CS 133a **November 8, 2023**

Problem Set 6

Problem 1 (Scaled Joint Velocities - from Last Year's Quiz 2) - 18 points:

part (a) Let J_1, J_2, J_3, J_4 denote the columns of the Jacobian. Since the first joint is only moving horizontally (and it is a prismatic joint) then we have:

$$J_1 = \begin{bmatrix} \vec{e_x} \\ \vec{0} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Now the rest of the joints are revolute joints. Let $\vec{p_2}$, $\vec{p_3}$, $\vec{p_4}$, $\vec{p_{tip}}$ denote the positions (relative to the world frame) of the 2nd, 3rd, 4th joints, and tip respectively. From the diagram we have that $\vec{p_{tip}} - \vec{p_2} = \begin{bmatrix} 2 & 1 & 0 \end{bmatrix}^T$, $\vec{p_{tip}} - \vec{p_3} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$, and $\vec{p_{tip}} - \vec{p_4} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T$. Note that all revolute joints rotate about the positive z axis $\vec{e_z}$ (relative to the world frame). Therefore, we have:

$$J_2 = \begin{bmatrix} \vec{e_z} \times (\vec{tip} - \vec{p_2}) \\ \vec{e_z} \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$J_3 = \begin{bmatrix} \vec{e_z} \times (\vec{tip} - \vec{p_3}) \\ \vec{e_z} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$J_4 = \begin{bmatrix} \vec{e_z} \times (\vec{tip} - \vec{p_4}) \\ \vec{e_z} \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Therefore we have: $J = \begin{bmatrix} 1 & -1 & 0 & -1 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$. Because we only want $\dot{x}$ and $\dot{y}$ then we have $J = \begin{bmatrix} 1 & -1 & 0 & -1 \\ 0 & 2 & 1 & 0 \end{bmatrix}$

part (a) The cross products for part a were calculated with the code below:

```
import numpy as np

e_z = np.array([0,0,1])
cross2 = np.cross(e_z, np.array([2,1,0])) #For J2
cross3 = np.cross(e_z, np.array([1,0,0])) #For J3
cross4 = np.cross(e_z, np.array([0,1,0])) #For J4

print(cross2)
print(cross3)
print(cross4)
```

part (b)
Let M denote the number of rows J has, and N the number of columns J has. Because $M < N$, then we have redundant system. Generally we are trying to solve:

$$\min_{q,\lambda} \frac{1}{2}\dot{q}^T\dot{q} + \lambda^T(\dot{x}_r - J\dot{q})$$

However, since we are trying to minimize the norm of the relative joint velocities then in this case we have the minimization problem (where W is as defined in the problem):

$$\min_{q,\lambda} \frac{1}{2}(W\dot{q})^T(W\dot{q}) + \lambda^T(\dot{x}_r - J\dot{q})$$

Let $\dot{\overline{q}} = W\dot{q}$. So we have $J\dot{q} = JW^{-1}W\dot{q} = \overline{J}\dot{\overline{q}}$ where $\overline{J} = JW^{-1}$. Substituting we have:

$$\min_{q,\lambda} \frac{1}{2}||\dot{\overline{q}}||^2 + \lambda^T(\dot{x}_r - \overline{J}\dot{\overline{q}})$$

Using the solution from the notes we have:

$$\dot{\overline{q}} = \overline{J}^T\ (\overline{J}\ \overline{J}^T)^{-1}\dot{x}_r \Rightarrow \dot{q} = W^{-1}\ \overline{J}^T\ (\overline{J}\ \overline{J}^T)^{-1}\dot{x}_r$$

By substitution we have (note that $(W^{-1})^T = W^{-1}$ since W^{-1} is diagonal).:

$$\dot{q} = W^{-1}(JW^{-1})^T\ (JW^{-1}\ (JW^{-1})^T)^{-1}\dot{x}_r = W^{-2}\ J^T\ (JW^{-2}\ J^T)^{-1}\dot{x}_r$$

Where $J = \begin{bmatrix} 1 & -1 & 0 & -1 \\ 0 & 2 & 1 & 0 \end{bmatrix}$, $\dot{x}_r = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$, and $W = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{4} \end{bmatrix}$

part (c)

Using the equation and the values for J,W, and $\dot{x}_r$ from part b we have $\dot{q} = \begin{bmatrix} -0.03687636 \\ 0.37310195 \\ 0.2537961 \\ 0.59002169 \end{bmatrix}$.

The code used to calculate is shown below:

```
import numpy as np
J = np.array([[1, 0, -1],
              [0, 1, 0]])
xr_dot = np.array([[-1],
                   [1]])
W = np.diag([1, 1/3, 1/4])
W_inv = np.linalg.inv(W)
J_T = np.transpose(J)
A = J @ W_inv @ W_inv @ J_T
qdot = W_inv @ W_inv @ J_T @ np.linalg.inv(A) @ xr_dot
print(qdot)
```

part (d)

If joint 2 locks up, then the second joint would not contribute to the product $J\dot{q}$. It would also not contribute to the value of $||W\dot{q}||^2$. Therefore the values of J and W can be changed such that $J = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$ and $W = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{4} \end{bmatrix}$. Note that this still yields a redudant system (Jacobian has more columns than rows). Therefore the equation $\dot{q} = W^{-2}\ J^T\ (JW^{-2}\ J^T)^{-1}\dot{x}_r$ from part b still holds, except now $\dot{q}$ will consists of only the velocity of the first, third, and fourth joints. Adjusting the values of J and W as described then we have that $\dot{q} = \begin{bmatrix} \dot{d}_1 \\ \dot{\theta}_3 \\ \dot{\theta}_4 \end{bmatrix} = \begin{bmatrix} -0.05882353 \\ 1 \\ 0.94117647 \end{bmatrix}$.

With $\dot{\theta}_2 = 0$ then we have the complete $\dot{q} = \begin{bmatrix} \dot{d}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \\ \dot{\theta}_4 \end{bmatrix} = \begin{bmatrix} -0.05882353 \\ 0 \\ 1 \\ 0.94117647 \end{bmatrix}$.

The code used to calculate is shown below:

```
import numpy as np
J = np.array([[1, 0, -1],
              [0, 1, 0]])
xr_dot = np.array([[-1],
                   [1]])
W = np.diag([1, 1/3, 1/4])
W_inv = np.linalg.inv(W)
J_T = np.transpose(J)
A = J @ W_inv @ W_inv @ J_T
qdot = W_inv @ W_inv @ J_T @ np.linalg.inv(A) @ xr_dot
qdot = np.array([[qdot[0,0]],
                 [0],
                 [qdot[1,0]],
                 [qdot[2,0]]])
print(qdot)
```

part (e)

If joint 2 and 3 lock up, then the joints would not contribute to the product $J\dot{q}$. They would also not contribute to the value of $||W\dot{q}||^2$. Therefore the values of J and W can be changed such that $J = \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix}$ and $W = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{4} \end{bmatrix}$. The rank r of J in this case is 1, therefore we have the case where $r < min(M, N)$ where M is the number of rows in J and N is the number of columns in J. Let $\dot{\overline{q}} = W\dot{q}$. Note that $J\dot{q} = JW^{-1}W\dot{q} = \overline{J}\,\dot{\overline{q}}$ where $\overline{J} = JW^{-1}$, so we have $\dot{x}_r = \overline{J}\,\dot{\overline{q}}$. Because $r < min(M, N)$ then we used the pseudo inverse to have $\dot{\overline{q}} = (\overline{J})^+\dot{x}_r \Rightarrow \dot{q} = W^{-1}(\overline{J})^+\dot{x}_r$. Using this equation we have $\dot{q} = \begin{bmatrix} \dot{d}_1 \\ \dot{\theta}_4 \end{bmatrix} = \begin{bmatrix} -0.05882353 \\ 0.94117647 \end{bmatrix}$.

