In [ ]:
```
import numpy as np
from numpy import sin,cos,pi
from scipy.linalg import *
from scipy.optimize import fsolve,root
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
```

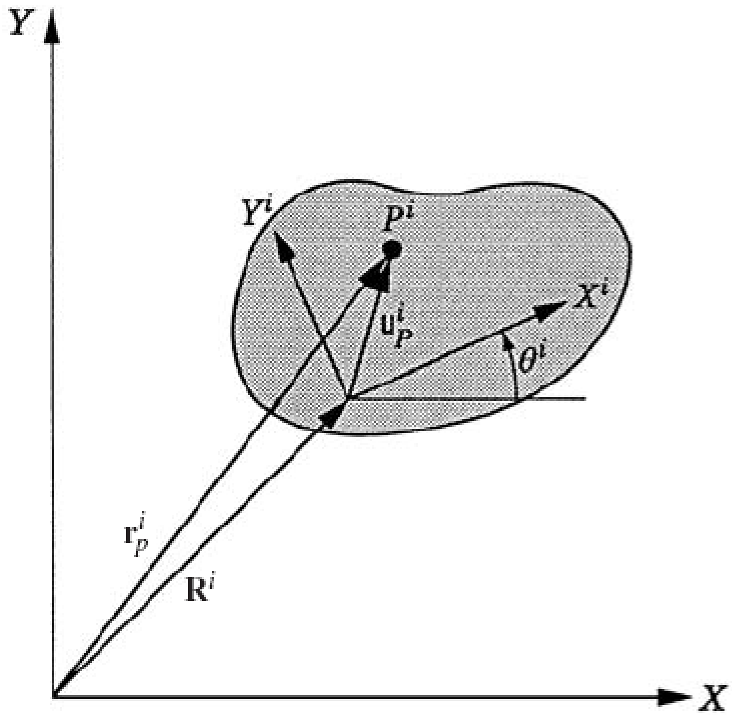
In [ ]:
```
from IPython.display import YouTubeVideo
YouTubeVideo('-A1w46iqUlE')
```

Out[ ]:

Homework #7 Kinematics

Kinematics is the study of the geometry of motion e.g. definitions of position, velocity, and acceleration

In this notebook we will explore kinematically-driven systems where the system degrees of freedom, $n_d = 0 = 3 \times n_b - n_c$, for planar problems. n_b bodies moving in a plane have $3 \times n_b$ degrees of freedom and the number of constraints is n_c.



https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9780470686157/files/figs/0303.png

In the figure above, there are three position vectors, $\mathbf{r}^i$, $\mathbf{R}^i$, and $\mathbf{u}^i$ and two coordinate systems, X-Y and X^i-Y^i.

The X^i-Y^i coordinate system moves with the rigid body and the point P is always in a fixed position $\bar{\mathbf{u}}_P^i = \bar{x}_P^i\hat{i}^i + \bar{y}_P^i\hat{j}^i$ in this coordinate system.

$$\mathbf{u}_P^i = \begin{bmatrix} \cos\theta^i & -\sin\theta^i \\ \sin\theta^i & \cos\theta^i \end{bmatrix} \begin{bmatrix} \bar{x}_P^i \\ \bar{y}_P^i \end{bmatrix}$$

or

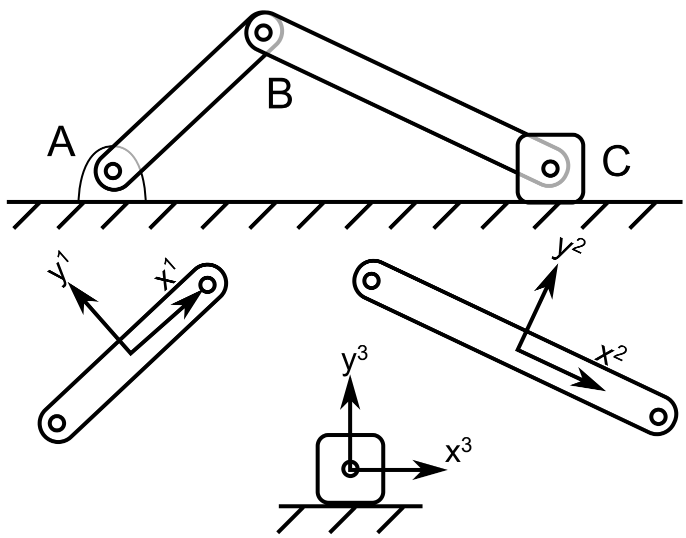
$$\mathbf{u}_P^i = \mathbf{A}^i\bar{\mathbf{u}}_P^i$$

In [ ]:
```
def rotA(theta):
    '''This function returns a 2x2 rotation matrix to convert the
    rotated coordinate to the global coordinate system
    input is angle in radians

    Parameters
    ----------
    theta : angle in radians

    Returns
    -------
    A : 2x2 array to rotate a coordinate system at angle theta to global x-y
    '''
    A=np.zeros((2,2))
    A=np.array([[np.cos(theta), -np.sin(theta)],
                [np.sin(theta), np.cos(theta)]])
```

```
    return A
```

In [ ]:
```
rotA(np.pi/3)
```

Out[ ]:
```
array([[ 0.5       , -0.8660254],
       [ 0.8660254,  0.5       ]])
```



Figs. Slider crank mechanism and body coordinate systems.

Computational Kinematics of Slider crank

Here you will create the computational kinematics of the slider crank in Fig. 3.35-3.36 above.

The first kinematic problem will drive the slider crank with a constraint

$$\theta^1 = \omega t + \theta_0$$

where $\omega = 150\ rad/s$ and $\theta_0 = \pi/6\ rad$.

Below you set up the function to return the constraint equations,

$\mathbf{C}(\mathbf{q}, t)$ = `C_slidercrank(q,t)`

In [ ]:
```
def links(l1 = 0.075*2, l2 = 0.125*2):
    '''function to define lengths of links for bodies 2 and 3
    in Fig. 3.35-3.36

    Parameters
    ----------
    l1 : length of body one, default 0.150 m
    l2 : lenght of body two, default 0.250 m
    Returns
    -------
    l1, l2 : link lengths for bodies 1 and 2
```

```
    '''

    return l1,l2
```

In [ ]:
```
def C_slidercrank(q,t):
    '''9 constraint equations for 9 generalized coords
       q=[R1x,R1y,a1,R2x,R2y,a2,R3x,R3y,a3]
       q=[R1,     a1,  R2,  ,a2,  R3,  ,a3]
         [0,1     2    3,4   5    6,7   8 ]

          1/\2
          /  \    slider-crank mechanism
         O   |3|
        ^^^-------------

    Parameters
    ----------
    q : numpy array for 9 generalized coordinates for bodies 1-3 in the slider cran
        q = [q1, q2, q3]
    t : current time

    Returns
    -------
    C : 9 constraint equation evaluations
    '''
    l1,l2=links()
    q1 = q[0:3]
    q2 = q[3:6]
    q3 = q[6:9]

    C=np.zeros(9)
    C[0:2] = q1[0:2]+rotA(q1[2])@np.array([-l1/2, 0])
    C[2:4] = q1[0:2]-q2[0:2]+rotA(q1[2])@np.array([l1/2, 0])-rotA(q2[2])@np.array([
    C[4:6] = q2[0:2]-q3[0:2]+rotA(q2[2])@np.array([l2/2, 0])-rotA(q3[2])@np.array([
    C[6] = q3[1]
    C[7] = q3[2]
    C[8] = q1[2] - pi/6 - 150*t
    return C
```

Problem 1

Solve for $\mathbf{q}(t=0) = [R_x^1,\ R_y^1, \theta^1,\ R_x^2,\ R_y^2, \theta^2,\ R_x^3,\ R_y^3, \theta^3]$ using the given system definitions:

- $l_1 = 0.15\ m$
- $l_2 = 0.25\ m$
- $\theta^1(t) = 150t + \frac{\pi}{6}$

Show that `C_slidercrank(q0, 0)` $= \mathbf{0}$.

In [ ]:
```
def C_slidercrank_2(q,t):
    '''9 constraint equations for 9 generalized coords
```

```
        q=[R1x,R1y,a1,R2x,R2y,a2,R3x,R3y,a3]
        q=[R1,     a1,  R2,  ,a2,  R3,  ,a3]
          [0,1     2    3,4   5    6,7   8 ]

           1/\2
           /  \    slider-crank mechanism
          O   |3|
         ^^^-------------

    Parameters
    ----------
    q : numpy array for 9 generalized coordinates for bodies 1-3 in the slider cran
        q = [q1, q2, q3]
    t : current time

    Returns
    -------
    C : 9 constraint equation evaluations
    '''
    l1 = 0.15
    l2 = 0.25
    q1 = q[0:3]
    q2 = q[3:6]
    q3 = q[6:9]

    C=np.zeros(9)
    C[0:2] = q1[0:2]+rotA(q1[2])@np.array([-l1/2, 0])
    C[2:4] = q1[0:2]-q2[0:2]+rotA(q1[2])@np.array([l1/2, 0])-rotA(q2[2])@np.array([
    C[4:6] = q2[0:2]-q3[0:2]+rotA(q2[2])@np.array([l2/2, 0])-rotA(q3[2])@np.array([
    C[6] = q3[1]
    C[7] = q3[2]
    C[8] = q1[2] - pi/6 - 150*t
    return C

func = lambda q,t: C_slidercrank_2(q, t)
sol = fsolve(func, np.zeros(9), 0)

print(sol)

np.round(C_slidercrank_2(sol, 0), 2) == 0
```

```
[ 0.06495191  0.0375      0.52359878  0.24914621  0.0375     -0.30469265
  0.36838861  0.          0.        ]
```

