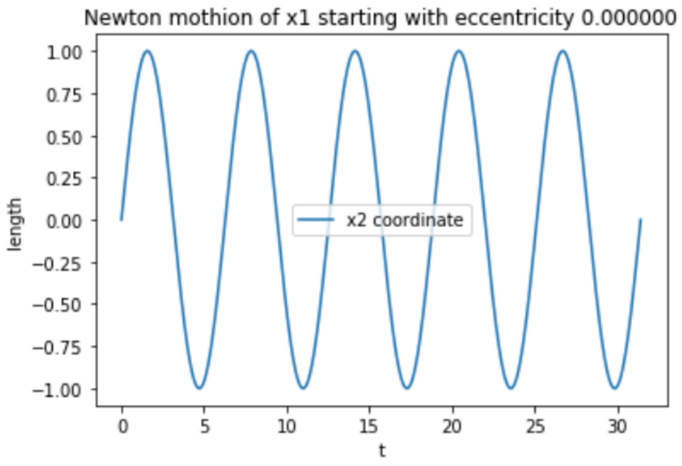
Use `integrate.solve_ivp()` to solve this system of ODEs with the initial conditions

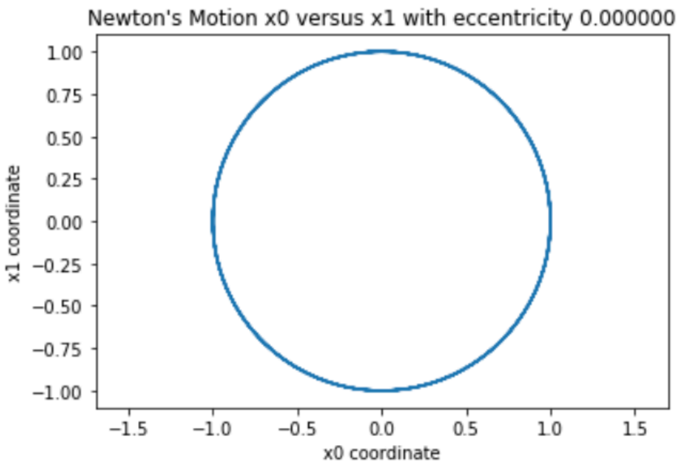
$$x_0(0) = 1 - e, \quad x_1(0) = 0, \tag{7}$$
$$\dot{x}_0(0) = 0, \quad \dot{x}_1(0) = \left(\frac{1+e}{1-e}\right)^{1/2}, \tag{8}$$

where e is the eccentricity of the resulting elliptical orbit, which has period 2π. Try the values $e = 0$ (which should give a circular orbit), $e = 0.5$, and $e = 0.9$. For each case, solve the ODE for at least one orbital period and obtain output at enough intermediate points to draw a smooth plot of the orbital trajectory. Make separate plots of x_0 versus t, x_1 versus t, and x_0 versus x_1, all with well-labeled axes and clear titles. For your plot of the orbit itself, x_0 versus x_1, use `plt.gca().axis('equal')` to make sure the scale is the same on both axes, so that a circle will look like a circle.

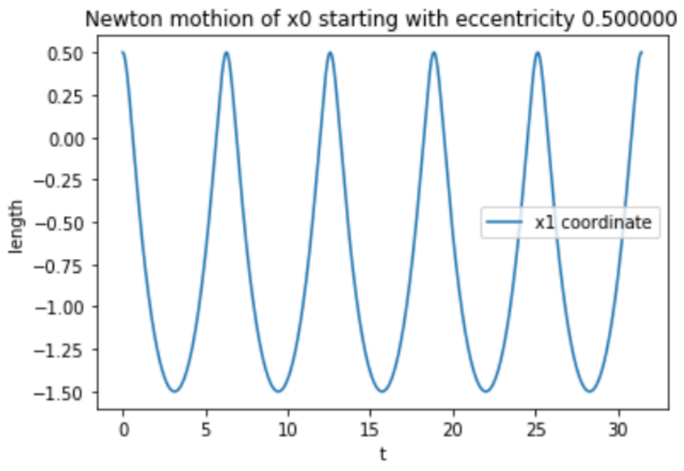
All the plot are drawn with $5 * 2\pi$ time.
When e = 0:

