KyotoUx-009x (/github/ryo0921/KyotoUx-009x/tree/master)
/ homework (/github/ryo0921/KyotoUx-009x/tree/master/homework)

Problems for #11

Q1

Check Python version using the command introduced in the video and choose the major version number (the first digit number of the output) from below.

- 1
- 2
- 3
- 4
- 5

A1

- 3

In [1]:
```
import sys
sys.version
```

Out[1]:
```
'3.5.2 |Anaconda 4.3.1 (x86_64)| (default, Jul  2 2016, 17:52:12) \n[GC
```

Q2

Which of the following commands (C1, C2, C3) correctly typesets the equation shown below?

$$\log N! \simeq N \log N - N$$

C1

```
$$\log N!\simeq N\log N - N $$
```

C2

```
\[\log N!\simeq N\log N - N \]
```

C3

```
\begin{equation}
\log N!\simeq N\log N - N
\end{equation}
```

- C1 only
- C2 only
- C3 only
- C1 and C2 only
- C1 and C3 only
- C2 and C3 only
- C1, C2, and C3

A2

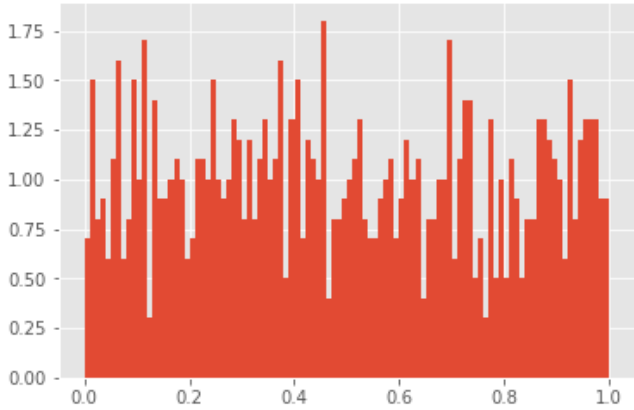
- C1 and C3 only

Problems for #12

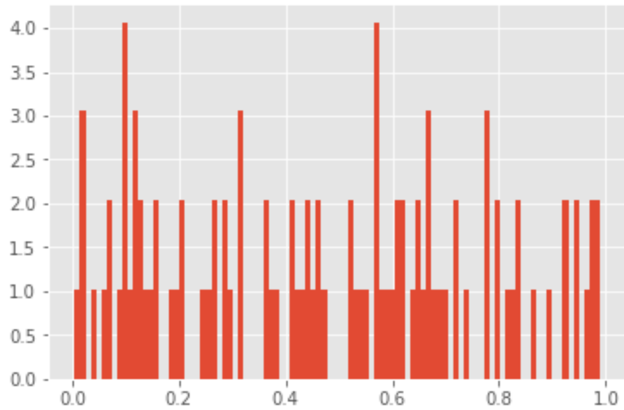
Q3

Make a histogram plot using the code example with only changing the size of uniform random numbers from 10^5 to 10^3. Which of the following graphs (G1, G2, G3, G4) is the closest to what you obtained?

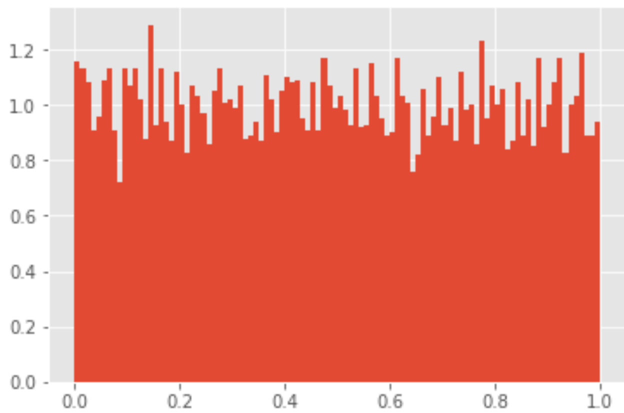
- G1



- G2



- G3



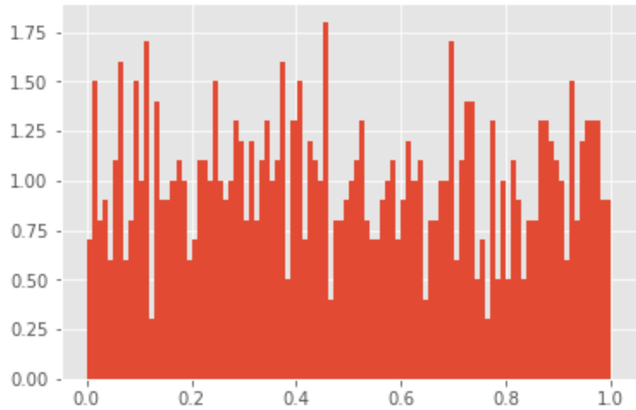
- G4

A3

- G1

In [2]:
```
% matplotlib inline
import numpy as np # import numpy library as np
import matplotlib.pyplot as plt # import pyplot library as plt
plt.style.use('ggplot') # use "ggplot" style for graphs
```

In [3]:
```
N = 1000 # size of R
np.random.seed(0) # initialization of the random number generator
R = np.random.rand(N) # generate random sequence and store it as R
# plot normalized histgram of R using 100 bins
plt.hist(R,bins=100,normed=True)
plt.show() # show plot
```



Q4

The following code was made to plot $\sin(\theta)\cos(\theta)$ versus θ for $-\pi \leq \theta \leq \pi$, but it possesses potential bug(s). Debug the code and answer the number of line(s) on which you find the bug(s).

- 1st line only
- 2nd line only
- 4th line only
- 5th line only
- 1st and 2nd lines only
- 1st and 5th lines only
- 1st, 4th, and 5th lines only

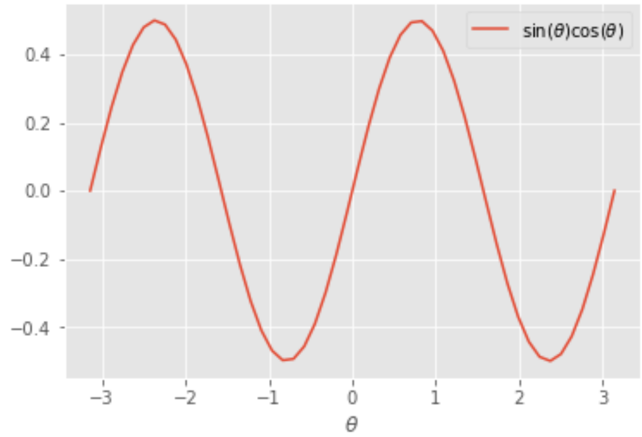
In [ ]:
```
# code containing potential bug(s)
x = np.linspace(-pi,pi)
y = np.sin(x)*cos(x)
plt.plot(x,y)
plt.xlabel(r"$\theta$")
plt.legend([r'$\sin(\theta)\cos(\theta)$'])
plt.show()
```

A4

- 1st and 2nd lines only

In [4]:

```
x = np.linspace(-np.pi,np.pi) # np. was missing
y = np.sin(x)*np.cos(x) # np. was missing
plt.plot(x,y)
plt.xlabel(r"$\theta$")
plt.legend([r'$\sin(\theta)\cos(\theta)$'])
plt.show()
```



Problems for #13

Q5

Compare the Euler method with the Taylor expansion, and answer the order of the Euler method in terms of Δt.