Out[ ]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True])
```

Set up solution for $\mathbf{C}(\mathbf{q},\ t)$

Next, set up the 9×9 Jacobian of

1. Set up the $\mathbf{A}_\theta$ function as `A_theta`
2. each pin $\mathbf{C}_{\mathbf{q},\ \mathbf{pin}} = \frac{\partial \mathbf{C}_{\mathbf{pin}}}{\partial \mathbf{q}}$ = `Cq_pin`
3. the total system: $\mathbf{C}_{\mathbf{q}} = \frac{\partial \mathbf{C}}{\partial \mathbf{q}}$ = `Cq_slidercrank`

In [ ]:
```
def A_theta(theta):
    '''This function returns a 2x2 rotation matrix derivative
    input is angle in radians

    Parameters
    ----------
    theta : angle in radians

    Returns
    -------
    dAda : 2x2 array derivative of `rotA`
    '''
    dAda=np.array([[-np.sin(theta), -np.cos(theta)],
                   [np.cos(theta), -np.sin(theta)]])
    return dAda
```

In [ ]:
```
def Cq_pin(qi, qj, ui, uj):
    '''Jacobian of a pinned constraint for planar motion

    Parameters
    ----------
    qi : generalized coordinates of the first body, i [Rxi, Ryi, thetai]
    qj : generalized coordinates of the 2nd body, i [Rxj, Ryj, thetaj]
    ui : position of the pin the body-i coordinate system
    uj : position of the pin the body-j coordinate system

    Returns
    -------
    Cq_pin : 2 rows x 6 columns Jacobian of pin constraint Cpin
    '''

    Cq_1=np.block([np.eye(2), A_theta(qi[2])@ui[:,np.newaxis] ])
    Cq_2=np.block([-np.eye(2), -A_theta(qj[2])@uj[:,np.newaxis] ])
    Cq_pin=np.block([Cq_1, Cq_2])
    return Cq_pin
```

In [ ]:
```
def Cq_slidercrank(q,t):
    '''return Jacobian of C_slidercrank(q,t) = dC/dq_i
       |dC1/dR1x dC1/dR1y ... dC9/da3 |
       |dC2/dR1x dC2/dR1y ... dC9/da3 |
       |... ..      .          ...    |
       |             .                |
       |              .               |
       |dC9/dR1x ...         dC9/da3  |
    Parameters
    ----------
    q : numpy array for 9 generalized coordinates for bodies 1-3 in the slider cran
        q = [q1, q2, q3]
    t : current time
    Returns
    -------
    Cq : 9 rows x 9 columns Jacobian of constraints `C_slidercrank`
    '''
    l1, l2 = links()
    q1 = q[0:3]
```

```
    q2 = q[3:6]
    q3 = q[6:9]

    Cq=np.zeros((9,9))
    Cq[0:2, 0:3] = Cq_pin(q1, np.array([0, 0, 0]),np.array([-l1/2, 0]),np.array([0,
    Cq[2:4, 0:6] = Cq_pin(q1, q2, np.array([l1/2, 0]), np.array([-l2/2, 0]))
    Cq[4:6, 3:10] = Cq_pin(q2, q3, np.array([l2/2, 0]), np.array([0, 0]))
    Cq[6:8, 7:10] = np.eye(2)
    Cq[8, 2]=1
    return Cq
```

Solve for $\mathbf{q(t)}$

Now, you solve for 1 full rotation of the driven crank.

t= 0-360^o = 0-2π/150

The solution requires an initial guess for the generalized coordinates, $\mathbf{q}$, set as `q0`. It is updated at each timestep to find the next solution. Here, you use the Jacobian of $\mathbf{C}$, $\mathbf{C_q}$, by specifying the `fprime = lambda q: Cq_slidercrank`.

```
In [ ]: t = np.linspace(0, 2*pi/150)
        q0 = np.array([0, 0.5, pi/6, 0, 0.5, pi/3, 0.5, 0, 0])
        q = np.zeros((len(q0), len(t)))
        q[:, 0] = q0
        for i,tt in enumerate(t):
            q[:,i]=fsolve(lambda q: C_slidercrank(q,tt),q0,\
                        fprime= lambda q: Cq_slidercrank(q,tt)) # <-- use the Jacobian
            q0=q[:,i]
```

Now, you can create the same figures as Shabana ch 3

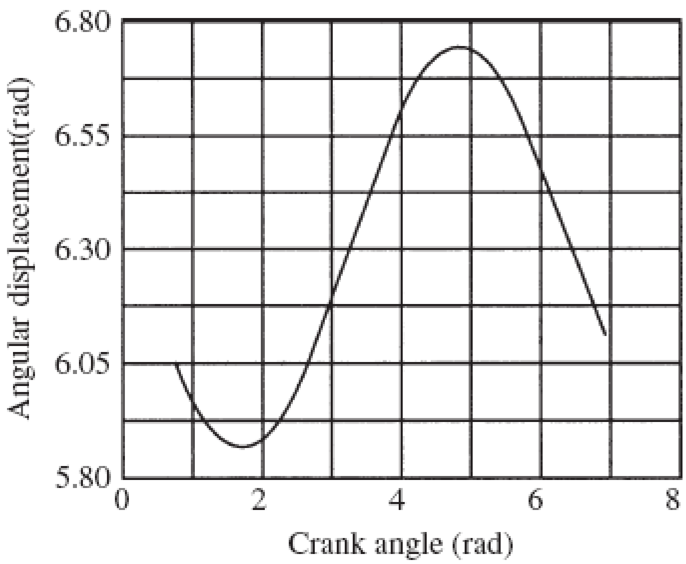


Fig. 3.37. Orientation of the connecting rod

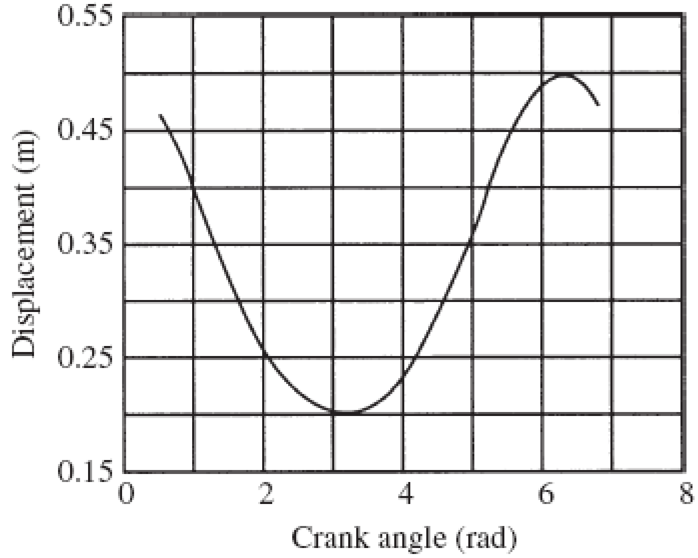


Fig. 3.37. Displacement of the slider block

In [ ]:
```
plt.plot(q[2,:],q[6,:])
#plt.plot(t,q[5,:]/pi*180)
```

Out[ ]: [<matplotlib.lines.Line2D at 0x2010da492e0>]

Problem 2

Recreate the displacement of the slider block graph in Fig. 3.38 from your solution.

In [ ]:
```
## your work here
plt.plot(q[2,:],q[5,:] + 2*pi)
```

Out[ ]: `[<matplotlib.lines.Line2D at 0x2010f0def10>]`

Animate the motion for constant $\dot{\theta}^1$

Next, you animate the motion of the system. Below, you create

1. `plot_shape` to create lines and markers to represent links and the sliding base
2. a figure that shows the path of the two link centers of mass

3. `init` to initialize the animation 4. `animate` to update the two links and sliding base 5. `FuncAnimation` to display the motion of the slidercrank system

In [ ]:
```
def plot_shape(shape,dims,q):
    '''
    function to plot a shape based upon the shape dimensions and coordinates
    arguments:
    ----------
    shape: either 'link' or 'base',
            - the link returns two points to plot as a line
            - the base returns one point to plot as a marker
            - if neither 'link' or 'base' are chosen, then 0 is returned and warnin
            `choose a \'link\' or \'base\' please`
    dims: the dimensions of the shape
            - the link uses the first value as the length
```

```
            - the base ignores the `dims`
    q: generalized coordinates in the form [Rx, Ry, theta]
            - the link returns the center of the link at (Rx, Ry) and oriented at t
            - the base returns the center at (Rx, Ry) and ignores theta
    returns:
    --------
    datax: coordinates to plot the x-positions
    datay: coordinates to plot the y-positions
            - the link returns array of length 2
            - the base returns array of length 1

    '''