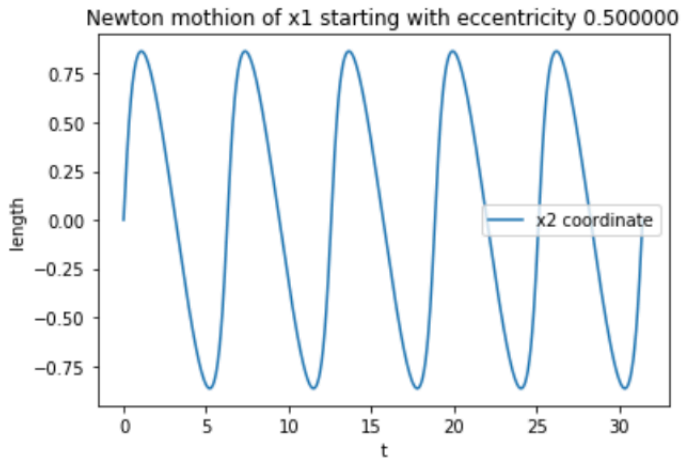


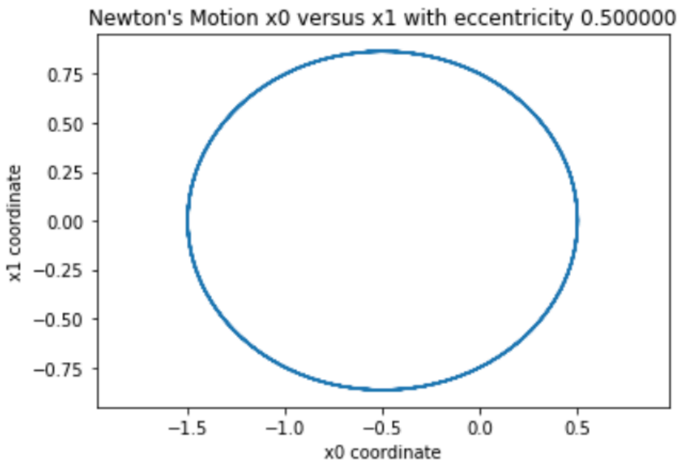




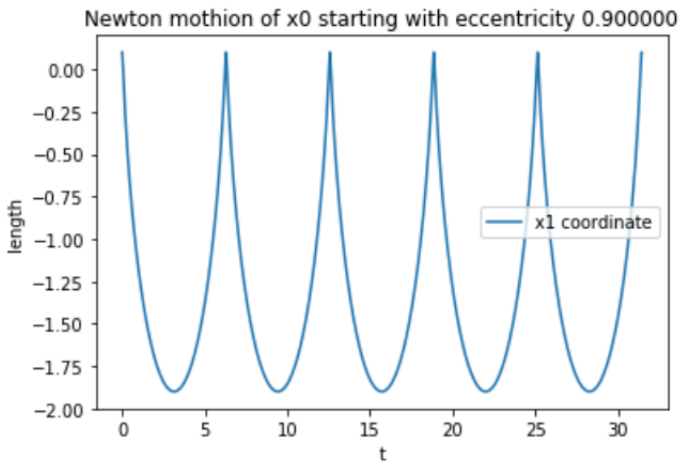
when e = 0.5:

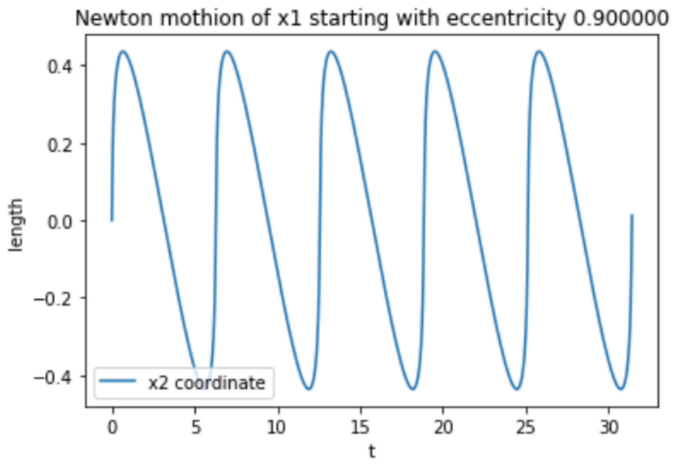

Newton mothion of x0 starting with eccentricity 0.500000
length
t
x1 coordinate

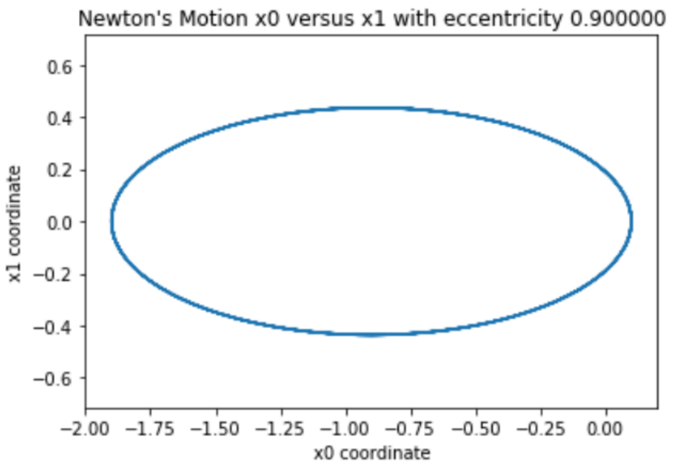

Newton mothion of x1 starting with eccentricity 0.500000
length
t
x2 coordinate


Newton's Motion x0 versus x1 with eccentricity 0.500000
x1 coordinate
x0 coordinate

when e = 0.9:


Newton mothion of x0 starting with eccentricity 0.900000
length
t
x1 coordinate


Newton mothion of x1 starting with eccentricity 0.900000
length
t
x2 coordinate


Newton's Motion x0 versus x1 with eccentricity 0.900000
x1 coordinate
x0 coordinate

Experiment with different error tolerances (use `help(integrate.solve_ivp)` to find out how to set error tolerances) to see how they affect (i) the amount of time required for the solution and (ii) how close the orbit comes to being closed. If you trace the orbit through several periods, does the orbit tend to wander or remain steady?

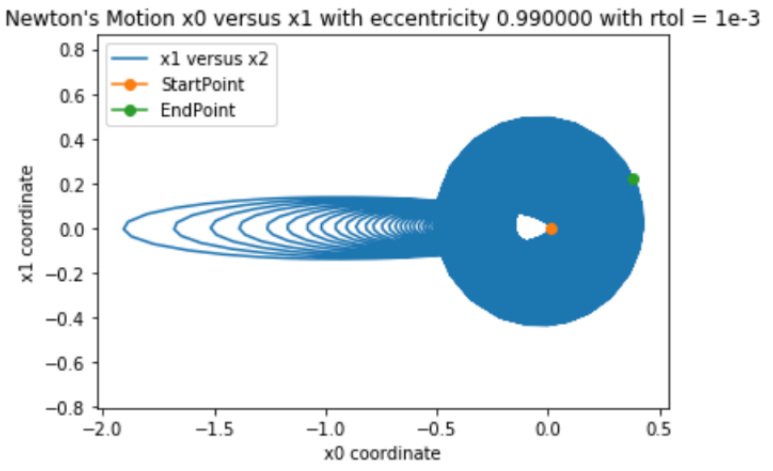
In addition to your plots, turn in an explanation in English of what experiments you did, what you observed, and what your conclusions were.

Here is a code for experiment, the trajectory of 100 period will be plot in blue line, the starting point will be plot as an orange point and the end point at 100 period will be plot as a green point. Based on math, these two points is supposed to be overlapped. Therefore, we can visualize the deviation of the stimulation by the shape of trajectory and the distance of starting and end point. For accuracy, eccentricity were set to 0.99

```
e = 0.99
tspan = (0,100*2*np.pi)
yinit = [1-e, 0, 0, math.sqrt((1+e)/(1-e))]
tol = -9
%time sol = integrate.solve_ivp(fun = Newton, t_span = tspan, y0 = yinit, method = 'RK23', rtol = 10**(tol))
FinalX = np.array([sol.y[0][-1],sol.y[2][-1]])
%matplotlib inline
plt.plot(sol.y[0], sol.y[2], label = 'x1 versus x2')
plt.plot(sol.y[0][0],sol.y[2][0], label = "StartPoint", marker = 'o')
plt.plot(sol.y[0][-1],sol.y[2][-1], label = "EndPoint", marker = 'o')
plt.legend()
plt.gca().axis('equal')
plt.xlabel('x0 coordinate')
plt.ylabel('x1 coordinate')
plt.title('Newton\'s Motion x0 versus x1 with eccentricity %f with rtol = 1e%i' % (e, tol))
```