- 0th order
- 1st order
- 2nd order
- 3rd order
- 4th order

A5

- 1st order

Euler method

$$y_{i+1} = y_i + \Delta t f_i = y_i + \Delta t \frac{dy}{dt}\bigg|_i \tag{AH1}$$

Taylor expansion $(t + \Delta t)$

$$y_{i+1} = y_i + \Delta t \frac{dy}{dt}\bigg|_i + \frac{\Delta t^2}{2}\frac{d^2y}{dt^2}\bigg|_i + \frac{\Delta t^3}{3}\frac{d^3y}{dt^3}\bigg|_i + \cdots \tag{AH2}$$

$\therefore$ 1st order

Q6

Compare the Leap-Frog method with the Taylor expansion, and answer the order of the Leap-Frog method in terms of Δt.

- 0th order
- 1st order
- 2nd order
- 3rd order
- 4th order

A6

- 2nd order

Leapfrog method

$$y_{i+1} = y_{i-1} + 2\Delta t f_i = y_i + 2\Delta t \left.\frac{dy}{dt}\right|_i \tag{AH3}$$

Taylor expansion ($-\Delta t$)

$$y_{i-1} = y_i - \Delta t \left.\frac{dy}{dt}\right|_i + \frac{\Delta t^2}{2}\left.\frac{d^2 y}{dt^2}\right|_i - \frac{\Delta t^3}{3}\left.\frac{d^3 y}{dt^3}\right|_i + \cdots \tag{AH4}$$

Eq.(AH2) $-$ Eq.(AH4)

$$y_{i+1} = y_{i-1} + 2\Delta t \left.\frac{dy}{dt}\right|_i - 0 + \frac{2\Delta t^3}{3}\left.\frac{d^3 y}{dt^3}\right|_i + \cdots \tag{AH5}$$

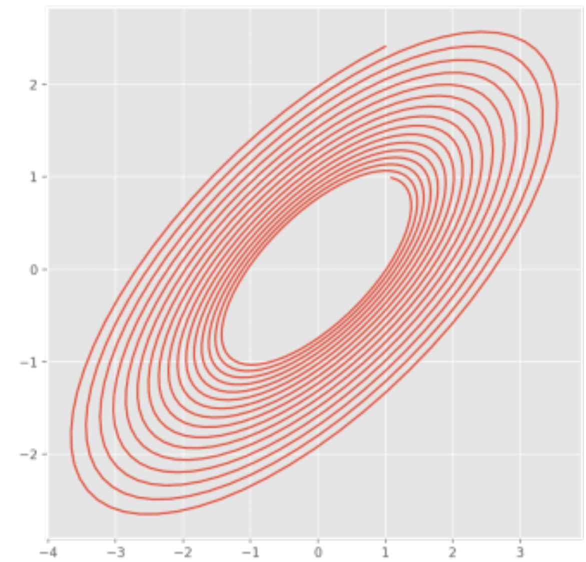
$\therefore$ 2nd order

Problems for #14

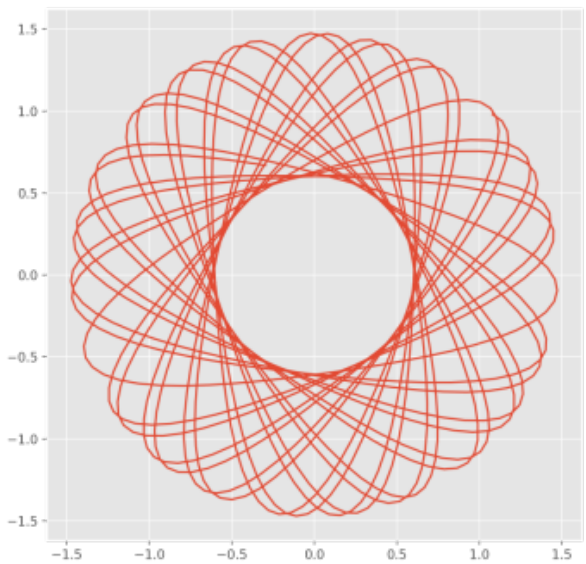
Q7

Perform the simulation using the code example for the Euler method with only changing $\zeta = 0.1$, and plot $R_x(t)$ vs. $R_y(t)$. Which of the following graphs (G11, G12, G13, G14) is the closest to what you obtained?