    if shape=='link':
        left = rotA(q[2])@np.array([-dims[0]/2, 0])
        right = rotA(q[2])@np.array([dims[0]/2, 0])
        Px=q[0]+np.array([left[0], right[0]])
        Py=q[1]+np.array([left[1], right[1]])
        datax = Px
        datay = Py
        #l,= plt.plot(Px,Py,'o-')
        return datax, datay
    elif shape=='base':
        Px=q[0]
        Py=q[1]
        data = [Px, Py]
        #l,=plt.plot(Px,Py,'s',markersize=20)
        return data
    else:
        print('choose a \'link\' or \'base\' please')
        return 0
```

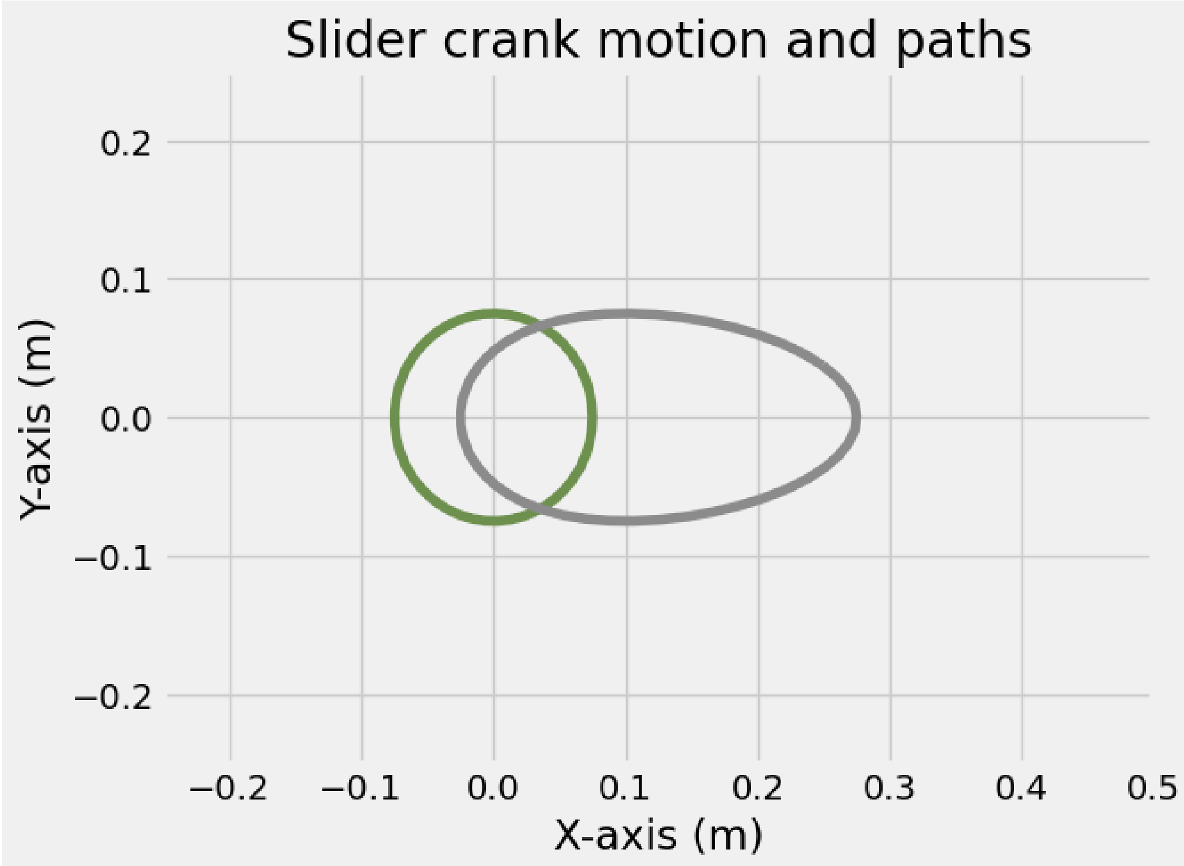
2. initialize the lines and coordinate system

In [ ]:

```
q1 = q[0:3, :]
q2 = q[3:6, :]
q3 = q[6:9, :]
l1, l2 = links()

fig, ax = plt.subplots()
link1, = ax.plot([], [], linewidth = 10)
link2, = ax.plot([], [], linewidth = 10)
body3, = ax.plot([], [], 's', markersize = 20)
path1, = ax.plot(q1[0, :], q1[1, :])
path2, = ax.plot(q2[0, :], q2[1, :])
ax.set_xlim((-0.25, 0.5))
ax.set_ylim((-0.25, 0.25))
ax.set_xlabel('X-axis (m)')
ax.set_ylabel('Y-axis (m)')
ax.set_title('Slider crank motion and paths')
```

Out[ ]:

```
Text(0.5, 1.0, 'Slider crank motion and paths')
```



3. and 4. create your `init` and `animation` functions to update the lines on the plot

Create an initializing (`init`) function that clears the previous lines and markers

Create an animating (`animate`) function that updates the link, base, and path

In [ ]:
```
def init():
    link1.set_data([], [])
    link2.set_data([], [])
    body3.set_data([], [])
    return (link1, link2, body3)
```

In [ ]:
```
def animate(i):
    '''function that updates the line and marker data
    arguments:
    ----------
    i: index of timestep
    outputs:
    --------
    link1: the line object plotted in the above ax.plot(...)
    link2: the line object plotted in the above ax.plot(...)
    body3: the marker for the piston in the slider-crank
    '''
    l1, l2 = links()
    datax, datay = plot_shape('link', np.array([l1]), q1[:, i])
    link1.set_data(datax, datay)
```

```
    datax, datay = plot_shape('link', np.array([l2]), q2[:, i])
    link2.set_data(datax, datay)
    pinx, piny = plot_shape('base', [], q3[:,i])
    body3.set_data(pinx, piny)
    return (link1, link2, body3, )
```

4. display the result in an HTML video

Import the `animation` and `HTML` functions. Then, create an animation (`anim`) variable using the `animation.FuncAnimation`

In [ ]:
```
from matplotlib import animation
from IPython.display import HTML
```

In [ ]:
```
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=range(0,len(t)), interval=50,
                               blit=True)
```

In [ ]:
```
HTML(anim.to_html5_video())
```

Out[ ]:

Velocity and Acceleration

Differentiating the constraint equations, $\mathbf{Cq} = \mathbf{0}$,

$\mathbf{C_q}\dot{\mathbf{q}} + \mathbf{C}_t = \mathbf{0}$ (3.119)

where

$\mathbf{C}_t = \left[\frac{\partial C_1}{\partial t}\frac{\partial C_2}{\partial t}\dots\frac{\partial C_n}{\partial t}\right]^T$ (3.120)

Solve for velocity $\dot{\mathbf{q}}_\mathbf{i}$ as such

$\mathbf{C_q}\dot{\mathbf{q}} = -\mathbf{C}_t$ (3.121)

Differentiating $\mathbf{Cq} = \mathbf{0}$ twice leads to the acceleration equation

$\mathbf{C_q}\ddot{\mathbf{q}} + (\mathbf{C_q}\dot{\mathbf{q}})_\mathbf{q}\dot{\mathbf{q}} + 2\mathbf{C}_{\mathbf{q}t}\dot{\mathbf{q}} + \mathbf{C}_{tt} = \mathbf{0}$ (3.123)

Solve for acceleration $\ddot{\mathbf{q}}$ as such

$\mathbf{C_q}\ddot{\mathbf{q}} = \mathbf{Q}_d$

where

$\mathbf{Q}_d = -(\mathbf{C_q}\dot{\mathbf{q}})_\mathbf{q}\dot{\mathbf{q}} - 2\mathbf{C}_{\mathbf{q}t}\dot{\mathbf{q}} - \mathbf{C}_{tt}$

For the current slider crank system,

$$\mathbf{Q}_d = \left[\begin{array}{c} (\dot{\theta}^1)^2\mathbf{A}^1\bar{\mathbf{u}}_A^1 \\ (\dot{\theta}^1)^2\mathbf{A}^1\bar{\mathbf{u}}_B^1 - (\dot{\theta}^2)^2\mathbf{A}^2\bar{\mathbf{u}}_B^2 \\ (\dot{\theta}^2)^2\mathbf{A}^2\bar{\mathbf{u}}_C^i \\ 0 \\ 0 \\ 0 \end{array}\right]$$

Here, you set up `vel_acc(q,t)` to return velocity and acceleration of $\mathbf{q_i}$ components as $\frac{d\mathbf{q}}{dt}$ and $\frac{d^2\mathbf{q}}{dt^2}$ (`dq` and `ddq`, respectively)

In [ ]:
```
def Qd_slidercrank(q, dq, t):
    '''return slidercrank Qd = Cq@ddq

    Parameters
    ----------
    q : numpy array for 9 generalized coordinates for bodies 1-3 in the slider cran
        q = [q1, q2, q3]
    t : current time
    Returns
    -------
    Qd : 1D array with length 9
    '''
```

```
    l1, l2 = links()
    q1 = q[0:3]
    q2 = q[3:6]
    q3 = q[6:9]
    dq1 = dq[0:3]
    dq2 = dq[3:6]
    dq3 = dq[6:9]

    Qd=np.zeros(9)
    Qd[0:2] = dq1[2]**2*rotA(q1[2])@np.array([-l1/2, 0])
    Qd[2:4] = dq1[2]**2*rotA(q1[2])@np.array([l1/2, 0]) -\
              dq2[2]**2*rotA(q2[2])@np.array([-l2/2, 0])
    Qd[4:6] = dq2[2]**2*rotA(q2[2])@np.array([l2/2, 0])
    Qd[6:9] = 0
    return Qd

def Ct_slidercrank(q, t):
    '''return slidercrank partial derivative of constraints dC/dt

    Parameters
    ----------
    q : numpy array for 9 generalized coordinates for bodies 1-3 in the slider cran
        q = [q1, q2, q3]
    t : current time
    Returns
    -------
    Ct : 1D array with length 9
    '''
    Ct = np.zeros(9)
    Ct[8] = -150
    return Ct
```

In [ ]:

```
t = np.linspace(0, 2*pi/150)
q0 = np.array([0, 0.5, pi/6, 0, 0.5, pi/3, 0.5, 0, 0])
q = np.zeros((len(q0), len(t)))
dq = np.zeros(q.shape)
ddq = np.zeros(q.shape)
q[:, 0] = q0
for i, ti in enumerate(t):
    q[:, i] = fsolve(lambda q: C_slidercrank(q, ti),q0,\
                     fprime= lambda q: Cq_slidercrank(q, ti)) # <-- use the Jacobian
    dq[:, i] = np.linalg.solve(Cq_slidercrank(q[:,i], ti), -Ct_slidercrank(q[:, i],
    Qd = Qd_slidercrank(q[:, i], dq[:, i], ti)
    ddq[:, i] = np.linalg.solve(Cq_slidercrank(q[:,i], ti), Qd)
    q0=q[:, i]
```

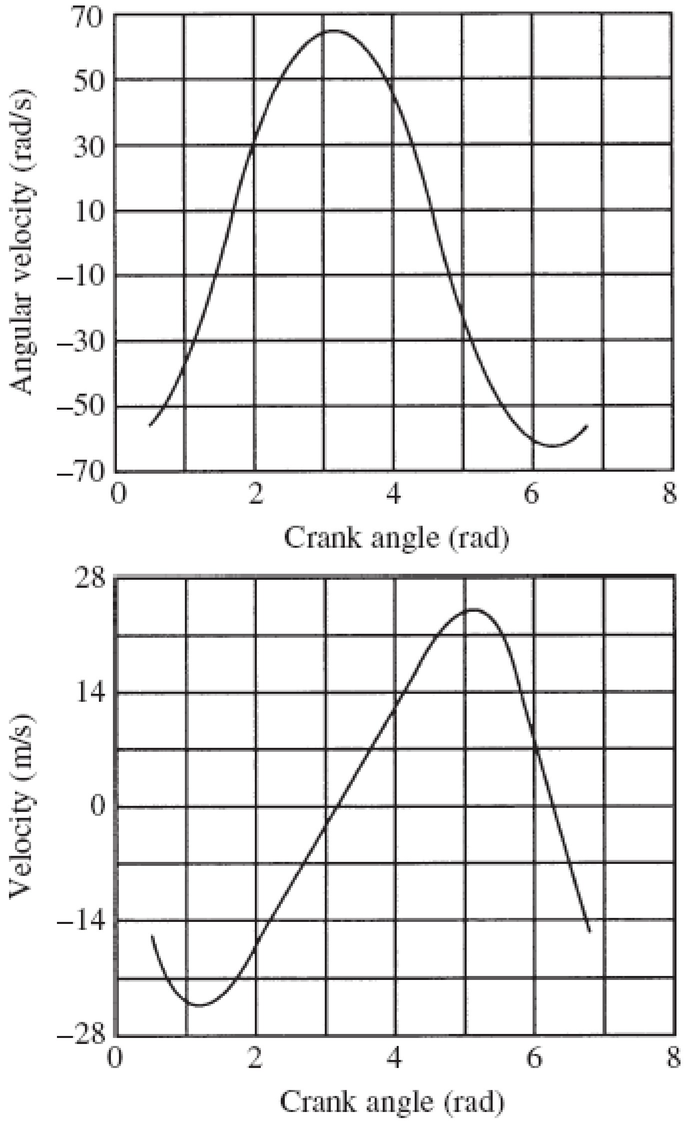


Fig. 3.38 velocity components

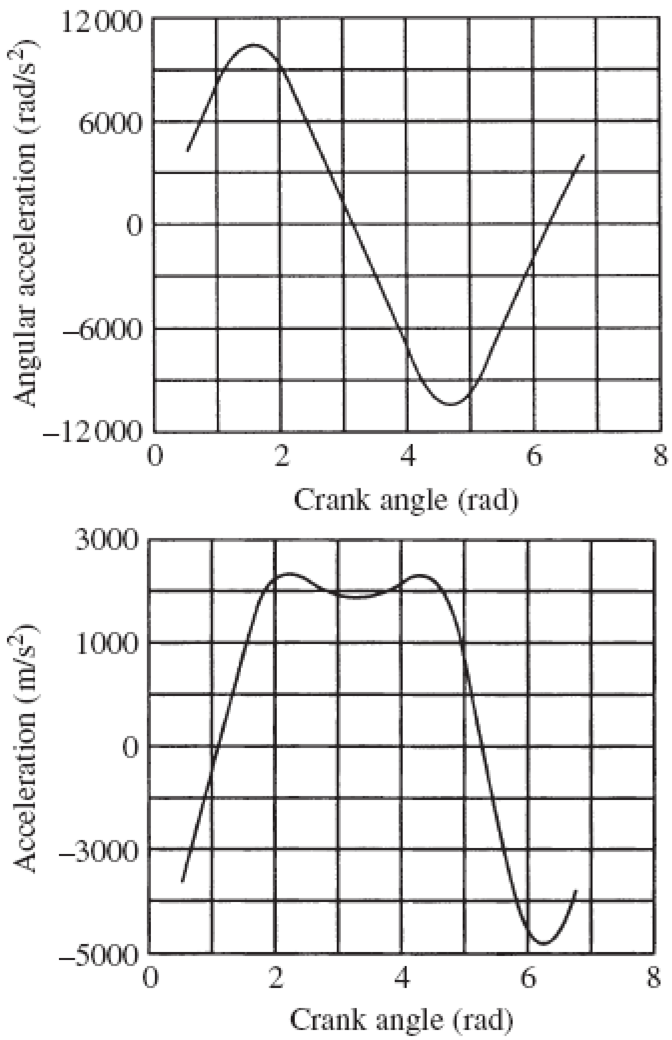


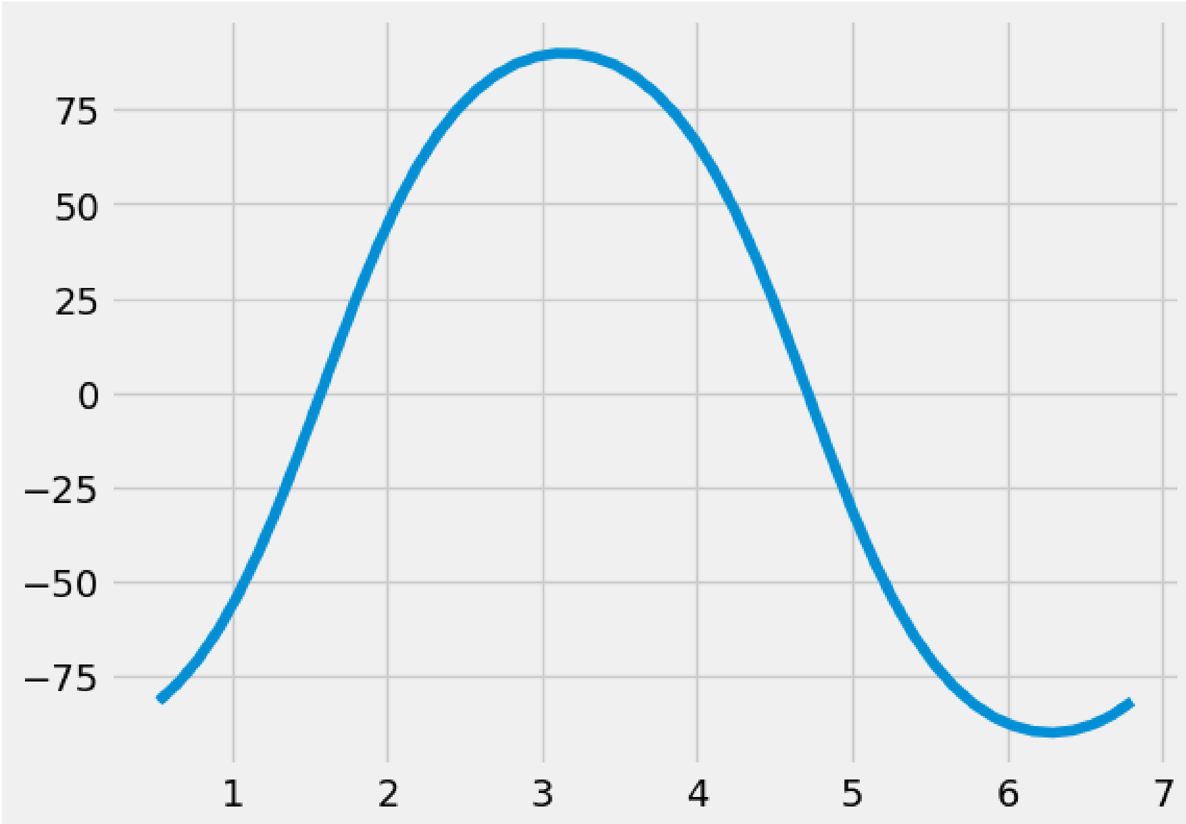
Fig. 3.39 acceleration components

Recreate the plots with your solution

Here, you can plot the terms $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ to compare to the Shabana solutions shown above in Figs 3.38-39. **Try plotting** $\dot{\theta}^2,\ \ddot{\theta}^2,\ \dot{R}_x^3,\ and\ \ddot{R}_x^3$

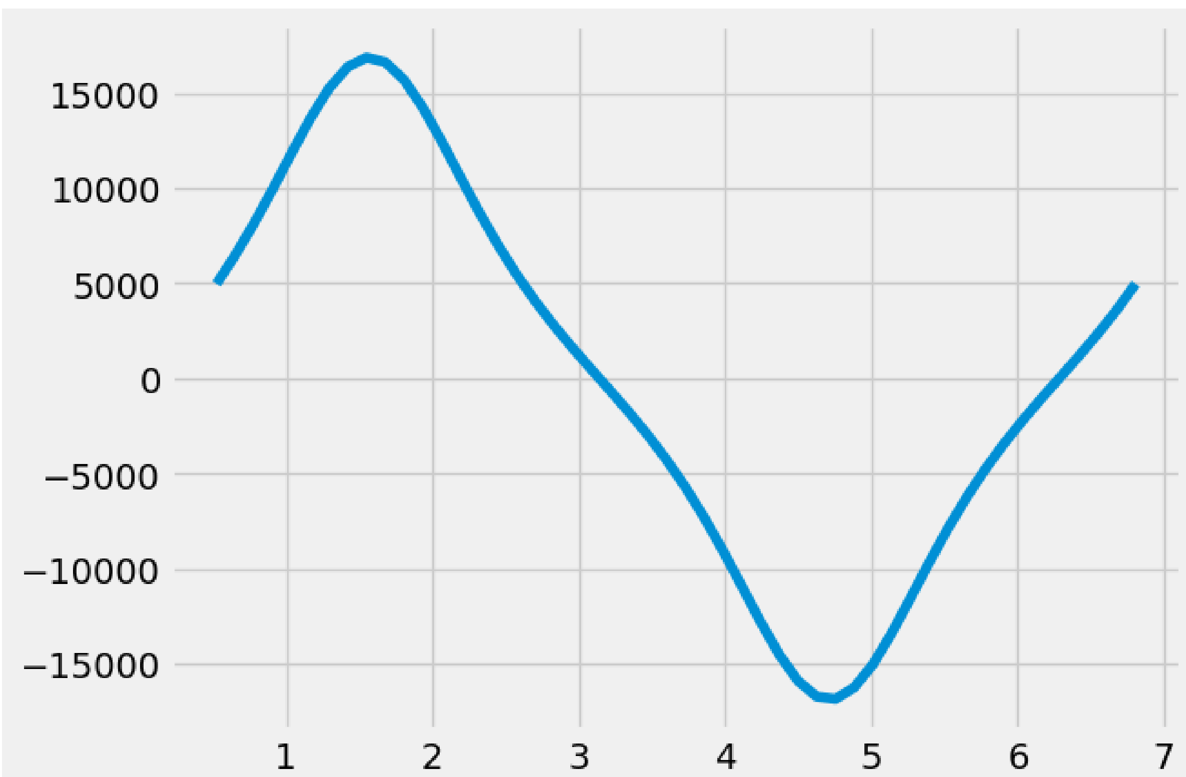
In [ ]:
```
plt.plot(q[2], dq[5, :])
```

Out[ ]: `[<matplotlib.lines.Line2D at 0x2010f455550>]`