With $\dot{\theta}_2 = \dot{\theta}_4 = 0$ then we have the complete $\dot{q} = \begin{bmatrix} \dot{d}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \\ \dot{\theta}_4 \end{bmatrix} = \begin{bmatrix} -0.05882353 \\ 0 \\ 0 \\ 0.94117647 \end{bmatrix}$.

The code used to calculate is shown below:

```
import numpy as np
J = np.array([[1, -1],
              [0, 0]])
xr_dot = np.array([[-1],
                   [1]])
W = np.diag([1, 1/4])
W_inv = np.linalg.inv(W)
J_bar = J @ W_inv
qdot = W_inv @ np.linalg.pinv(J_bar) @ xr_dot

qdot = np.array([[qdot[0,0]],
                 [0],
                 [0],
                 [qdot[1,0]]])
print(qdot)
```

Problem 2 (Jacobian Inversion near Singularities) - 18 points:

part (a)

$$J = \begin{bmatrix} -0.01234118 & 0 & 0 \\ 0 & -1.41415971 & -0.71325045 \\ 0 & 0.01234118 & -0.70090926 \end{bmatrix}$$

$$U = \begin{bmatrix} 0 & 0 & 1 \\ -0.97521688 & -0.22125108 & 0 \\ -0.22125108 & 0.97521688 & 0 \end{bmatrix}$$

$$S = \begin{bmatrix} 1.61803399 & 0 & 0 \\ 0 & 0.61803399 & 0 \\ 0 & 0 & 0.01234118 \end{bmatrix}$$

$$V^T = \begin{bmatrix} 0 & 0.85065081 & 0.52573111 \\ 0 & 0.52573111 & -0.85065081 \\ -1 & 0 & 0 \end{bmatrix}$$

Code used to calculate matrices:

```
import numpy as np
def Jac(q):
    theta_pan, theta_1, theta_2 = q[0,0], q[1,0],  q[2,0]
    sum_cos = np.cos(theta_1) + np.cos(theta_1 + theta_2)
    sum_sin = np.sin(theta_1) + np.sin(theta_1 + theta_2)
    J = np.eye(3)
    # first row
    theta_12 = theta_1 + theta_2
    J[0] = np.array([-np.cos(theta_pan) * sum_cos,
                     np.sin(theta_pan) * sum_sin,
                     np.sin(theta_pan) * np.sin(theta_12)])
    # second row
    J[1] = np.array([-np.sin(theta_pan) * sum_cos,
                     -np.cos(theta_pan) * sum_sin,
                     -np.cos(theta_pan) * np.sin(theta_12)])
    # third row
    J[2] = np.array([0,
                     np.cos(theta_1) + np.cos(theta_12),
                     np.cos(theta_12)])
    # Return the Jacobian as a numpy 3x3 matrix.
    return J

q = np.array([[np.radians(0)],
              [np.radians(44.5)],
              [np.radians(90)]])
J = Jac(q)
print("J: {}".format(J))
u, s, vT = np.linalg.svd(J)
print("U: {}".format(u))
print("S: {}".format(np.diag(s)))
print("V^T: {}".format(vT))
```

part (b)

$$\dot{q} = J^{-1}\dot{x}_r = \begin{bmatrix} -81.02949691 \\ 0.71325045 \\ -1.41415971 \end{bmatrix}$$

This gives us

$$\dot{x} = J(q)\dot{q} = \begin{bmatrix} 1.0 \\ -9.6606 \cdot 10^{-17} \\ 1.0 \end{bmatrix}$$

Nothing bad seems to be occurring.

Code used to calculate the matrices (refer to part A for the definition of the function Jac())

```
import numpy as np
q = np.array([[np.radians(0)],
              [np.radians(44.5)],
              [np.radians(90)]])
xr_dot = np.array([[1],
                   [0],
                   [1]])
J = Jac(q)
qdot = np.linalg.inv(J) @ xr_dot
print("qdot : {}".format(qdot))
xdot = J @ qdot
print("xdot : {}".format(xdot))
```

part (c)

$$\dot{q} = J^{-1}\dot{x}_r = \begin{bmatrix} -48.91378492 \\ 0.71303776 \\ -1.41380565 \end{bmatrix}$$

This gives us

$$\dot{x} = J(q)\dot{q} = \begin{bmatrix} 6.03654062 \cdot 10^{-1} \\ 4.82326738 \cdot 10^{-5} \\ 9.99749208 \cdot 10^{-1} \end{bmatrix}$$

Compared to part b, the only change to $\dot{q}$ was the velocity of the pan angle $\dot{\theta}_{pan}$, it changes by about 31 rad/s. The achieved $\dot{x}$ seems to have changed more. The velocity in the x direction decreased from 1.0 m/s to about 0.6 m/s. The velocity in the y direction increased from about 0 to about $4.85 \cdot 10^{-5}$ m/s. Lastly, the velocity in the z direction barely decreased.

Code used to calculate the matrices (refer to part A for the definition of the function Jac())

```
import numpy as np
def problem2cd(gamma):
    q = np.array([[np.radians(0)],
                  [np.radians(44.5)],
                  [np.radians(90)]])
    xr_dot = np.array([[1],
                  [0],
                  [1]])
    J = Jac(q)
    JT = np.transpose(J)
    A = J @ JT + gamma**2 * np.eye(3)
    JW_inv = JT @ np.linalg.inv(A)
    qdot = JW_inv @ xr_dot
    print("qdot : {}".format(qdot))
    xdot = J @ qdot
    print("xdot : {}".format(xdot))

problem2cd(0.01)
```

part (d)

$$\dot{q} = J^{-1}\dot{x}_r = \begin{bmatrix} -1.21560424 \\ 0.69252875 \\ -1.37964159 \end{bmatrix}$$

This gives us

$$\dot{x} = J(q)\dot{q} = \begin{bmatrix} 0.015002 \\ 0.00468373 \\ 0.9755502 \end{bmatrix}$$

Increasing γ made $\dot{\theta}_{pan}$ in $\dot{q}$ increase significantly (from -48.9 m/s to -1.21 m/s). On the other hand, $\dot{\theta}_1$ and $\dot{\theta}_2$ did not change by much. Increasing γ made the x velocity decrease (from 0.6 m/s to 0.15 m/s), the y velocity increase (from $4.85 \cdot 10^{-5}$ to 0.0047 m/s). The z velocity did not decrease by much (by only about 0.12 m/s). Code used for calculations:

```
problem2cd(0.1)
```

Refer to part c for the definition of problem2cd().

part (e)

$$\dot{q} = J^{-1}\dot{x}_r = \begin{bmatrix} -1.23411849 \\ 0.71325045 \\ -1.41415971 \end{bmatrix}$$

This gives us

$$\dot{x} = J(q)\dot{q} = \begin{bmatrix} 0.0152304844 \\ 2.83812319 \cdot 10^{-16} \\ 1.0 \end{bmatrix}$$

Compared to above (part d), this achieved $\dot{x}$ is closer to $\dot{x}_r$ than the achieved tip velocity in part d. This $\dot{x}$ has matches $\dot{x}_r$ in terms of the z velocity. The y velocity is extremely close ($2.838 \cdot 10{-}^{16}$ is close to zero). The made difference between this $\dot{x}$ and $\dot{x}_r$ is in the x velocity. Where the desired x velocity is 1.0 m/s but the achieved is 0.01523 m/s.