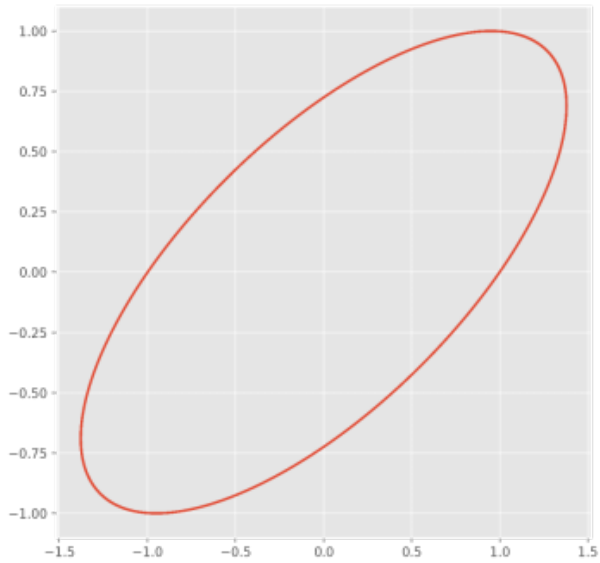
- G11



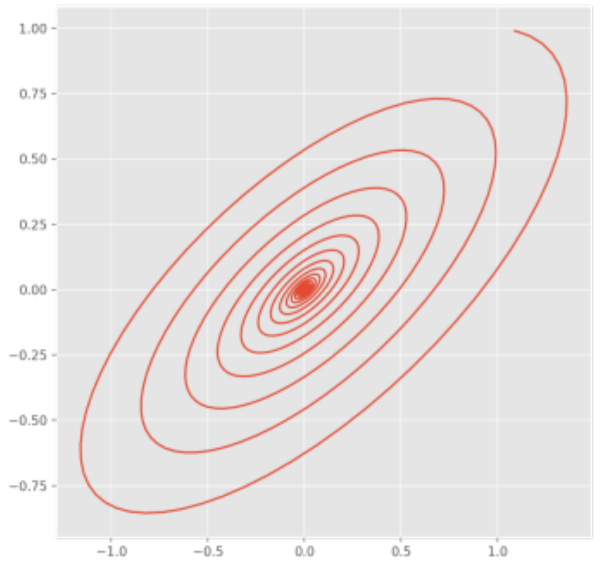
- G12



- G13



- G14



A7

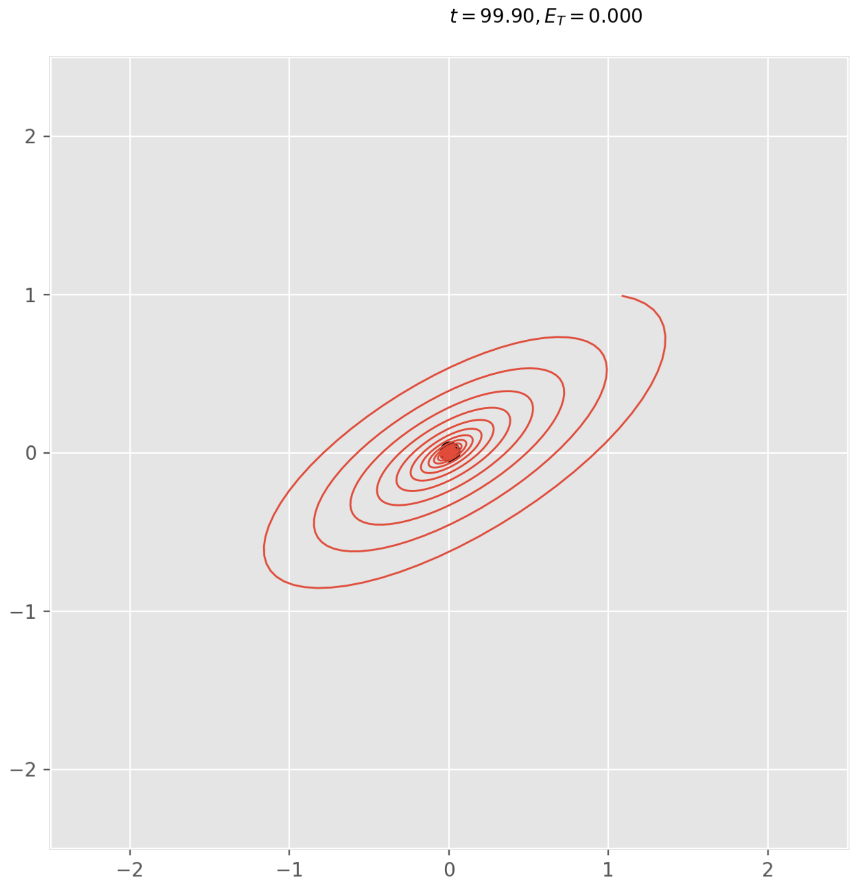
- G14

In [1]:
```
% matplotlib nbagg
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
plt.style.use('ggplot')
```

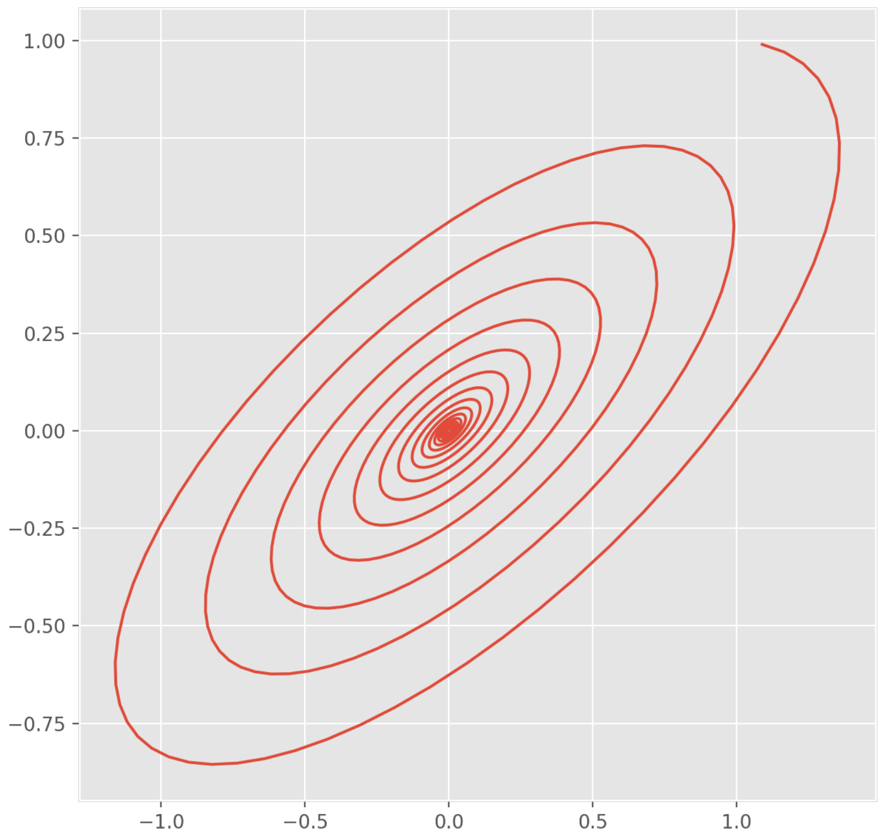
In [2]:
```
dim  = 2    # system dimension (x,y)
nums = 1000 # number of steps
R    = np.zeros(dim) # particle position
V    = np.zeros(dim) # particle velocity
Rs   = np.zeros([dim,nums]) # particle position (at all steps)
Vs   = np.zeros([dim,nums]) # particle velocity (at all steps)
Et   = np.zeros(nums) # total enegy of the system (at all steps)
time = np.zeros(nums) # time (at all steps)
```

In [3]:
```
def init(): # initialize animation
    particles.set_data([], [])
    line.set_data([], [])
    title.set_text(r'')
    return particles,line,title
def animate(i): # define amination
    global R,V,F,Rs,Vs,time,Et
    V = V*(1-zeta/m*dt)-k/m*dt*R # Euler method Eq.(B4)
    R = R + V*dt                 # Euler method Eq.(B3)
    Rs[0:dim,i]=R
    Vs[0:dim,i]=V
    time[i]=i*dt
    Et[i]=0.5*m*np.linalg.norm(V)**2+0.5*k*np.linalg.norm(R)**2
    particles.set_data(R[0], R[1])      # current position
    line.set_data(Rs[0,0:i], Rs[1,0:i]) # add latest position Rs
    title.set_text(r"$t = {0:.2f},E_T = {1:.3f}$".format(i*dt,Et[i]))
    return particles,line,title
```

In [4]:
```
# System parameters
# particle mass, spring & friction constants
m, k, zeta = 1.0, 1.0, 0.1
# Initial condition
R[0], R[1] = 1., 1. # Rx(0), Ry(0)
V[0], V[1] = 1., 0. # Vx(0), Vy(0)
dt   = 0.1*np.sqrt(k/m) # set \Delta t
box  = 5 # set size of draw area
# set up the figure, axis, and plot element for animatation
fig, ax = plt.subplots(figsize=(7.5,7.5)) # setup plot
ax = plt.axes(xlim=(-box/2,box/2),ylim=(-box/2,box/2)) # draw range
particles, = ax.plot([],[],'ko', ms=10) # setup plot for particle
line,=ax.plot([],[],lw=1) # setup plot for trajectry
title=ax.text(0.5,1.05,r'',transform=ax.transAxes,va='center') # title
anim=animation.FuncAnimation(fig,animate,init_func=init,
     frames=nums,interval=5,blit=True,repeat=False) # draw animation
# anim.save('movie.mp4',fps=20,dpi=400)
```



In [5]:
```
fig, ax = plt.subplots(figsize=(7.5,7.5))
ax.plot(Rs[0,0:nums],Rs[1,0:nums]) # parameteric plot Rx(t) vs. Ry(t)
plt.show()
```



Q8

Modify the original code for the harmonic spring $\mathbf{F}_{\rm spring} = -k\mathbf{R}$ to an unharmonic spring $\mathbf{F}_{\rm spring} = -kR^2\mathbf{R}$ with setting back to $\zeta = 0.0$, and plot $R_x(t)$ vs. $R_y(t)$. Which of the previous graphs (G11, G12, G13, G14) is the closest to what you obtained? Be careful that the changes will take place on the two lines indicated below.