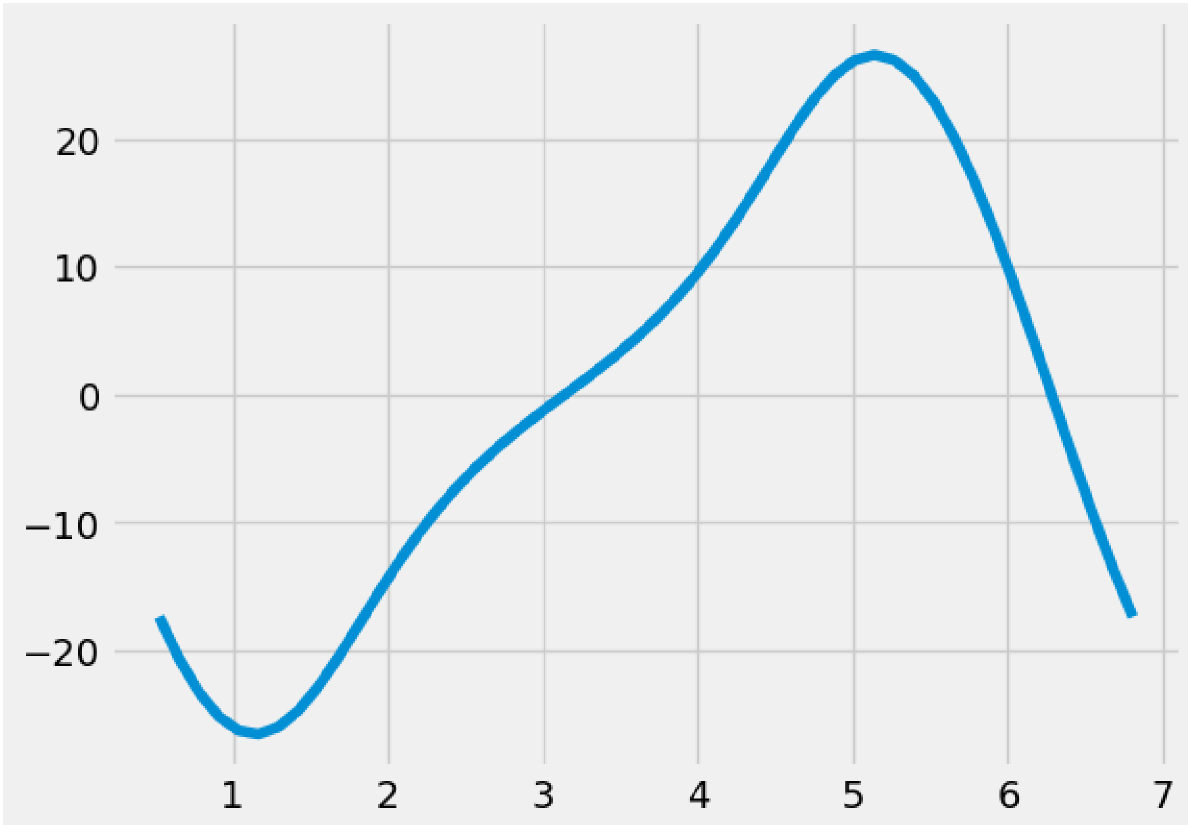
In [ ]:
```
plt.plot(q[2], ddq[5, :])
```

Out[ ]: [<matplotlib.lines.Line2D at 0x2010f4cb8e0>]



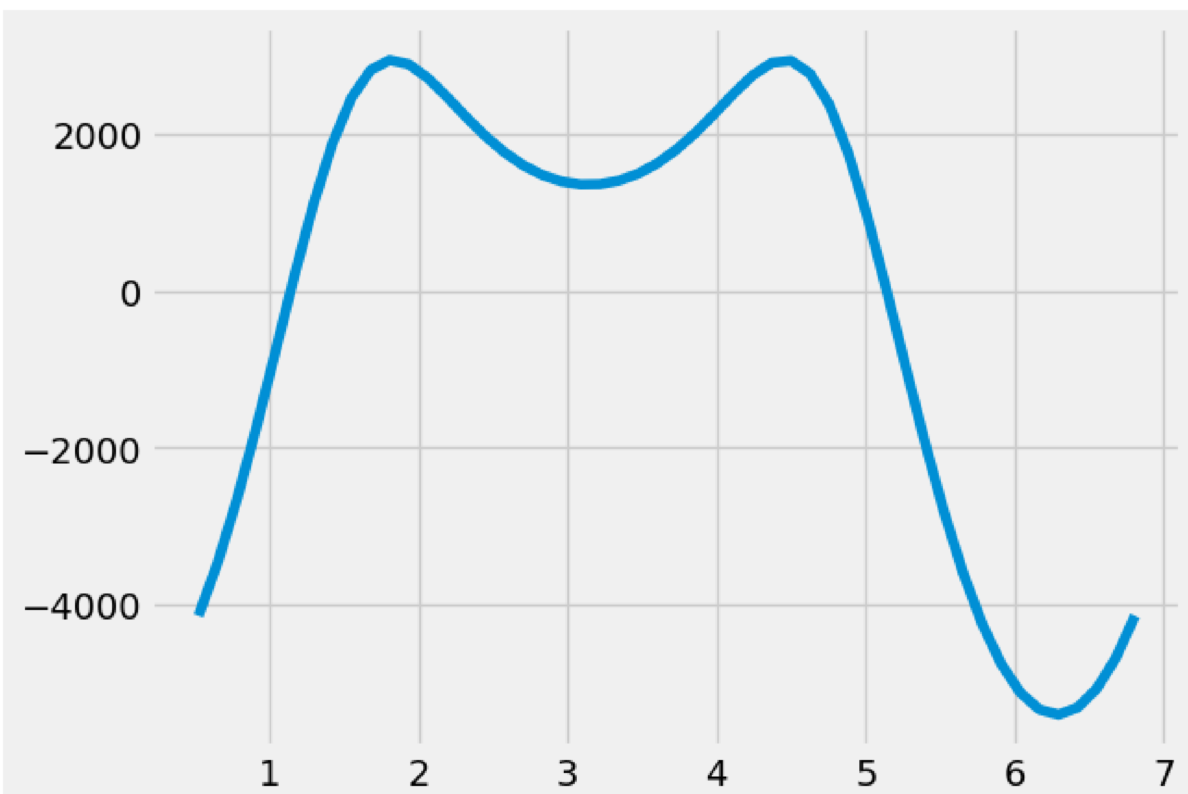
In [ ]:
```
plt.plot(q[2], dq[6, :])
```

Out[ ]: [<matplotlib.lines.Line2D at 0x2010f546580>]



In [ ]:
```
plt.plot(q[2], ddq[6, :])
```

Out[ ]: [<matplotlib.lines.Line2D at 0x201101cd700>]



Problem 3

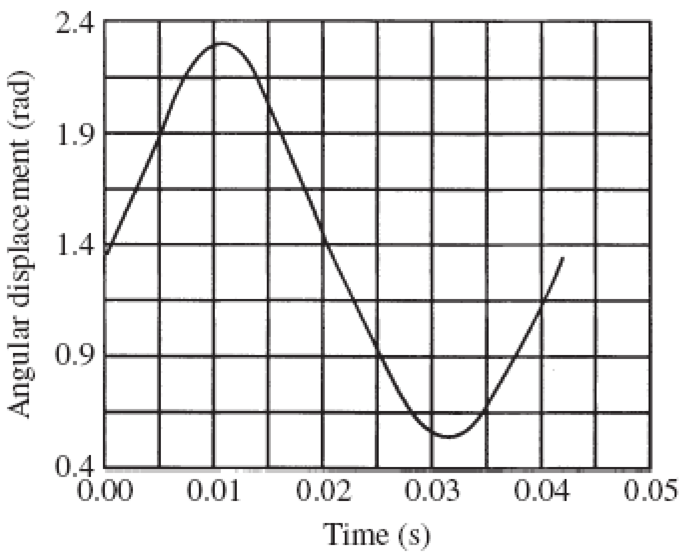
Change the constraints for the slidercrank such that

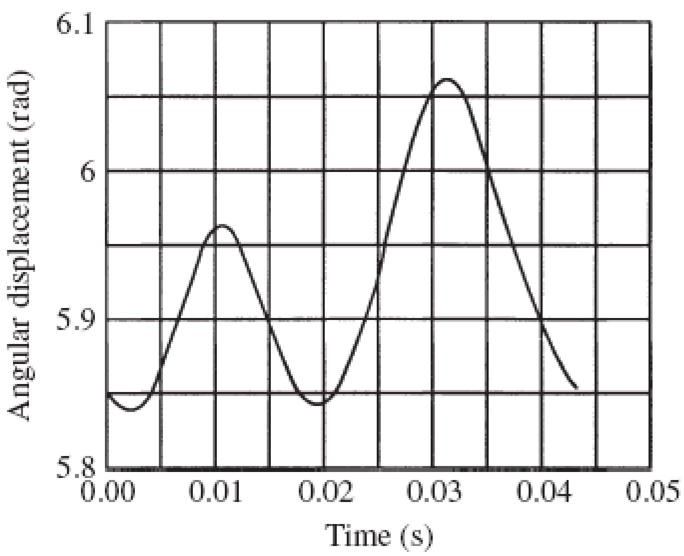
$$R_x^3 - f(t) = 0$$

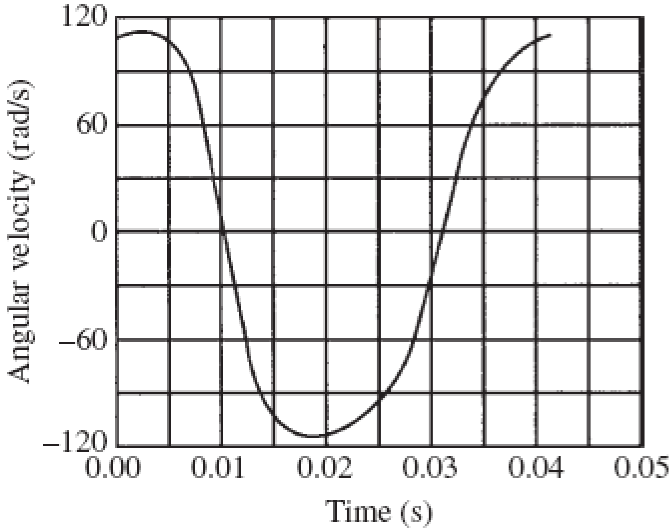
where

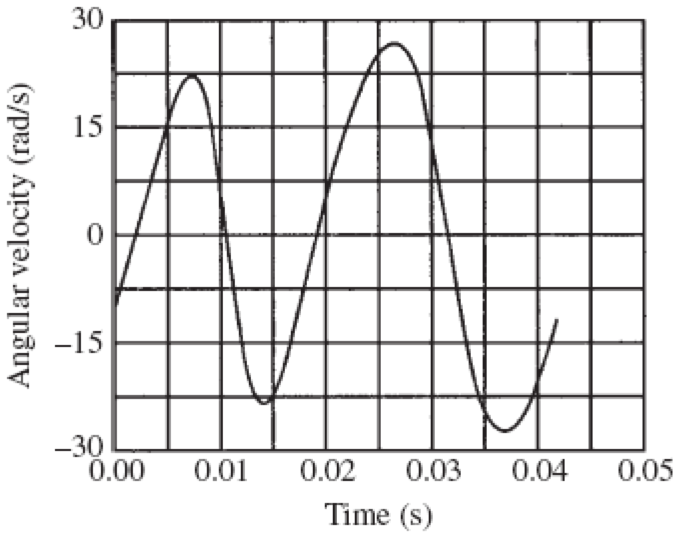
$$f(t) = 0.35 - 0.8l^2 \sin 150t$$

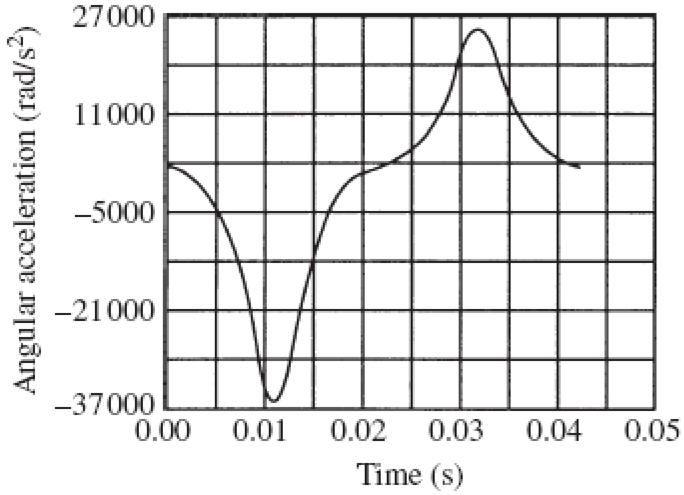
Recreate Figs. 3.43-3.48 for the slidercrank.

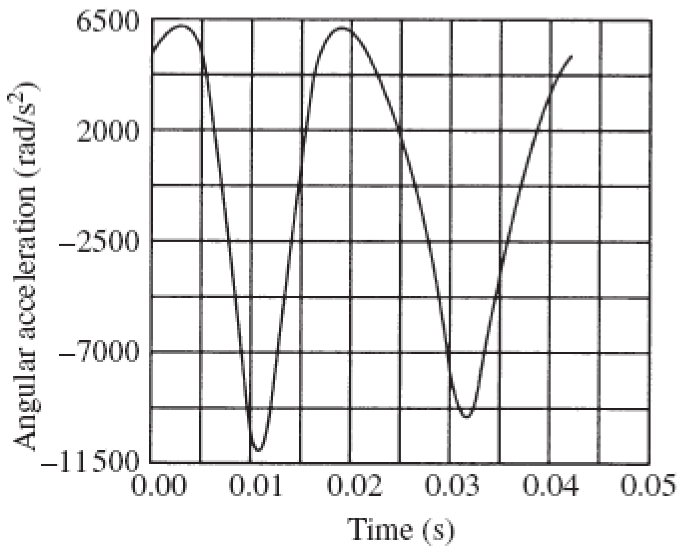





120
60
0
−60
−120
Angular velocity (rad/s)
0.00
0.01
0.02
0.03
0.04
0.05
Time (s)


30
15
0
−15
−30
Angular velocity (rad/s)
0.00
0.01
0.02
0.03
0.04
0.05
Time (s)





In [ ]:
```
def C_slidercrank3(q,t):
    '''9 constraint equations for 9 generalized coords
       q=[R1x,R1y,a1,R2x,R2y,a2,R3x,R3y,a3]
       q=[R1,     a1,  R2,  ,a2,  R3,  ,a3]
         [0,1     2    3,4   5    6,7   8 ]