Code used for calculations (refer to part A for the definition of Jac()):

```
import numpy as np
def problem2e(gamma):
    q = np.array([[np.radians(0)],
                  [np.radians(44.5)],
                  [np.radians(90)]])
    xr_dot = np.array([[1],
                  [0],
                  [1]])
    J = Jac(q)

    u, s, vT = np.linalg.svd(J)
    uT = np.transpose(u)
    v = np.transpose(vT)
    diagonals = []
    for si in s:
        if np.abs(si) >= gamma:
            diagonals.append(1/si)
        else:
            diagonals.append(si/(gamma**2))
    diagonals = np.array(diagonals)
    S = np.diag(diagonals)
    qdot = v @ S @ uT @ xr_dot
    print("qdot : {}".format(qdot))
    xdot = J @ qdot
    print("xdot : {}".format(xdot))

problem2e(0.1)
```

Problem 3 (Gimbal Rotational Motion) - 20 points:

part (a)
Going from R_0 to R_A we have $\alpha_0 = 0°$ and $\alpha_f = -90°$. Let a,b,c,d denote the coefficients for the cubic spline $a + bt + ct^2 + dt^3$. So we have $a = \alpha_0 = 0$, and $b = \dot{\alpha}_f = 0$. Let $T = 2s$ denote the time of the motion, so we have $c = 3 * (\alpha_f - \alpha_0)/T^2 - \dot{\alpha_f}/T - 2\dot{\alpha}_0/T = 3\frac{-\pi/2}{4} = -\frac{3\pi}{8}$, since the initial and final velocities must also be zero. Lastly, we have $d = -2 * (\alpha_f - \alpha_0)/T^3 + \dot{\alpha_f}/T^2 + \dot{\alpha}_0/T^2 = \frac{2\pi}{16}$.
Therefore using a cubic spline for this motion to be performed withing 2 seconds we have (note that the angle and velocity are given in radian and radians/s respectively using the equation below):

$$\alpha(t) = -\frac{3\pi}{8}t^2 + \frac{2\pi}{16}t^3$$

$$\dot{\alpha}(t) = -\frac{6\pi}{8}t + \frac{6\pi}{16}t^2$$

Now for the desired rotation matrix we have:

$$R_d(t) = R_d(\alpha) \begin{bmatrix} cos(\alpha) & 0 & sin(\alpha) \\ 0 & 1 & 0 \\ -sin(\alpha) & 0 & cos(\alpha) \end{bmatrix}$$

And we also have (since we are rotating about the y axis)

$$\omega_d(t) = \omega(\alpha, \dot{\alpha}) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \cdot \dot{\alpha} = \begin{bmatrix} 0 \\ \dot{\alpha} \\ 0 \end{bmatrix}$$

part (b)
Let $\vec{e}_{tip}$ denote the axis of rotation for the second phase relative to the tip frame. Let $\vec{e}_w$ denote the axis of rotation for the second phase relative to world frame. So we have

$$\vec{e}_{tip} = \frac{1}{\sqrt{0.05^2 + 0.05^2}} \begin{bmatrix} 0 \\ 0.05 \\ -0.05 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.70710678 \\ -0.70710678 \end{bmatrix}$$

$$\vec{e}_w = \frac{1}{\sqrt{0.05^2 + 0.05^2}} \begin{bmatrix} 0.05 \\ 0.05 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.70710678 \\ 0.70710678 \\ 0 \end{bmatrix}$$

For the angular velocity we have

$$w_d(t) = Rot_y\left(\frac{-\pi}{2}\right)\vec{e}_{tip} \cdot \dot{\beta}(t) = \vec{e}_w \cdot \dot{\beta}(t) = \begin{bmatrix} 0.70710678 \cdot \dot{\beta}(t) \\ 0.70710678 \cdot \dot{\beta}(t) \\ 0 \end{bmatrix}$$

Now let $Rot_{\vec{e}_{tip}}(\beta(t))$ describe a rotation about the axis $\vec{e}_{tip}$ of $\beta(t)$ radians. Note that this is the rotation in the second phase relative to the the tip frame. Therefore the full rotation relative to the world frame is:

$$R_d(t) = Rot_y\left(\frac{-\pi}{2}\right) Rot_{\vec{e}_{tip}}(\beta(t))$$

where Rot_y is a rotation about the standard y axis $\vec{e}_y$. $Rot_{\vec{e}_{tip}}(\beta(t))$ is defined on the next page.

part (b)

$$ex = \begin{bmatrix} 0 & 0.70710678 & 0.70710678 \\ -0.70710678 & 0 & 0 \\ -0.70710678 & 0 & 0 \end{bmatrix}$$