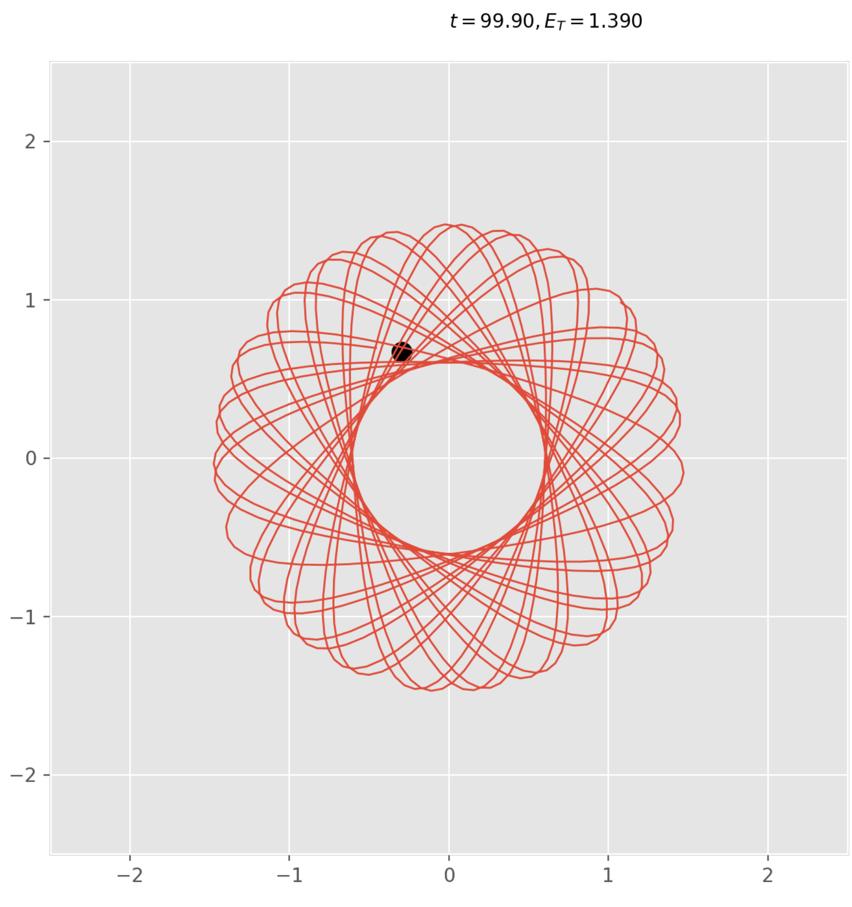
In [ ]:
```
V = V*(1-zeta/m*dt)-k/m*dt*R
Et[i]=0.5*m*np.linalg.norm(V)**2+0.5*k*np.linalg.norm(R)**2
```

A8

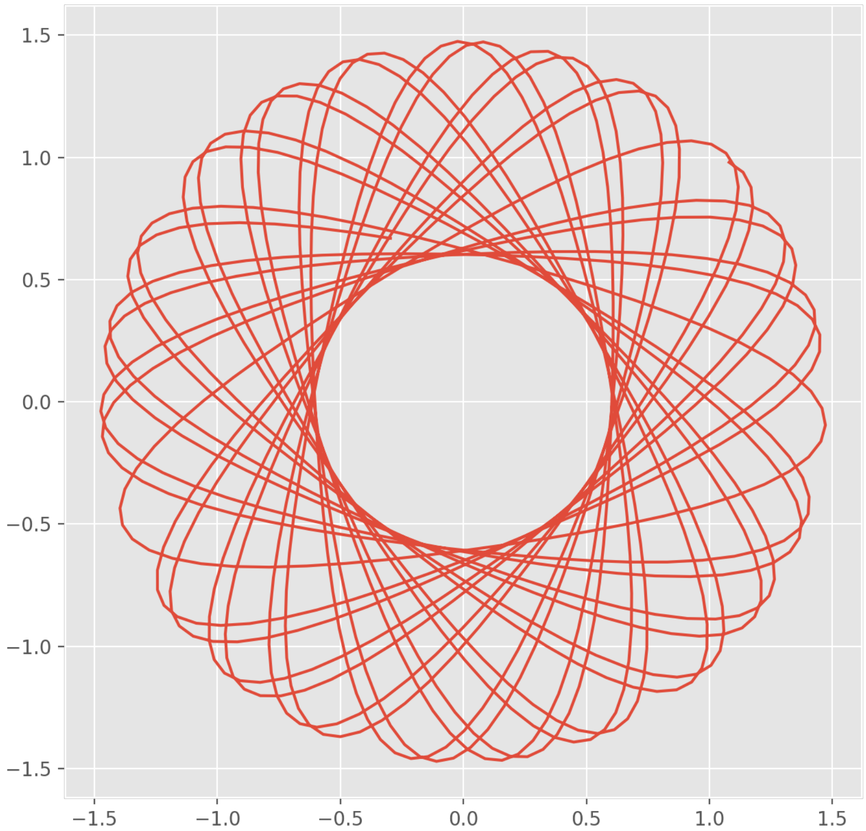
- G12

In [6]:
```
def animate(i): # define amination
    global R,V,F,Rs,Vs,time,Et
    # Euler method Eqs.(B3) and (B4)
    V = V*(1-zeta/m*dt)-k/m*dt*np.linalg.norm(R)**2*R
    R = R + V*dt
    Rs[0:dim,i]=R
    Vs[0:dim,i]=V
    time[i]=i*dt
    Et[i]=0.5*m*np.linalg.norm(V)**2+0.25*k*np.linalg.norm(R)**4
    particles.set_data(R[0], R[1]) # current position
    line.set_data(Rs[0,0:i], Rs[1,0:i]) # add latest position Rs
    title.set_text(r"$t = {0:.2f},E_T = {1:.3f}$".format(i*dt,Et[i]))
    return particles,line,title
```