tol = -3, extremely inaccurate

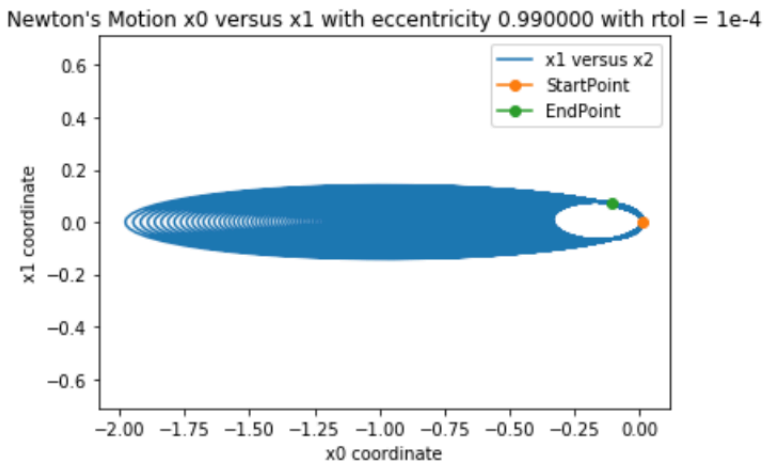
```
CPU times: user 2.27 s, sys: 10.7 ms, total: 2.28 s
Wall time: 2.27 s

Text(0.5, 1.0, "Newton's Motion x0 versus x1 with eccentricity 0.990000 with rtol = 1e-3")
```



.

tol = -4, it take 7.18s to calculate

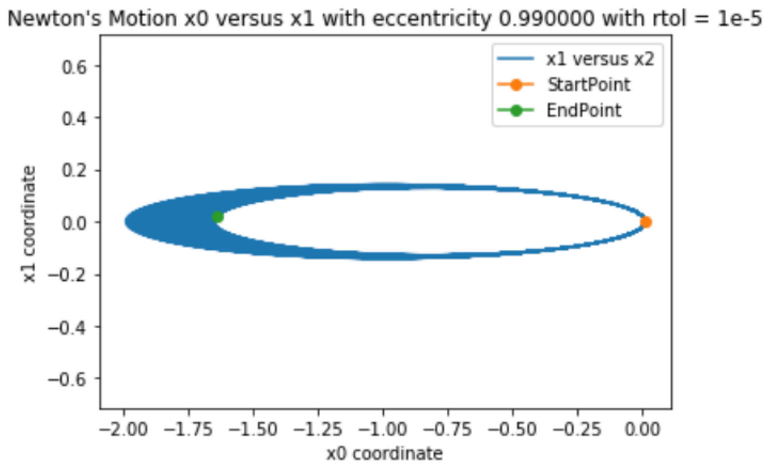
```
CPU times: user 7.03 s, sys: 147 ms, total: 7.18 s
Wall time: 7.18 s

Text(0.5, 1.0, "Newton's Motion x0 versus x1 with eccentricity 0.990000 with rtol = 1e-4")
```



---

tol = -5, extremely inaccurate

```
CPU times: user 2.77 s, sys: 12.1 ms, total: 2.78 s
Wall time: 2.78 s

Text(0.5, 1.0, "Newton's Motion x0 versus x1 with eccentricity 0.990000 with rtol = 1e-5")
```

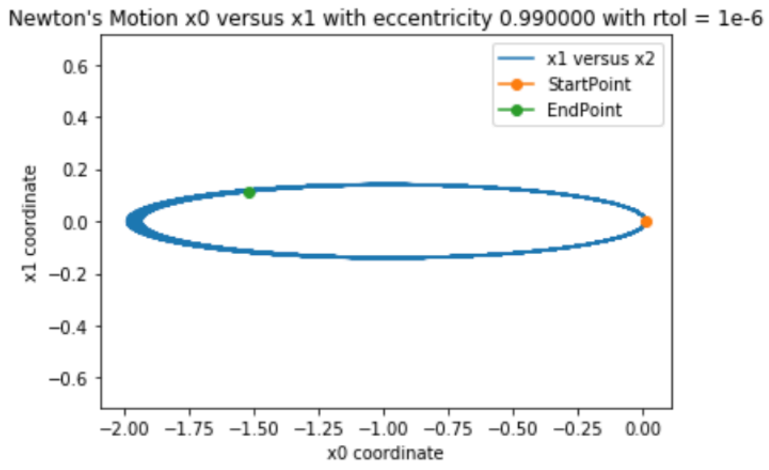


.