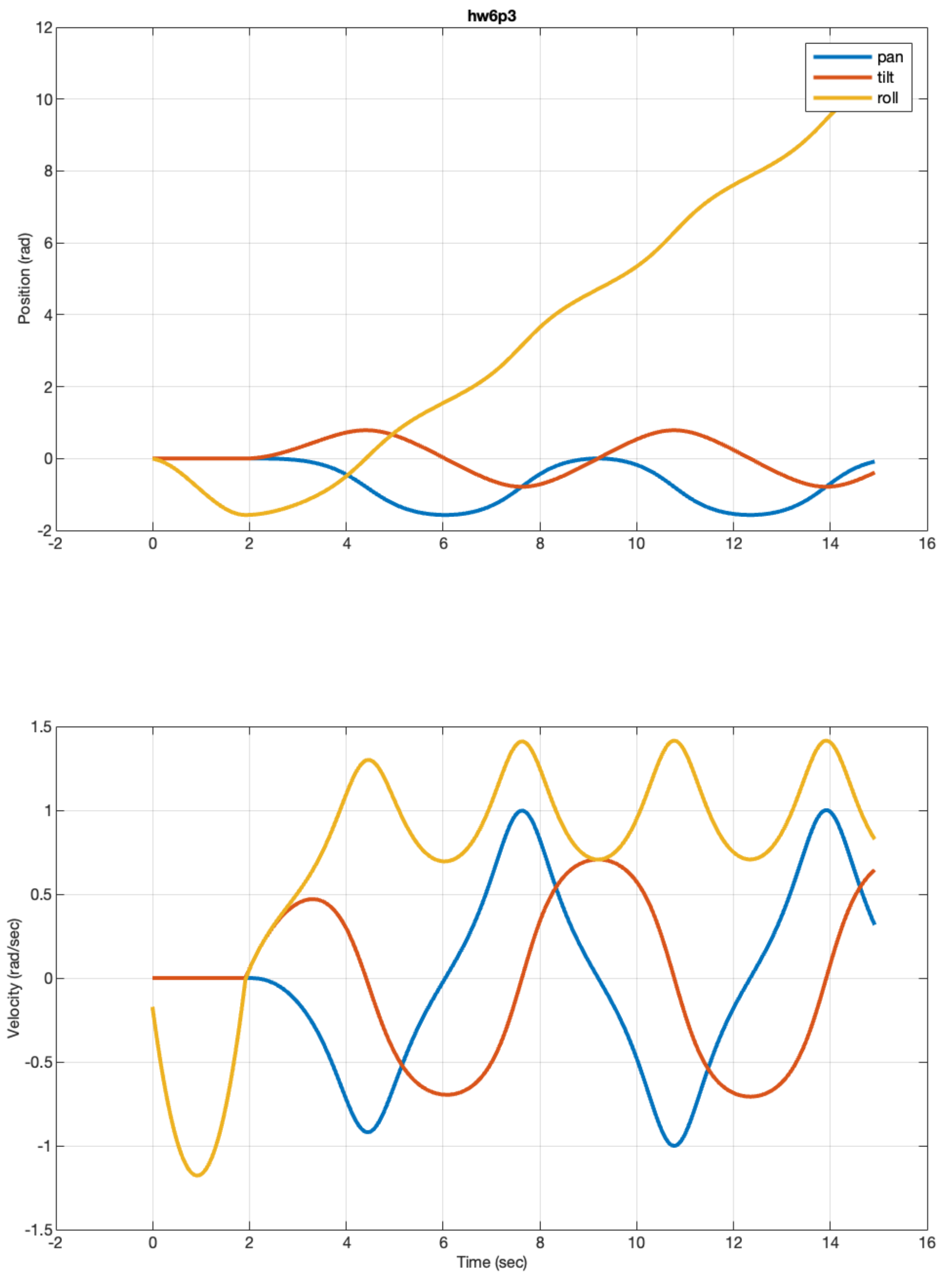
$$Rot_{\vec{e}_{tip}}(\beta(t)) = I_n + sin(\beta(t))ex + (1.0 - cos(\beta(t)))(ex)(ex)$$

part (a)-(b) code

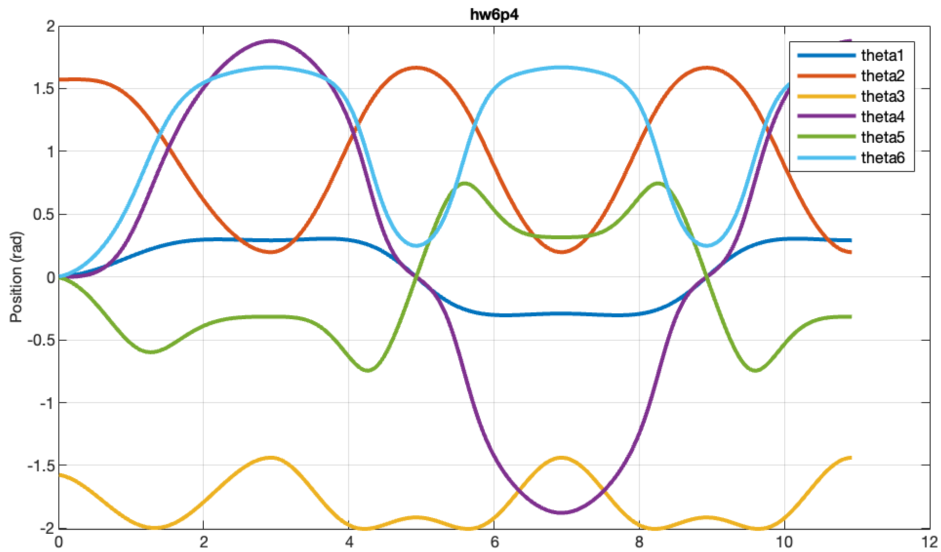
```
class Trajectory():
    # Initialization.
    def __init__(self, node):
        # Set up the kinematic chain object.
        self.chain = KinematicChain(node, 'world', 'tip', self.jointnames())

        # Initialize the current joint position to the starting
        # position and set the desired orientation to match.
        self.qlast = np.zeros((3,1))
        (_, self.Rd_last, _, _) = self.chain.fkin(self.qlast)

        # Pick the convergence bandwidth.
        self.lam = 20

        # rotation axis for t>2
        # relative to tip frame, and world frame
        self.e_tip = exyz(0, 0.05, -0.05)
        self.e_w =  exyz(0.05, 0.05, 0)

    # Declare the joint names.
    def jointnames(self):
        # Return a list of joint names FOR THE EXPECTED URDF!
        return ['pan', 'tilt', 'roll']

    # Evaluate at the given time.  This was last called (dt) ago.
    def evaluate(self, t, dt):
        # Choose the alpha/beta angles based on the phase.
        if t <= 2.0:
            # Part A (t<=2):
            (alpha, alphadot) = goto(t, 2, 0, -np.pi/2)
            (beta,  betadot)  = (0.0, 0.0)

            Rd = Roty(alpha)
            wd = ey() * alphadot
        else:
            # Part B (t>2):
            (alpha, alphadot) = (-np.pi/2, 0)
            beta = t - 3 + math.e**(2-t)
            betadot = 1 - math.e**(2-t)

            # Compute the desired rotation and angular velocity.
            Rd = Roty(alpha) @ Rote(self.e_tip, beta)
            wd = self.e_w * betadot
```

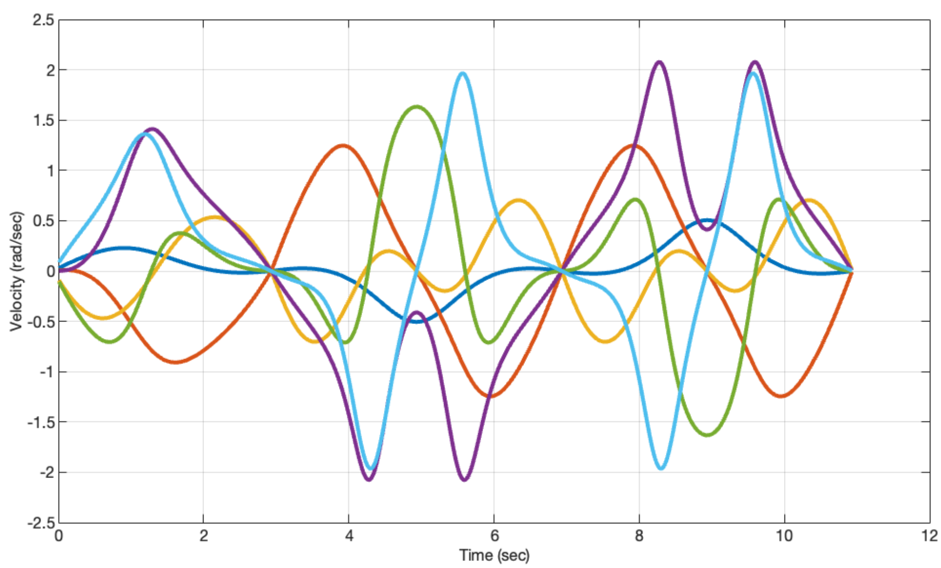
```
# Compute the old forward kinematics.
(_, R, _, Jw) = self.chain.fkin(self.qlast)

# Compute the inverse kinematics
error = eR(self.Rd_last, R)
A = wd + self.lam * error
qdot = np.linalg.pinv(Jw) @ A

# Integrate the joint position.
q = self.qlast + dt * qdot

# Save the data needed next cycle.
self.qlast = q
self.Rd_last = Rd

# Return the position and velocity as python lists.
return (q.flatten().tolist(), qdot.flatten().tolist())
```