In [7]:
```
# System parameters
# particle mass, spring & friction constants
m, k, zeta = 1.0, 1.0, 0.0
# Initial condition
R[0], R[1] = 1., 1. # Rx(0), Ry(0)
V[0], V[1] = 1., 0. # Vx(0), Vy(0)
dt   = 0.1*np.sqrt(k/m) # set \Delta t
box  = 5 # set size of draw area
# set up the figure, axis, and plot element for animatation
fig, ax = plt.subplots(figsize=(7.5,7.5)) # setup plot
ax = plt.axes(xlim=(-box/2,box/2),ylim=(-box/2,box/2)) # draw range
particles, = ax.plot([],[],'ko', ms=10) # setup plot for particle
line,=ax.plot([],[],lw=1) # setup plot for trajectry
title=ax.text(0.5,1.05,r'',transform=ax.transAxes,va='center') # title
anim=animation.FuncAnimation(fig,animate,init_func=init,
     frames=nums,interval=5,blit=True,repeat=False) # draw animation
# anim.save('movie.mp4',fps=20,dpi=400)
```



In [8]:
```
fig, ax = plt.subplots(figsize=(7.5,7.5))
ax.plot(Rs[0,0:nums],Rs[1,0:nums]) # parameteric plot Rx(t) vs. Ry(t)
plt.show()
```



Homework for #13

Q13-1

We modify the original code using the Euler method to solve Eq.(A22) for implementing the 2nd order Runge-Kutta method. It can be done by replacing the following code fragment with another one.

In [ ]:
```
for i in range(step-1):
    y[i+1]=y[i]-dt*y[i]
```

Choose the most appropriate one from the code fragment shown below (C11, C12, C13, C14).

In [ ]:
```
# C11
y1 = np.zeros(step)
for i in range(step-1):
    y1[i]=y[i]-dt*y[i]
    y[i+1]=y[i]-0.5*dt*y1[i]
```

In [ ]:
```
# C12
y1 = np.zeros(step)
for i in range(step-1):
    y1[i]=y[i]-0.5*dt*y[i]
    y[i+1]=y[i]-dt*y1[i]
```

In [ ]:
```
# C13
y1 = np.zeros(step)
y2 = np.zeros(step)
y3 = np.zeros(step)
for i in range(step-1):
    y1[i]=y[i]-0.5*dt*y[i]
    y2[i]=y[i]-0.5*dt*y1[i]
    y3[i]=y[i]-dt*y2[i]
    y[i+1]=y[i]-dt*(y[i]+y1[i]+y2[i]+y3[i])/4.0
```

In [ ]:
```
# C14
y1 = np.zeros(step)
y2 = np.zeros(step)
y3 = np.zeros(step)
for i in range(step-1):
    y1[i]=y[i]-0.5*dt*y[i]
    y2[i]=y[i]-0.5*dt*y1[i]
    y3[i]=y[i]-dt*y2[i]
    y[i+1]=y[i]-dt*(y[i]+2.0*y1[i]+2.0*y2[i]+y3[i])/6.0
```

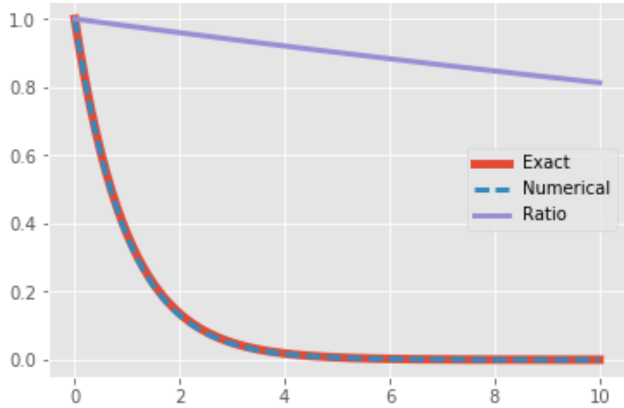
A13-1

- C12

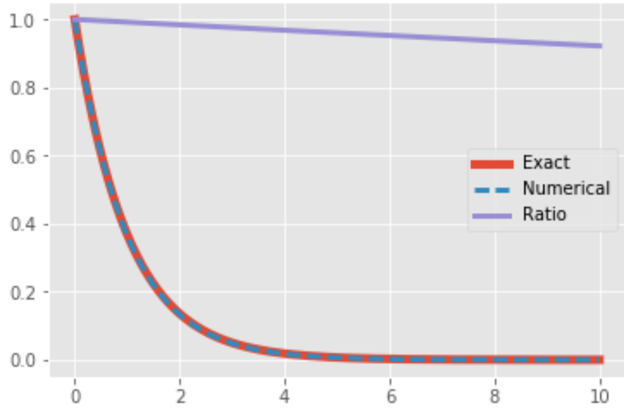
Q13-2

Perform the simulation using the modified code with the 2nd order Runge-Kutta method, and make the same graph to the one which we made with the Euler method in the video lessen. Which of the following graphs (G21, G22, G23, G24) is the closest to what you obtained?

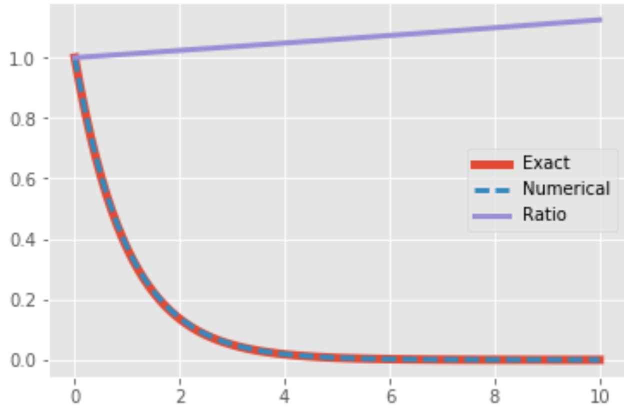
- G21



- G22



- G23



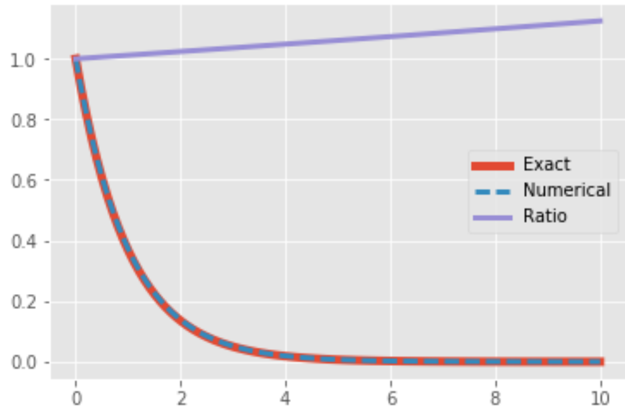
- G24

A13-2

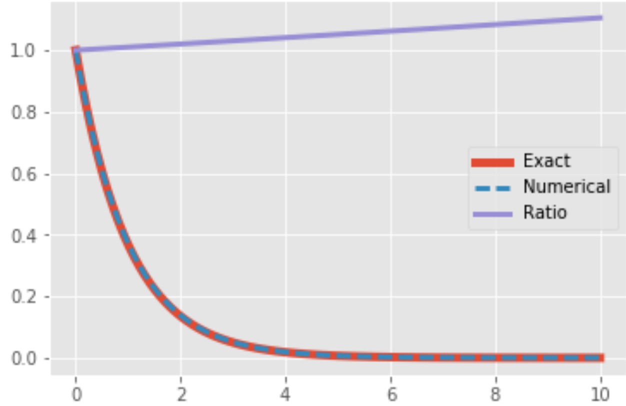
- G23

In [9]:
```
% matplotlib inline
import numpy as np                 # import numpy library as np
import matplotlib.pyplot as plt # import pyplot library as plt
plt.style.use('ggplot')          # use "ggplot" style for graphs
```