          1/\2
          /  \    slider-crank mechanism
         O   |3|
        ^^^-------------

    Parameters
    ----------
    q : numpy array for 9 generalized coordinates for bodies 1-3 in the slider cran
        q = [q1, q2, q3]
    t : current time

    Returns
    -------
    C : 9 constraint equation evaluations
    '''
    l1 = 0.15
    l2 = 0.25
    q1 = q[0:3]
```

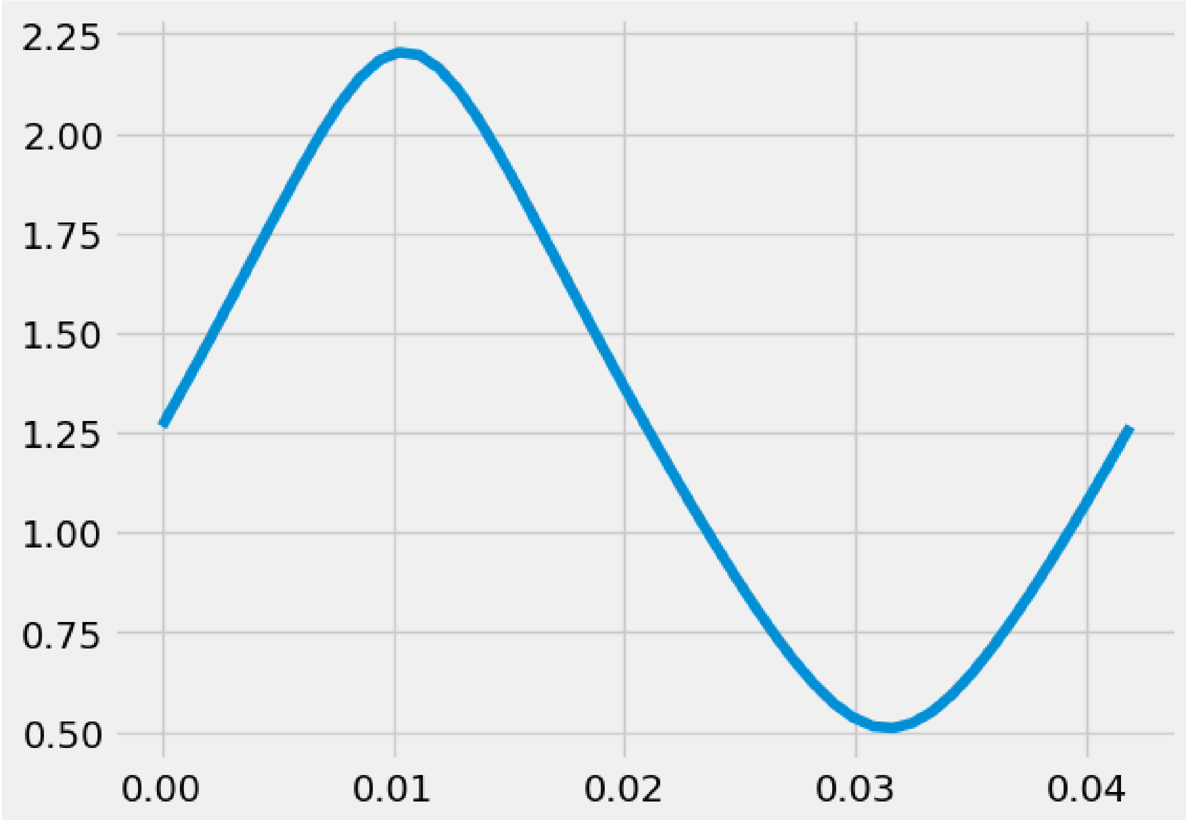
```
    q2 = q[3:6]
    q3 = q[6:9]

    f = 0.25 - 0.8*l1*np.sin(150*t)     # New constraint

    C=np.zeros(9)
    C[0:2] = q1[0:2]+rotA(q1[2])@np.array([-l1/2, 0])
    C[2:4] = q1[0:2]-q2[0:2]+rotA(q1[2])@np.array([l1/2, 0])-rotA(q2[2])@np.array([
    C[4:6] = q2[0:2]-q3[0:2]+rotA(q2[2])@np.array([l2/2, 0])-rotA(q3[2])@np.array([
    C[6] = q3[1]
    C[7] = q3[2]
    C[8] = q3[0] - f    # New constraint in C matrix
    return C

def Cq_slidercrank3(q,t):
    '''return Jacobian of C_slidercrank(q,t) = dC/dq_i
       |dC1/dR1x dC1/dR1y ... dC9/da3 |
       |dC2/dR1x dC2/dR1y ... dC9/da3 |
       |... ..     .           ...    |
       |            .                 |
       |             .                |
       |dC9/dR1x ...         dC9/da3  |
    Parameters
    ----------
    q : numpy array for 9 generalized coordinates for bodies 1-3 in the slider cran
        q = [q1, q2, q3]
    t : current time
    Returns
    -------
    Cq : 9 rows x 9 columns Jacobian of constraints `C_slidercrank`
    '''
    l1 = 0.15
    l2 = 0.25
    q1 = q[0:3]
    q2 = q[3:6]
    q3 = q[6:9]