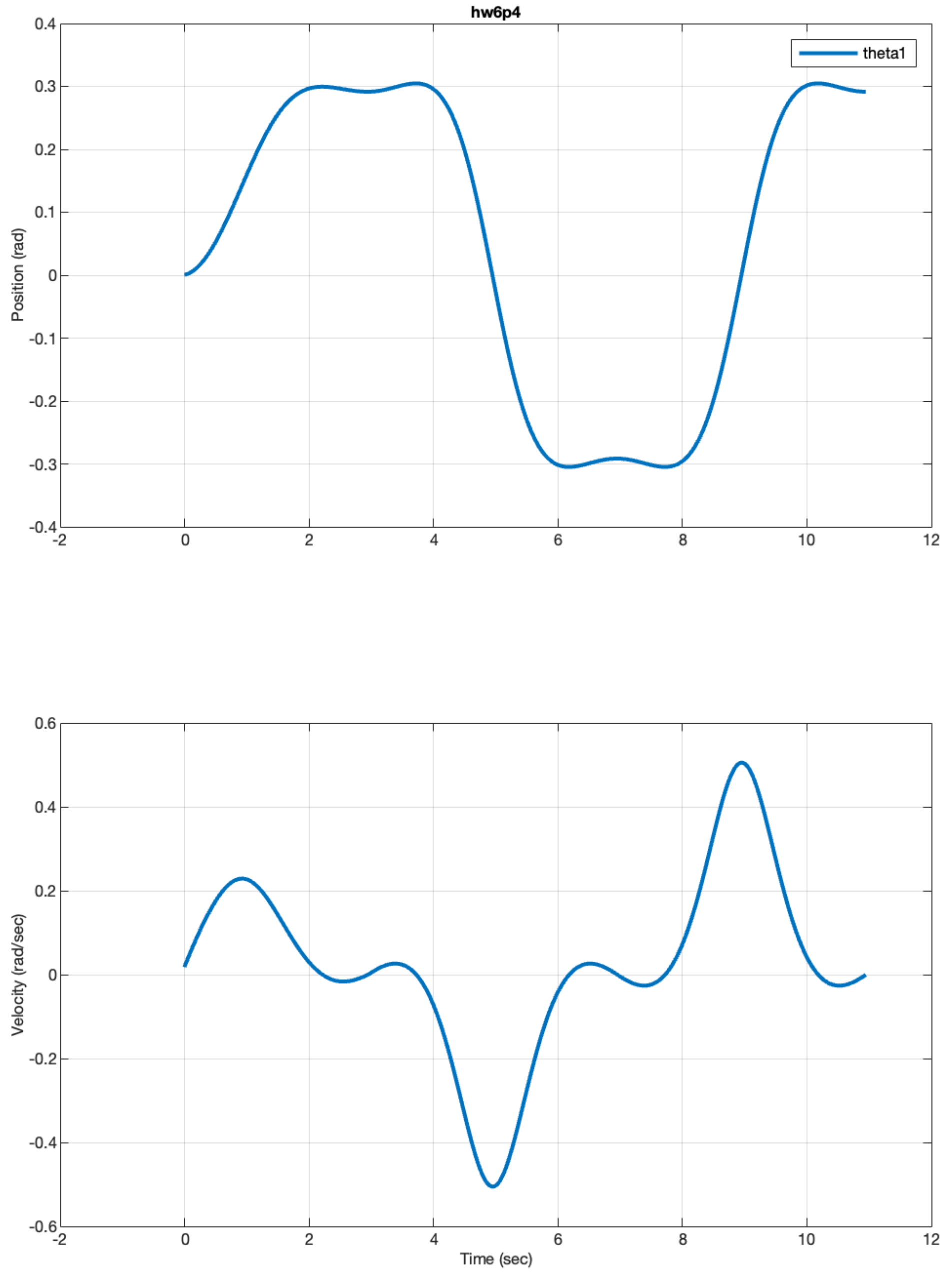
part (a)-(b) plots

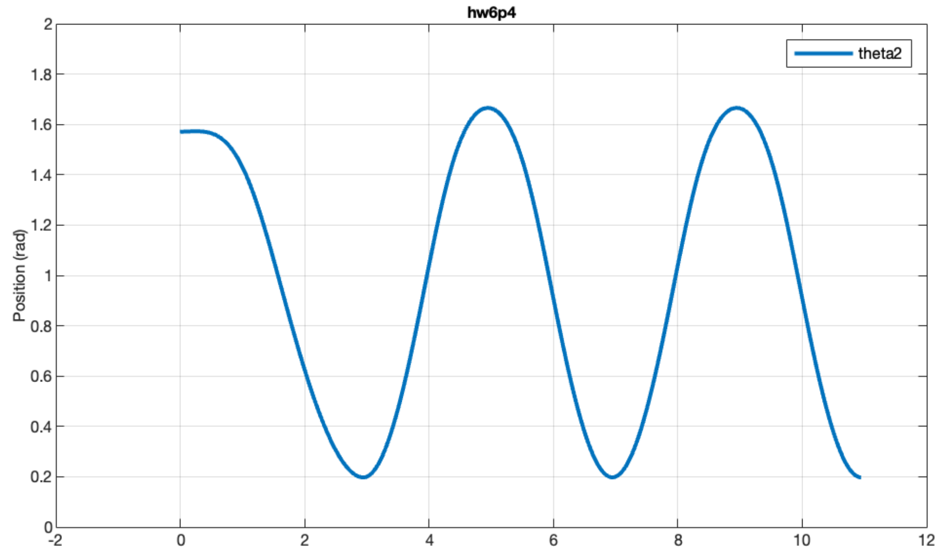


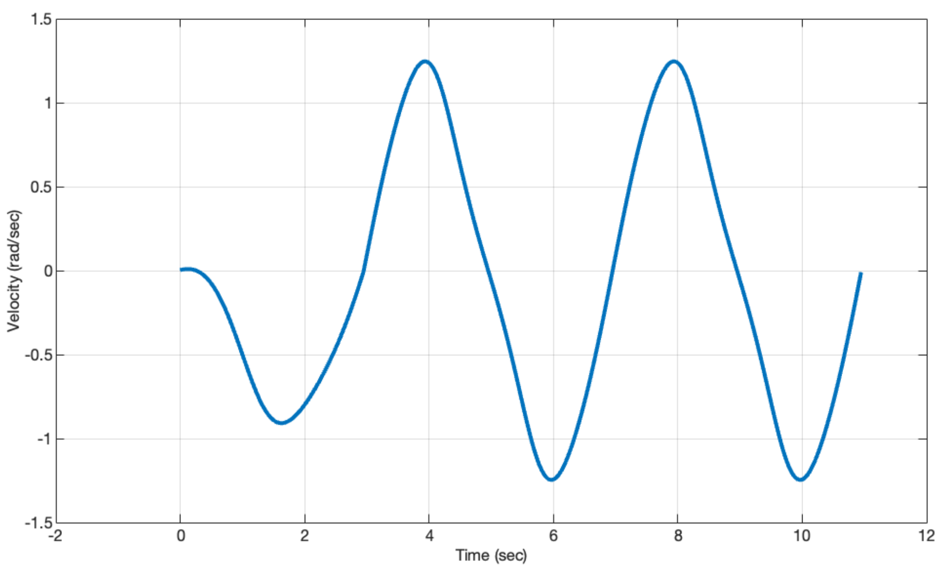
Problem 4 (6 DOF Inverse Kinematics - Up-Down Movements) - 20 points:

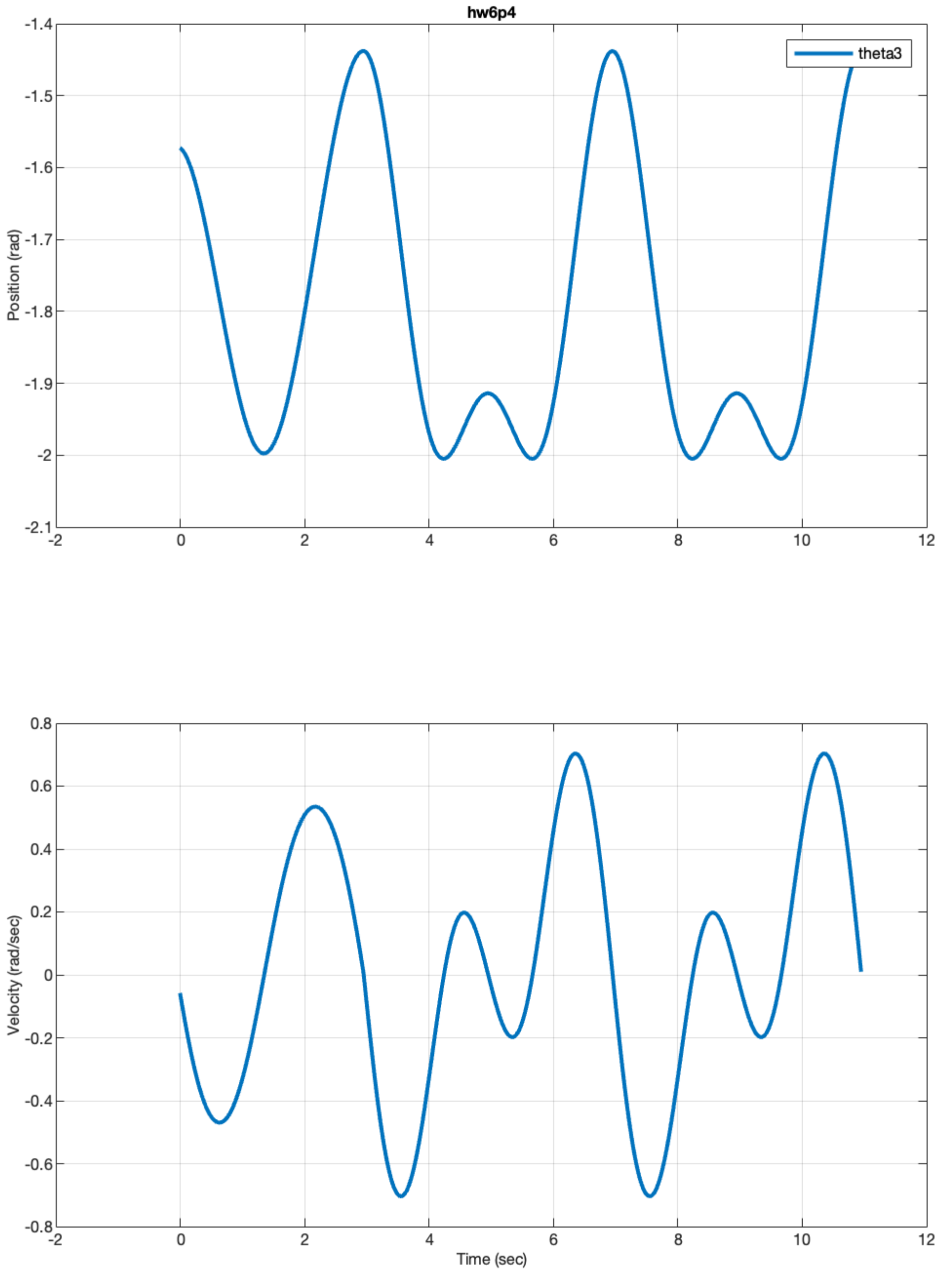


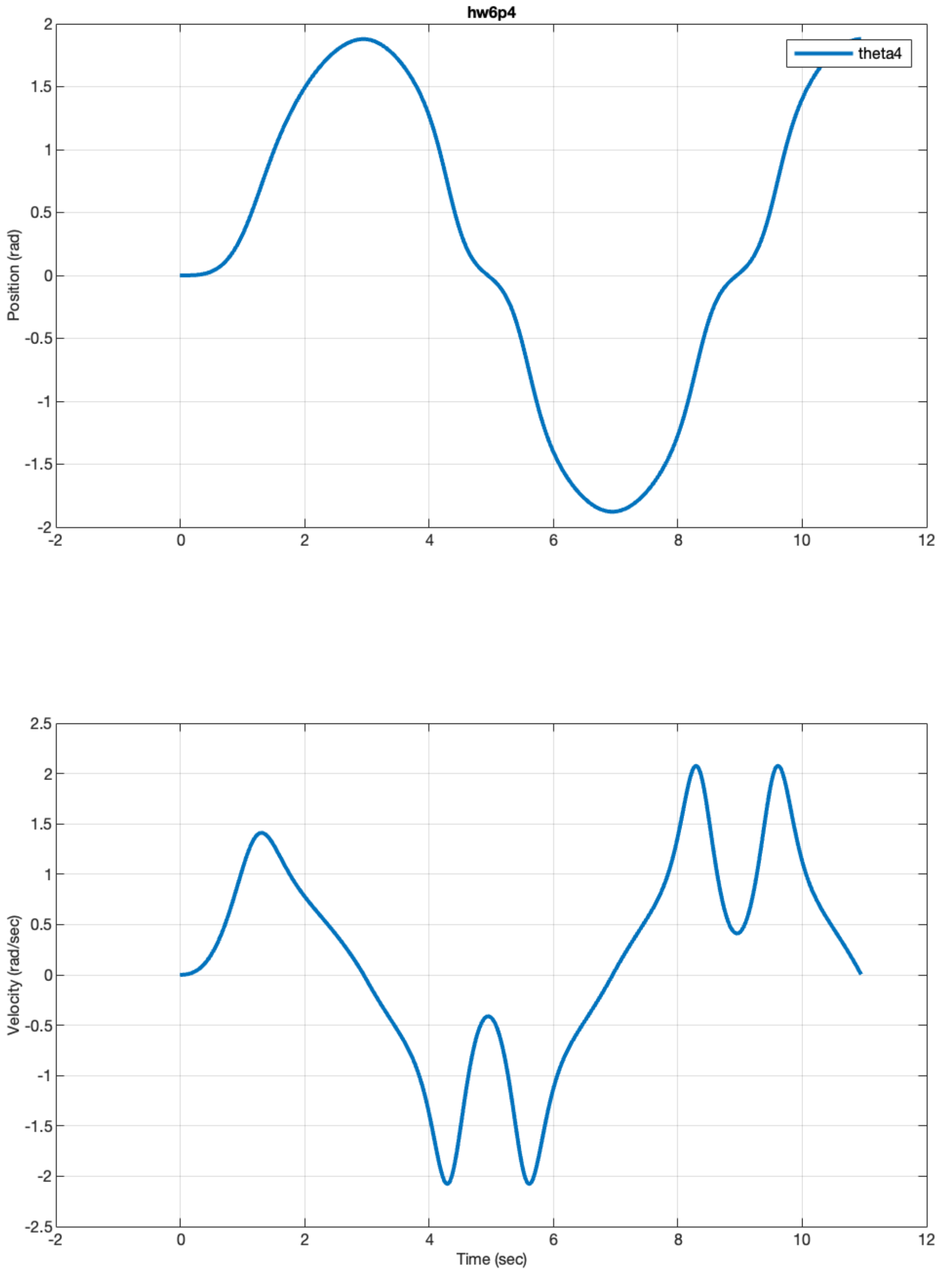


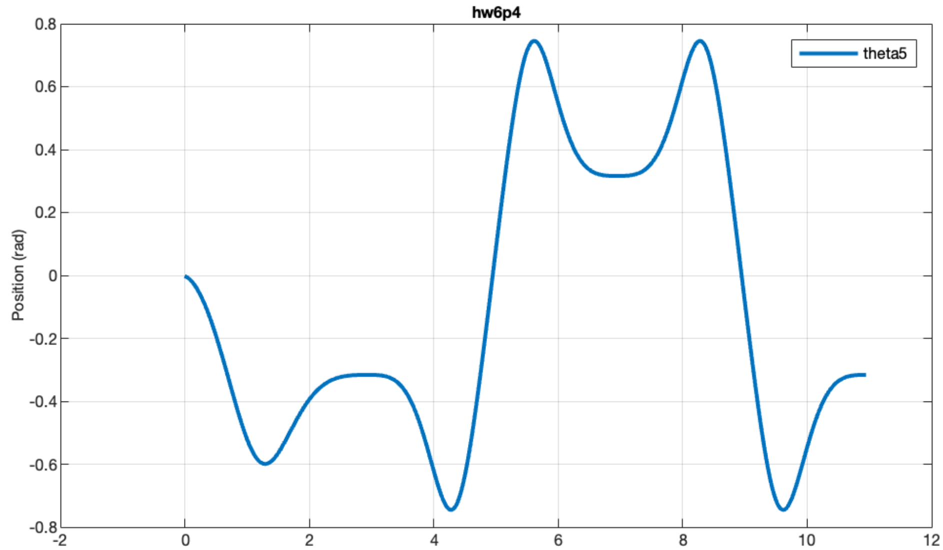

hw6p4
theta1
Position (rad)
Velocity (rad/sec)
Time (sec)

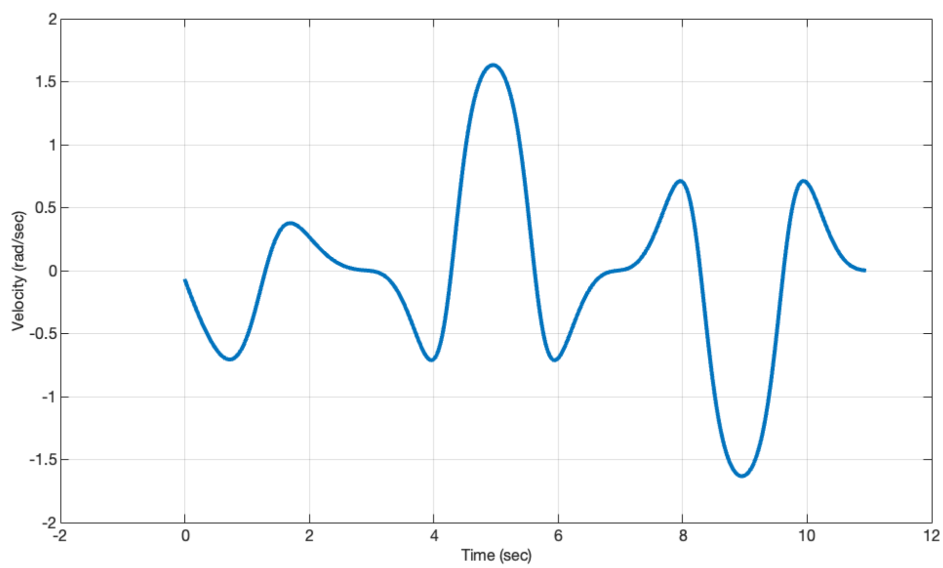