In [10]:
```
# Runge-Kutta 2nd
dt, tmin, tmax = 0.1, 0.0, 10.0 # set \Delta t, t0, tmax
step=int((tmax-tmin)/dt)
# create array t from tmin to tmax with equal interval dt
t = np.linspace(tmin,tmax,step)
y = np.zeros(step) # initialize array y as all 0
ya = np.exp(-t) # analytical solution y=exp(-t)
plt.plot(t,ya,lw=5,label='Exact') # plot y vs. t (analytical)
y[0]=1.0 # initial condition
#####
y1 = np.zeros(step) # initialize array y1 as all 0
for i in range(step-1):
    y1[i]=y[i]-0.5*dt*y[i] # Runge-Kutta Eq.(A15)
    y[i+1]=y[i]-dt*y1[i] # Runge-Kutta Eq.(A16)
#####
plt.plot(t,y, lw=3,ls='--',label='Numerical')# plot y vs t (numerical)
plt.plot(t,y/ya,lw=3,label='Ratio') # plot y/ya vs. t
plt.legend() # display legend
plt.show() # display plots
```



In [11]:
```
# Runge-Kutta 4th
dt,tmin,tmax =0.1,0.0,10.0 # set \Delta t, t0, tmax
step=int((tmax-tmin)/dt)
# create array t from tmin to tmax with equal interval dt
t = np.linspace(tmin,tmax,step)
y = np.zeros(step) # initialize array y as all 0
ya = np.exp(-t) # analytical solution y=exp(-t)
plt.plot(t,ya,lw=5,label='Exact') # plot y vs. t (analytical)
y[0]=1.0 # initial condition
#####
y1 = np.zeros(step) # initialize array y1 as all 0
y2 = np.zeros(step) # initialize array y2 as all 0
y3 = np.zeros(step) # initialize array y3 as all 0
for i in range(step-1):
    y1[i]=y[i]-0.5*dt*y[i] # Runge-Kutta Eq.(A18)
    y2[i]=y[i]-0.5*dt*y1[i] # Runge-Kutta Eq.(A19)
    y3[i]=y[i]-dt*y2[i] # Runge-Kutta Eq.(A20)
    y[i+1]=y[i]-dt*(y[i]+2.0*y1[i]+2.0*y2[i]+y3[i])/6.0 # Runge-Kutta Eq
#####
plt.plot(t,y,lw=3,ls='--',label='Numerical') # plot y vs t (numerical)
plt.plot(t,y/ya,lw=3, label="Ratio") # plot y/ya vs. t
plt.legend() # display legend
plt.show() # display plots
```



Homework for #14

Q14-1

We modify the original code using the Euler method to simulate the motion of harmonic oscillator for implementing the 4th order Runge-Kutta method. It can be done by modifying the following original `animate` function appropriately.

In [ ]:
```
def animate(i):
    global R,V,F,Rs,Vs,time,Et
    V = V*(1-zeta/m*dt)-k/m*dt*R
    R = R + V*dt
    Rs[0:dim,i]=R
    Vs[0:dim,i]=V
    time[i]=i*dt
    Et[i]=0.5*m*np.linalg.norm(V)**2+0.5*k*np.linalg.norm(R)**2
    particles.set_data(R[0], R[1])
    line.set_data(Rs[0,0:i], Rs[1,0:i])
    title.set_text(r"$t = {0:.2f},E_T = {1:.3f}$".format(i*dt,Et[i]))
    return particles,line,title
```

Choose the most appropriate `animate` function from the codes shown below (C21, C22, C23, C24).

In [ ]:

```
# C21
R1    = np.zeros(dim)
V1    = np.zeros(dim)
def animate(i):
    global R,V,F,Rs,Vs,time,Et
    V1 = V - zeta/m*dt*V - k/m*dt*R
    R1 = R + V*dt
    V = V - V1*zeta/m*0.5*dt - k/m*0.5*dt*R1
    R = R + V1*dt
    Rs[0:dim,i]=R
    Vs[0:dim,i]=V
    time[i]=i*dt
    Et[i]=0.5*m*np.linalg.norm(V)**2+0.5*k*np.linalg.norm(R)**2
    particles.set_data(R[0], R[1])
    line.set_data(Rs[0,0:i], Rs[1,0:i])
    title.set_text(r"$t = {0:.2f},E_T = {1:.3f}$".format(i*dt,Et[i]))
    return particles,line,title
```

In [ ]:

```
# C22
R1    = np.zeros(dim)
V1    = np.zeros(dim)
def animate(i):
    global R,V,F,Rs,Vs,time,Et
    V1 = V - zeta/m*0.5*dt*V - k/m*0.5*dt*R
    R1 = R + V*0.5*dt
    V = V - V1*zeta/m*dt - k/m*dt*R1
    R = R + V1*dt
    Rs[0:dim,i]=R
    Vs[0:dim,i]=V
    time[i]=i*dt
    Et[i]=0.5*m*np.linalg.norm(V)**2+0.5*k*np.linalg.norm(R)**2
    particles.set_data(R[0], R[1])
    line.set_data(Rs[0,0:i], Rs[1,0:i])
    title.set_text(r"$t = {0:.2f},E_T = {1:.3f}$".format(i*dt,Et[i]))
    return particles,line,title
```

In [ ]:
```
# C23
R1    = np.zeros(dim)
V1    = np.zeros(dim)
R2    = np.zeros(dim)
V2    = np.zeros(dim)
R3    = np.zeros(dim)
V3    = np.zeros(dim)
R4    = np.zeros(dim)
V4    = np.zeros(dim)
def animate(i):
    global R,V,F,Rs,Vs,time,Et
    V1 = V - zeta/m*0.5*dt*V - k/m*0.5*dt*R
    R1 = R + V*0.5*dt
    V2 = V - zeta/m*0.5*dt*V1 - k/m*0.5*dt*R1
    R2 = R + V1*0.5*dt
    V3 = V - zeta/m*dt*V2 - k/m*dt*R2
    R3 = R + V2*dt
    V4 = V - (V+V1+V2+V3)/4.*zeta/m*dt - k/m*dt*(R+R1+R2+R3)/4.
    R4 = R + (V+V1+V2+V3)/4.*dt
    R  = R4
    V  = V4
    Rs[0:dim,i]=R
    Vs[0:dim,i]=V
    time[i]=i*dt
    Et[i]=0.5*m*np.linalg.norm(V)**2+0.5*k*np.linalg.norm(R)**2
    particles.set_data(R[0], R[1]) # current position
    line.set_data(Rs[0,0:i], Rs[1,0:i]) # add latest position Rs
    title.set_text(r"$t = {0:.2f},E_T = {1:.3f}$".format(i*dt,Et[i]))
    return particles,line,title
```