tol = -6, more accurate

```
CPU times: user 4.33 s, sys: 68.8 ms, total: 4.4 s
Wall time: 4.4 s
```

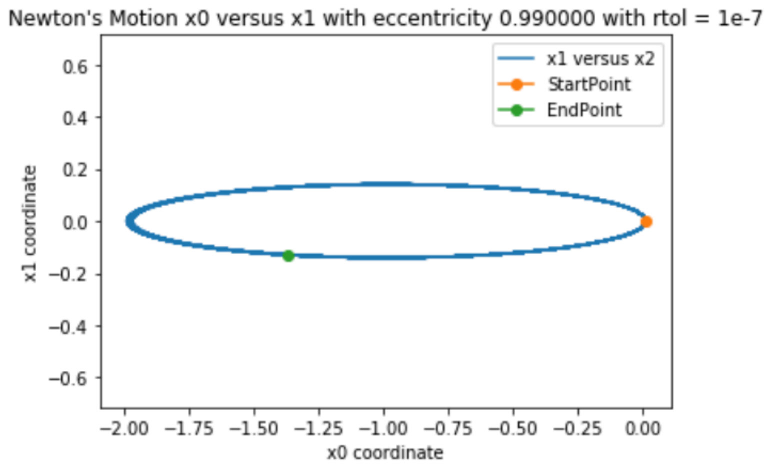
```
Text(0.5, 1.0, "Newton's Motion x0 versus x1 with eccentricity 0.990000 with rtol = 1e-6")
```



tol = -7, same, time gradually rise

```
CPU times: user 5.8 s, sys: 65.1 ms, total: 5.87 s
Wall time: 5.87 s
```

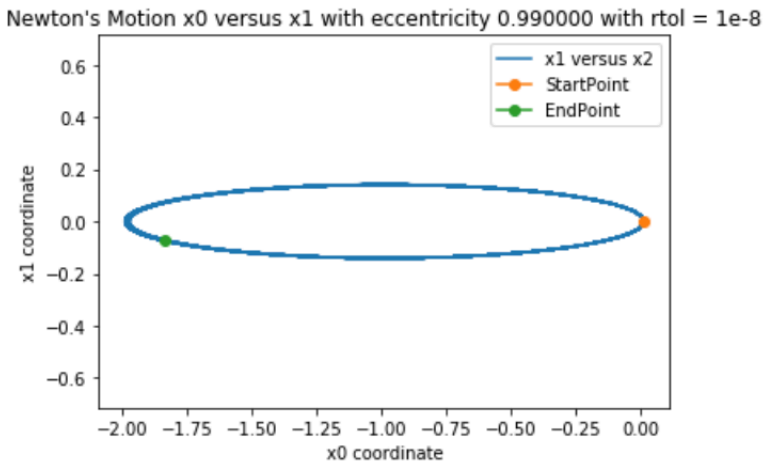
```
Text(0.5, 1.0, "Newton's Motion x0 versus x1 with eccentricity 0.990000 with rtol = 1e-7")
```



.

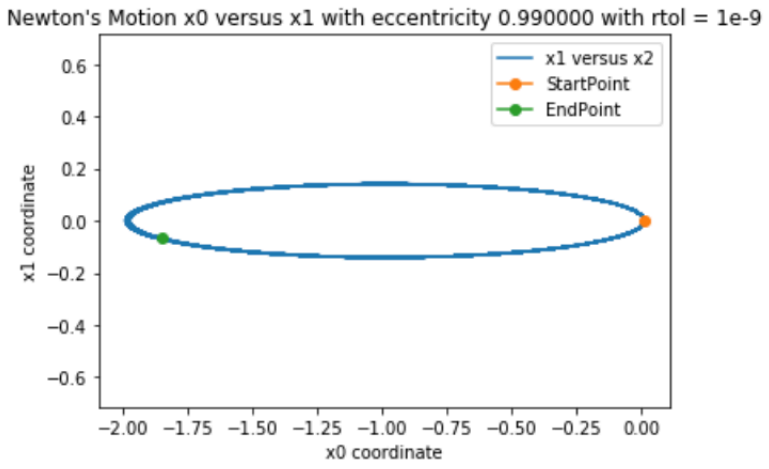
tol = -8, accuracy grow, running time rise drastically

```
CPU times: user 7.95 s, sys: 273 ms, total: 8.22 s
Wall time: 8.22 s

Text(0.5, 1.0, "Newton's Motion x0 versus x1 with eccentricity 0.990000 with rtol = 1e-8")
```