hw6p4
theta2
Position (rad)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-2
0
2
4
6
8
10
12

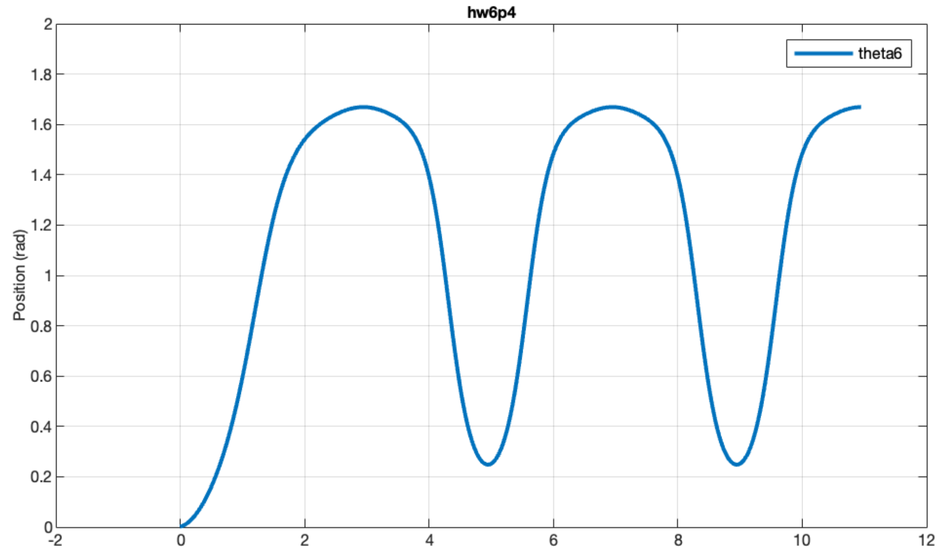

Velocity (rad/sec)
1.5
1
0.5
0
-0.5
-1
-1.5
-2
0
2
4
6
8
10
12
Time (sec)