In [ ]:
```
# C24
R1    = np.zeros(dim)
V1    = np.zeros(dim)
R2    = np.zeros(dim)
V2    = np.zeros(dim)
R3    = np.zeros(dim)
V3    = np.zeros(dim)
R4    = np.zeros(dim)
V4    = np.zeros(dim)
def animate(i):
    global R,V,F,Rs,Vs,time,Et
    V1 = V - zeta/m*0.5*dt*V - k/m*0.5*dt*R
    R1 = R + V*0.5*dt
    V2 = V - zeta/m*0.5*dt*V1 - k/m*0.5*dt*R1
    R2 = R + V1*0.5*dt
    V3 = V - zeta/m*dt*V2 - k/m*dt*R2
    R3 = R + V2*dt
    V4 = V - (V+V1*2+V2*2+V3)/6.*zeta/m*dt - k/m*dt*(R+R1*2+R2*2+R3)/6.
    R4 = R + (V+V1*2+V2*2+V3)/6.*dt
    R  = R4
    V  = V4
    Rs[0:dim,i]=R
    Vs[0:dim,i]=V
    time[i]=i*dt
    Et[i]=0.5*m*np.linalg.norm(V)**2+0.5*k*np.linalg.norm(R)**2
    particles.set_data(R[0], R[1])
    line.set_data(Rs[0,0:i], Rs[1,0:i])
    title.set_text(r"$t = {0:.2f},E_T = {1:.3f}$".format(i*dt,Et[i]))
    return particles,line,title
```

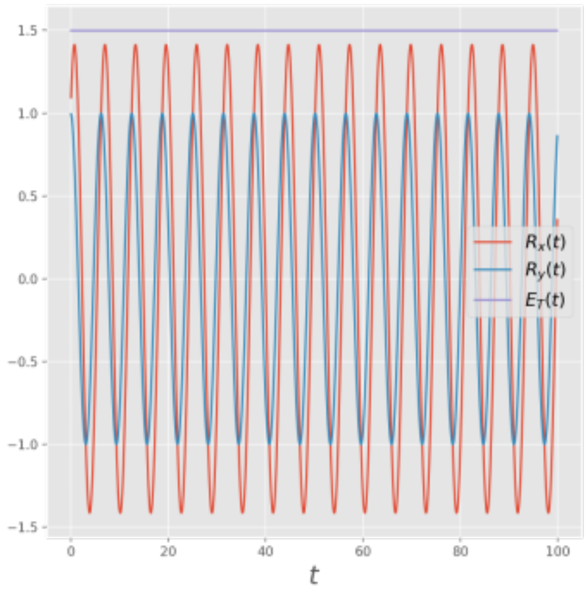
A14-1

- C24

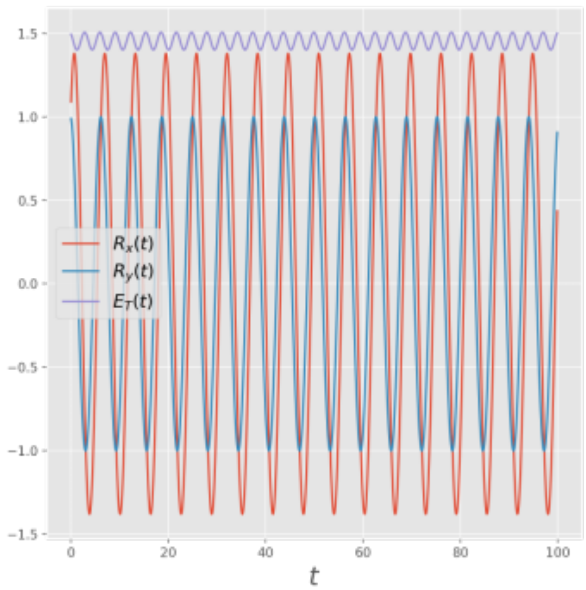
Q14-2

Perform the simulation using the modified code with the 4th order Runge-Kutta method, and make the same graph to the one which we made with the Euler method in the video lessen. Which of the following graphs (G31, G32, G33, G34) is the closest to what you obtained?

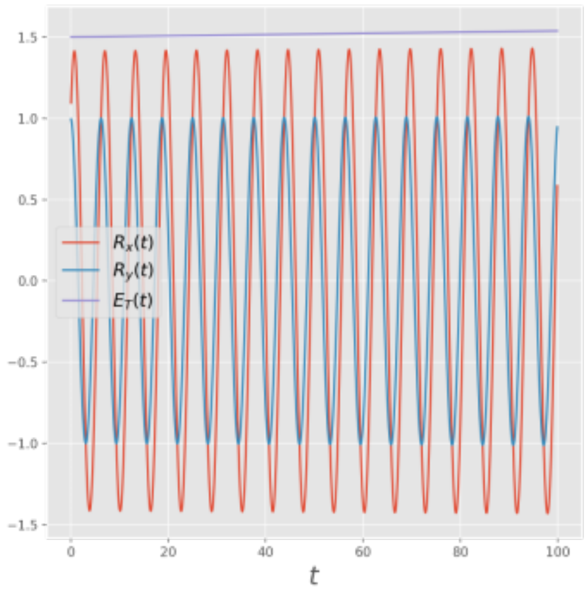
- G31



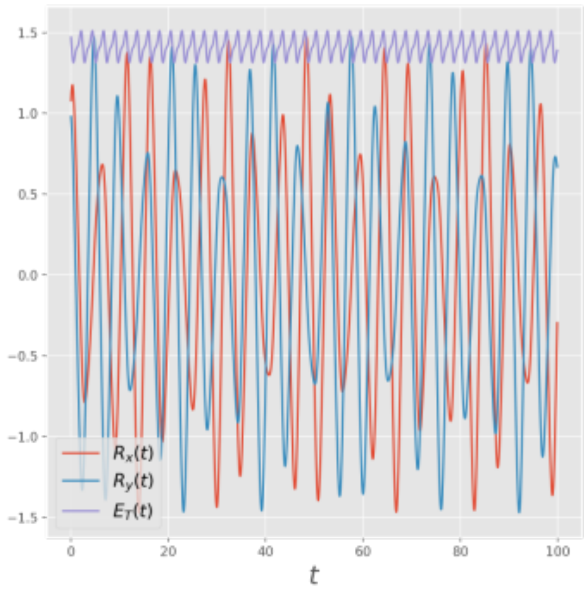
- G32



- G33



- G34



A14-2

- G31

In [14]:

```
% matplotlib nbagg
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
plt.style.use('ggplot')
def init(): # initialize animation
    particles.set_data([], [])
    line.set_data([], [])
    title.set_text(r'')
    return particles,line,title
```

In [15]:

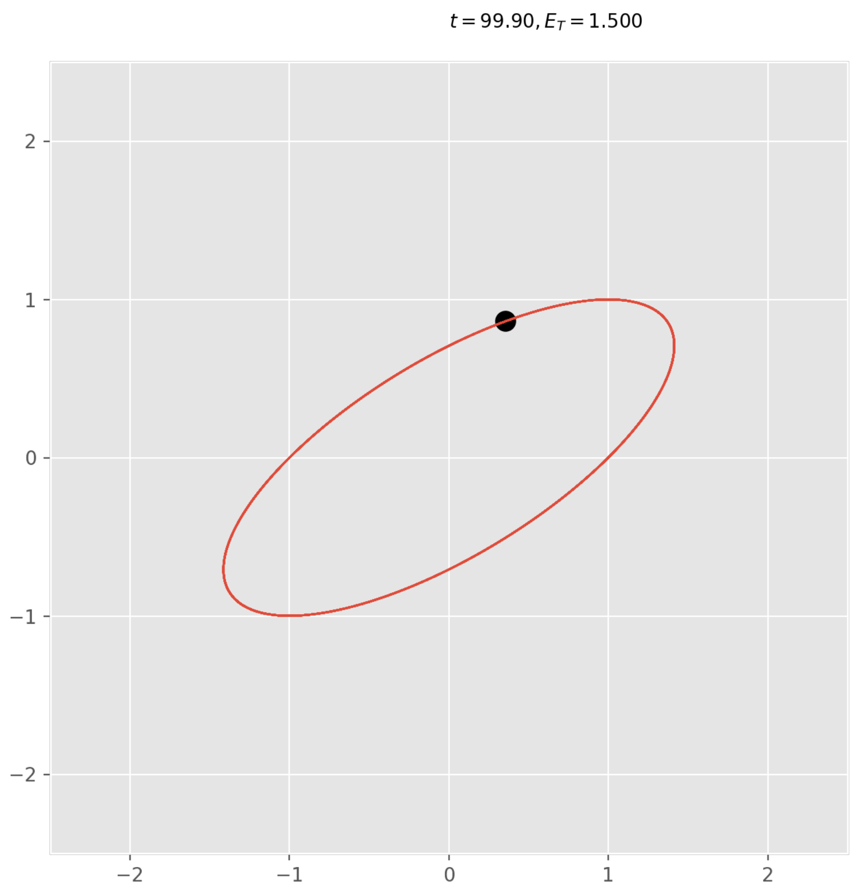
```
# Euler method
def animate(i): # define amination
    global R,V,F,Rs,Vs,time,Et
    V = V*(1-zeta/m*dt)-k/m*dt*R # Euler method Eq.(B4)
    R = R + V*dt                 # Euler method Eq.(B3)
    Rs[0:dim,i]=R
    Vs[0:dim,i]=V
    time[i]=i*dt
    Et[i]=0.5*m*np.linalg.norm(V)**2+0.5*k*np.linalg.norm(R)**2
    particles.set_data(R[0], R[1])      # current position
    line.set_data(Rs[0,0:i], Rs[1,0:i]) # add latest position Rs
    title.set_text(r"$t = {0:.2f},E_T = {1:.3f}$".format(i*dt,Et[i]))
    return particles,line,title
```

In [15]:

```
# 2nd order Runge-Kutta method
R1    = np.zeros(dim)
V1    = np.zeros(dim)
def animate(i): # define amination
    global R,V,F,Rs,Vs,time,Et
    V1 = V - zeta/m*0.5*dt*V - k/m*0.5*dt*R # RK 2nd
    R1 = R + V*0.5*dt # RK 2nd
    V = V - V1*zeta/m*dt - k/m*dt*R1 # RK 2nd
    R = R + V1*dt # RK 2nd
    Rs[0:dim,i]=R
    Vs[0:dim,i]=V
    time[i]=i*dt
    Et[i]=0.5*m*np.linalg.norm(V)**2+0.5*k*np.linalg.norm(R)**2
    particles.set_data(R[0], R[1]) # current position
    line.set_data(Rs[0,0:i], Rs[1,0:i]) # add latest position Rs
    title.set_text(r"$t = {0:.2f},E_T = {1:.3f}$".format(i*dt,Et[i]))
    return particles,line,title
```

In [16]:
```
# 4th order Runge-Kutta method
R1    = np.zeros(dim)
V1    = np.zeros(dim)
R2    = np.zeros(dim)
V2    = np.zeros(dim)
R3    = np.zeros(dim)
V3    = np.zeros(dim)
R4    = np.zeros(dim)
V4    = np.zeros(dim)
def animate(i):
    global R,V,F,Rs,Vs,time,Et
    V1 = V - zeta/m*0.5*dt*V - k/m*0.5*dt*R
    R1 = R + V*0.5*dt
    V2 = V - zeta/m*0.5*dt*V1 - k/m*0.5*dt*R1
    R2 = R + V1*0.5*dt
    V3 = V - zeta/m*dt*V2 - k/m*dt*R2
    R3 = R + V2*dt
    V4 = V - (V+V1*2+V2*2+V3)/6.*zeta/m*dt - k/m*dt*(R+R1*2+R2*2+R3)/6.
    R4 = R + (V+V1*2+V2*2+V3)/6.*dt
    R  = R4
    V  = V4
    Rs[0:dim,i]=R
    Vs[0:dim,i]=V
    time[i]=i*dt
    Et[i]=0.5*m*np.linalg.norm(V)**2+0.5*k*np.linalg.norm(R)**2
    particles.set_data(R[0], R[1]) # current position
    line.set_data(Rs[0,0:i], Rs[1,0:i]) # add latest position Rs
    title.set_text(r"$t = {0:.2f},E_T = {1:.3f}$".format(i*dt,Et[i]))
    return particles,line,title
```

In [17]:
```
# System parameters
# particle mass, spring & friction constants
m, k, zeta = 1.0, 1.0, 0.0
# Initial condition
R[0], R[1] = 1., 1. # Rx(0), Ry(0)
V[0], V[1] = 1., 0. # Vx(0), Vy(0)
dt   = 0.1*np.sqrt(k/m) # set \Delta t
box  = 5 # set size of draw area
# set up the figure, axis, and plot element for animatation
fig, ax = plt.subplots(figsize=(7.5,7.5)) # setup plot
ax = plt.axes(xlim=(-box/2,box/2),ylim=(-box/2,box/2)) # draw range
particles, = ax.plot([],[],'ko', ms=10) # setup plot for particle
line,=ax.plot([],[],lw=1) # setup plot for trajectry
title=ax.text(0.5,1.05,r'',transform=ax.transAxes,va='center') # title
anim=animation.FuncAnimation(fig,animate,init_func=init,
     frames=nums,interval=5,blit=True,repeat=False) # draw animation
# anim.save('movie.mp4',fps=20,dpi=400)
```



In [18]:

```
fig, ax = plt.subplots(figsize=(7.5,7.5))
ax.set_xlabel(r"$t$", fontsize=20)
ax.plot(time,Rs[0]) # plot R_x(t)
ax.plot(time,Rs[1]) # plot R_y(t)
ax.plot(time,Et) # plot E(t) (ideally constant if \deta=0)
ax.legend([r'$R_x(t)$',r'$R_y(t)$',r'$E_T(t)$'], fontsize=14)
plt.show()
```