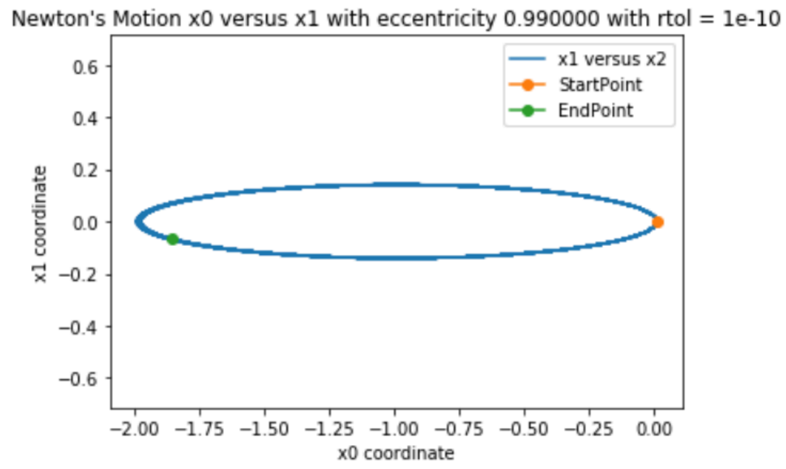
tol = -9, running time drop slightly

```
CPU times: user 7.6 s, sys: 274 ms, total: 7.87 s
Wall time: 7.9 s

Text(0.5, 1.0, "Newton's Motion x0 versus x1 with eccentricity 0.990000 with rtol = 1e-9")
```



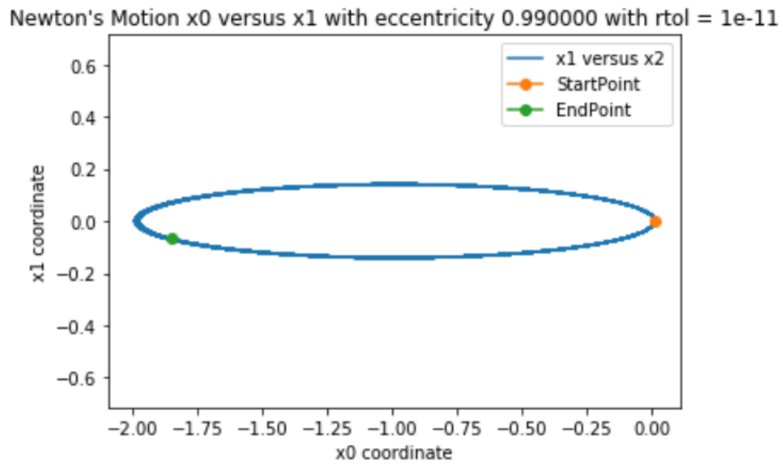
tol = -10, running drop slightly

```
CPU times: user 6.08 s, sys: 112 ms, total: 6.19 s
Wall time: 6.19 s

Text(0.5, 1.0, "Newton's Motion x0 versus x1 with eccentricity 0.990000 with rtol = 1e-10")
```



tol = -11, extremely inaccurate

```
CPU times: user 6.44 s, sys: 166 ms, total: 6.6 s
Wall time: 6.62 s

Text(0.5, 1.0, "Newton's Motion x0 versus x1 with eccentricity 0.990000 with rtol = 1e-11")
```



Conclusion: (tol = n meaning rtol = 10^{-n}) in general, running time grows as accuracy gets precise. However, it is not the all situation, in some cases, like tol = 4, tol = 8, their running time get very long. Actually, tol = 4 takes more times than any others tol except 8. in some local area, running time even decrease as tol increase. Accuracy is clearly getting better as rtol get smaller. Basing on the observation on the alignment of trajectory. However, the deviation of ending point is inevitable, which maybe related to step size.