    Cq=np.zeros((9,9))
    Cq[0:2, 0:3] = Cq_pin(q1, np.array([0, 0, 0]),np.array([-l1/2, 0]),np.array([0,
    Cq[2:4, 0:6] = Cq_pin(q1, q2, np.array([l1/2, 0]), np.array([-l2/2, 0]))
    Cq[4:6, 3:10] = Cq_pin(q2, q3, np.array([l2/2, 0]), np.array([0, 0]))
    Cq[6:8, 7:10] = np.eye(2)
    Cq[8, 6]= 1          # Changed to Rx_3 position
    return Cq

def Qd_slidercrank3(q, dq, t):
    '''return slidercrank Qd = Cq@ddq

    Parameters
    ----------
    q : numpy array for 9 generalized coordinates for bodies 1-3 in the slider cran
        q = [q1, q2, q3]
    t : current time
    Returns
    -------
    Qd : 1D array with length 9
```

```
    '''
    l1 = 0.15
    l2 = 0.25
    q1 = q[0:3]
    q2 = q[3:6]
    q3 = q[6:9]
    dq1 = dq[0:3]
    dq2 = dq[3:6]
    dq3 = dq[6:9]

    Qd=np.zeros(9)
    Qd[0:2] = dq1[2]**2*rotA(q1[2])@np.array([-l1/2, 0])
    Qd[2:4] = dq1[2]**2*rotA(q1[2])@np.array([l1/2, 0]) -\
              dq2[2]**2*rotA(q2[2])@np.array([-l2/2, 0])
    Qd[4:6] = dq2[2]**2*rotA(q2[2])@np.array([l2/2, 0])
    Qd[6:8] = 0
    Qd[8] = 150*150*0.8*l1*sin(150*t)
    return Qd

def Ct_slidercrank3(q, t):
    '''return slidercrank partial derivative of constraints dC/dt

    Parameters
    ----------
    q : numpy array for 9 generalized coordinates for bodies 1-3 in the slider cran
        q = [q1, q2, q3]
    t : current time
    Returns
    -------
    Ct : 1D array with length 9
    '''
    Ct = np.zeros(9)
    Ct[8] = -150*0.8*l1*cos(150*t)
    return Ct
```

In [ ]:
```
t = np.linspace(0, 2*pi/150)
q0 = np.array([0, 0.5, pi/6, 0, 0.5, pi/4, 0.5, 0, 0])
q = np.zeros((len(q0), len(t)))
dq = np.zeros(q.shape)
ddq = np.zeros(q.shape)
q[:, 0] = q0
for i, ti in enumerate(t):
    q[:, i] = fsolve(lambda q: C_slidercrank3(q, ti),q0,\
                    fprime= lambda q: Cq_slidercrank3(q, ti)) # <-- use the Jacobia
    dq[:, i] = np.linalg.solve(Cq_slidercrank3(q[:,i], ti), -Ct_slidercrank3(q[:, i
    Qd = Qd_slidercrank3(q[:, i], dq[:, i], ti)
    ddq[:, i] = np.linalg.solve(Cq_slidercrank3(q[:,i], ti), Qd)
    q0=q[:, i]
```

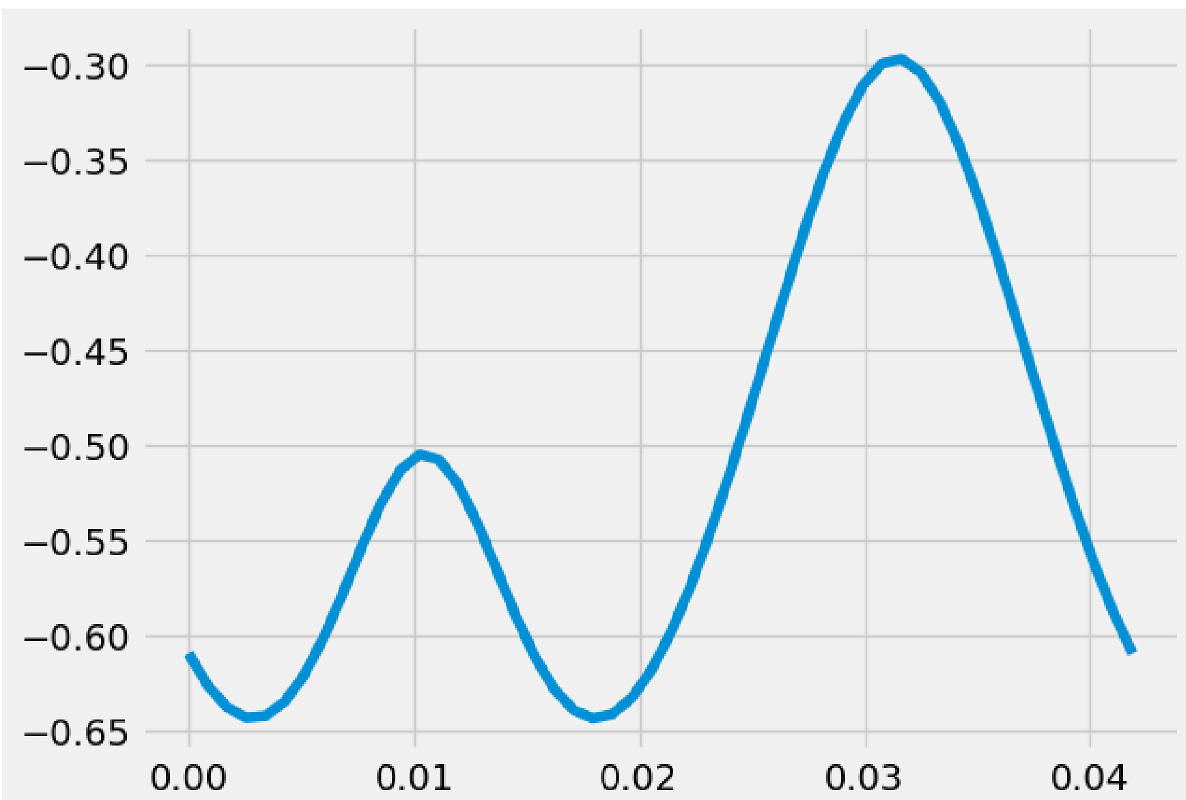
In [ ]:
```
plt.plot(t,q[2,:])
```

Out[ ]: [<matplotlib.lines.Line2D at 0x20126c223a0>]



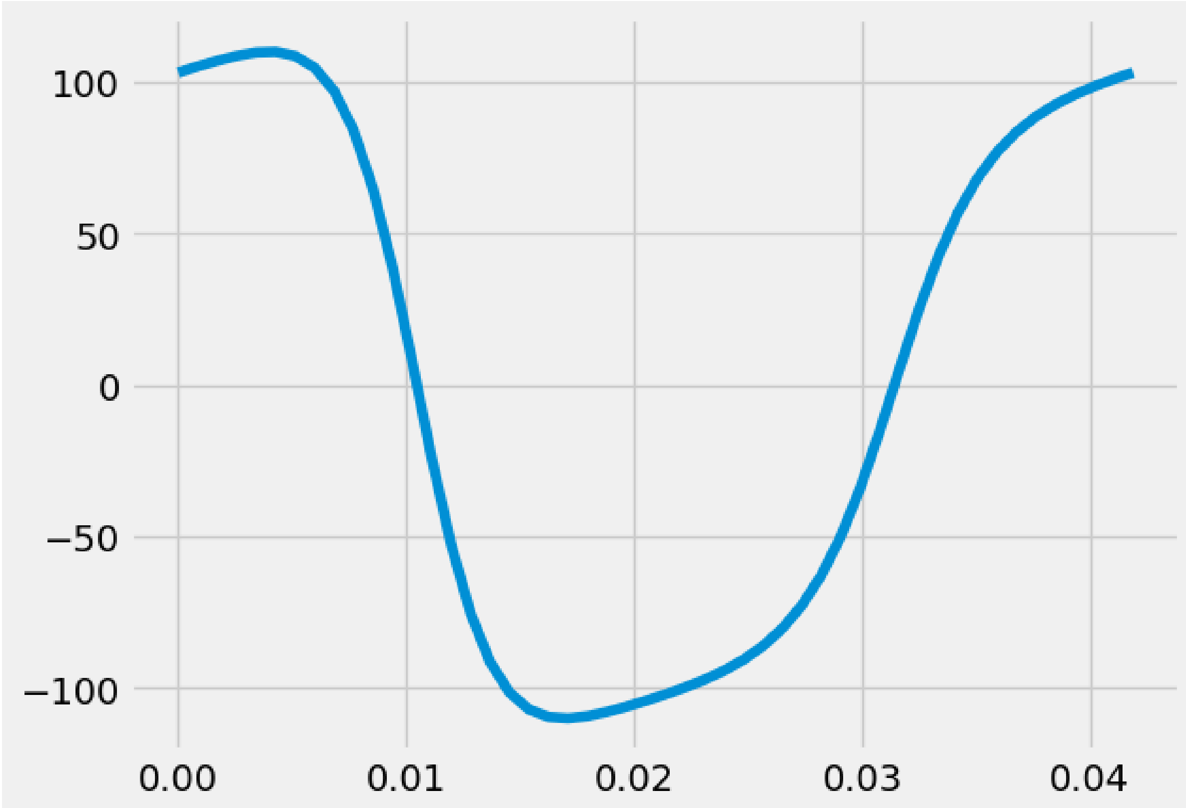
In [ ]:
```
plt.plot(t,q[5,:])
```

Out[ ]: [<matplotlib.lines.Line2D at 0x20126c91c40>]



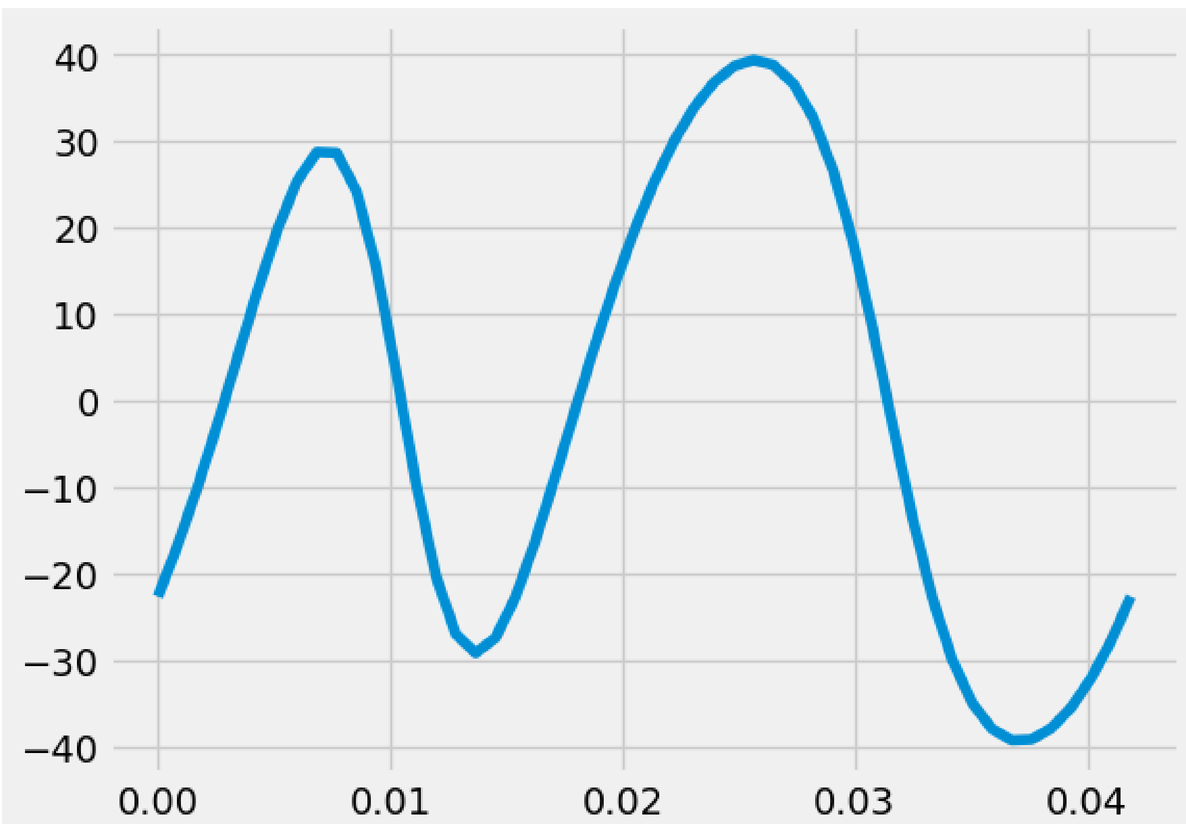
In [ ]:
```
plt.plot(t,-dq[2,:])
```

Out[ ]: [<matplotlib.lines.Line2D at 0x2012732b610>]



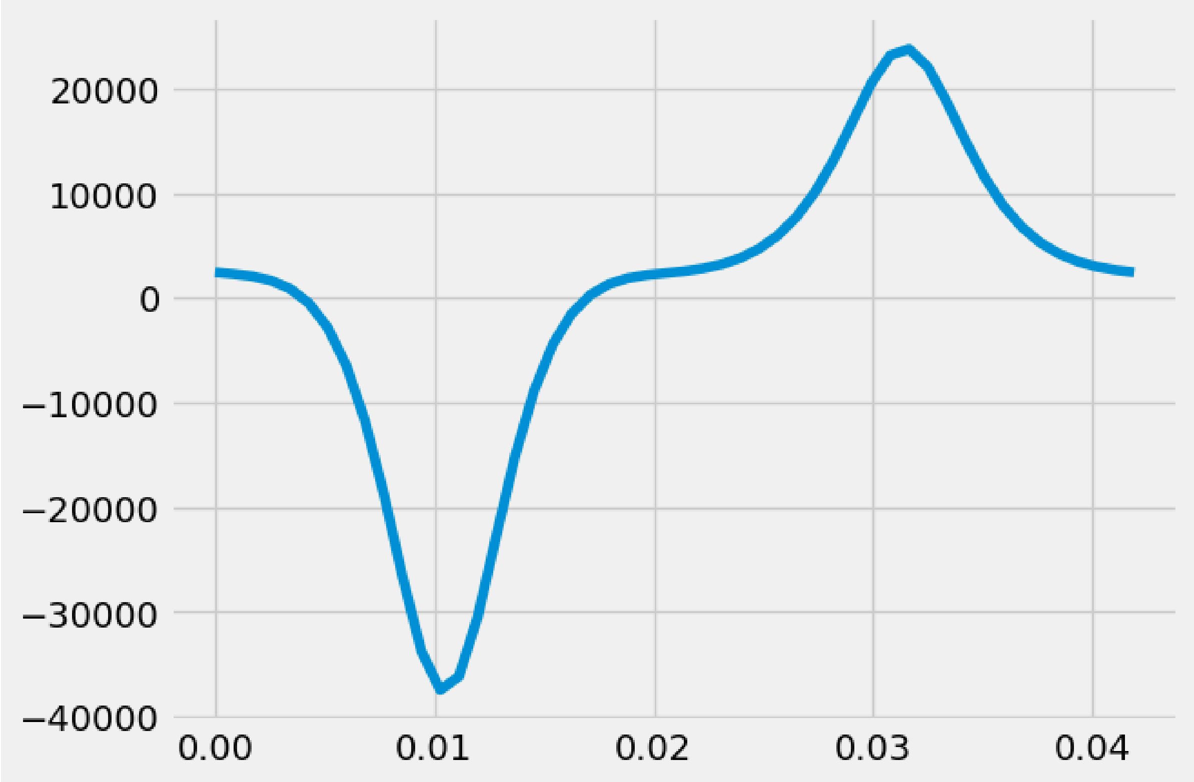
In [ ]:
```
plt.plot(t,-dq[5,:])
```

Out[ ]: [<matplotlib.lines.Line2D at 0x201270404f0>]



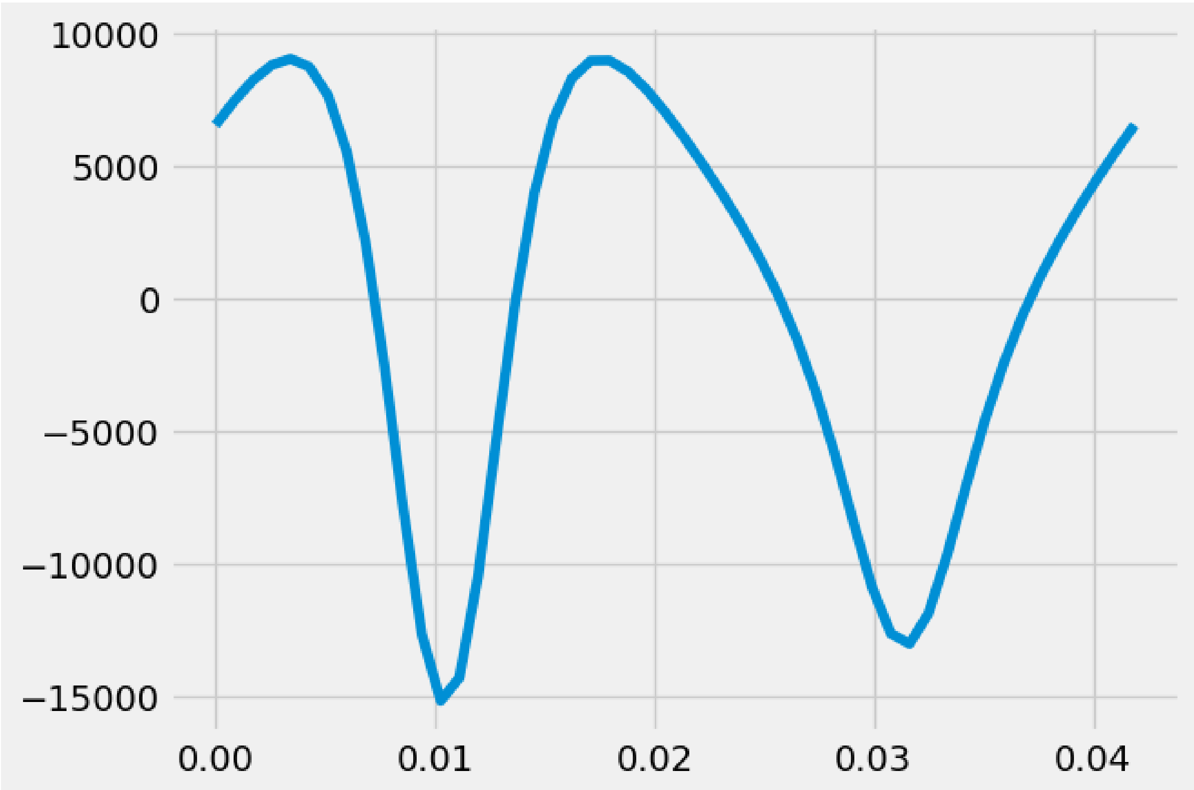
In [ ]:
```
plt.plot(t,ddq[2,:])
```

Out[ ]: [<matplotlib.lines.Line2D at 0x201274691c0>]



In [ ]:
```
plt.plot(t,ddq[5,:])
```

Out[ ]: [<matplotlib.lines.Line2D at 0x201274dd1f0>]



It seems I have a minor error in my solution, though I am unsure where it is. The figures seem to match the expected values and shapes of the reference figures.

```
In [ ]:
```