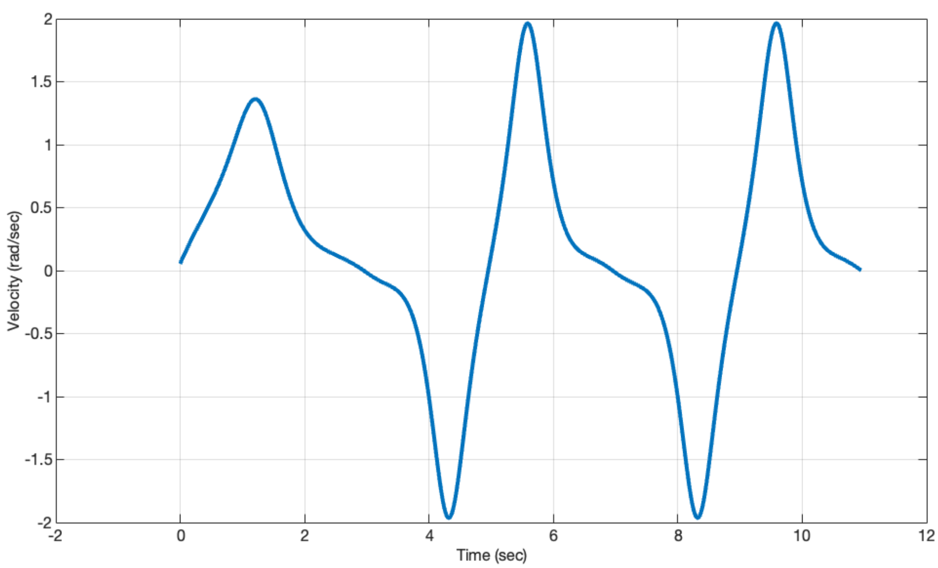

hw6p4
theta3
Position (rad)
Velocity (rad/sec)
Time (sec)


hw6p4
theta4
Position (rad)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
0
2
4
6
8
10
12
Velocity (rad/sec)
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-2
0
2
4
6
8
10
12
Time (sec)


hw6p4
theta5
Position (rad)
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-2
0
2
4
6
8
10
12


Velocity (rad/sec)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
0
2
4
6
8
10
12
Time (sec)


hw6p4
theta6
Position (rad)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-2
0
2
4
6
8
10
12


Velocity (rad/sec)
Time (sec)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
0
2
4
6
8
10
12

Code:

```
class Trajectory():
    # Initialization.
    def __init__(self, node):
        # Set up the kinematic chain object.
        self.chain = KinematicChain(node, 'world', 'tip', self.jointnames())

        # Define the various points.
        self.q0 = np.radians(np.array([0, 90, -90, 0, 0, 0]).reshape((-1,1)))
        self.p0 = np.array([0.0, 0.55, 1.0]).reshape((-1,1))
        self.R0 = Reye()

        self.plow  = np.array([0.0, 0.5, 0.3]).reshape((-1,1))
        self.phigh = np.array([0.0, 0.5, 0.9]).reshape((-1,1))

        # Initialize the current/starting joint position.
        self.qlast  = self.q0
        self.xd_last = self.p0
        self.Rd_last = self.R0
        self.lam = 20

    # Declare the joint names.
    def jointnames(self):
        # Return a list of joint names FOR THE EXPECTED URDF!
        return ['theta1', 'theta2', 'theta3', 'theta4', 'theta5', 'theta6']

    # Evaluate at the given time.  This was last called (dt) ago.
    def evaluate(self, t, dt):
        # Decide which phase we are in:
        if t < 3.0:
            # Approach movement:
            (s0, s0dot) = goto(t, 3.0, 0.0, 1.0)

            pd = self.p0 + (self.plow - self.p0) * s0
            vd =           (self.plow - self.p0) * s0dot

            Rd = Rotz(-pi/2 * s0)
            wd = ez() * (-pi/2 * s0dot)

        else:
            # Pre-compute the path variables.  To show different
            # options, we compute the position path variable using
            # sinusoids and the orientation variable via splines.
            sp    =      - cos(pi/2 * (t-3.0))
            spdot = pi/2 * sin(pi/2 * (t-3.0))

            t1 = (t-3) % 8.0
            if t1 < 4.0:
                (sR, sRdot) = goto(t1,     4.0, -1.0,  1.0)
            else:
                (sR, sRdot) = goto(t1-4.0, 4.0,  1.0, -1.0)
```

```
        # Use the path variables to compute the trajectory.
        pd = 0.5*(self.phigh+self.plow) + 0.5*(self.phigh-self.plow) * sp
        vd =                              + 0.5*(self.phigh-self.plow) * spdot

        Rd = Rotz(pi/2 * sR)
        wd = ez() * (pi/2 * sRdot)


    # Compute the old forward kinematics.
    (ptip, R, Jv, Jw) = self.chain.fkin(self.qlast)

    # Compute the errors
    error_pos = ep(self.xd_last, ptip)
    error_rot = eR(self.Rd_last, R)
    error = np.vstack((error_pos, error_rot))

    # compute qdot
    v = np.vstack((vd,wd))
    A = v + self.lam * error
    J = np.vstack((Jv, Jw))
    qdot = np.linalg.pinv(J) @ A

    # Integrate the joint position.
    q = self.qlast + dt * qdot

    # Save the data needed next cycle.
    self.qlast = q
    self.xd_last = pd
    self.Rd_last = Rd

    # Return the position and velocity as python lists.
    return (q.flatten().tolist(), qdot.flatten().tolist())
```

Problem 5 (Touch and Go Movements) - 20 points:

part (a) Rotation relative to world for both targets:

$$R_{right} = I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_{left} = Rot_y\left(\frac{-\pi}{2}\right) Rot_z\left(\frac{\pi}{2}\right) = \begin{bmatrix} 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

part (b)

Let $p_{high} = \begin{bmatrix} 0 \\ 0.5 \\ 0.9 \end{bmatrix}$

The motion starts at a known $(p_0, R_0) = fkin(q_0)$, then cycles through (p_{right}, R_{right}), (p_{high}, I), (p_{left}, R_{left}), (p_{high}, I), and ever repeating.

For $t < 3$ (from p_0 to p_{right}):

$$(s_0, \dot{s}_0) = goto5(t, 3.0, 0.0, 1.0) \Rightarrow \qquad s_0(t) = \frac{10}{3^2}t^3 - \frac{15}{3^4}t^4 + \frac{6}{3^5}t^5$$

$$p_d(t) = p_0 + (p_{right} - p_0)s_0 \qquad R_d(t) = I$$

$$v_d(t) = (p_{right} - p_0)\dot{s}_0 \qquad w_d(t) = \vec{0}$$

For $t \geq 3$ with $t_1 = (t-3)\ \%\ 5$:

For $0 \leq t_1 < 1.25$ (from p_{right} to p_{high}):

$$(s_{p1}, \dot{s}_{p1}) = goto5(t_1, 1.25, -1.0, 1.0) \Rightarrow \qquad s_{p1}(t_1) = -1 + \frac{20}{1.25^2}t_1^3 - \frac{30}{1.25^4}t_1^4 + \frac{12}{1.25^5}t_1^5$$

$$(s_{R1}, \dot{s}_{R1}) = goto5(t_1, 1.25, 0, 1.0) \Rightarrow \qquad s_{R1}(t_1) = \frac{10}{1.25^2}t_1^3 - \frac{15}{1.25^4}t_1^4 + \frac{6}{1.25^5}t_1^5$$

$$p_d(t) = \frac{1}{2}(p_{high} + p_{right}) + \frac{1}{2}(p_{high} - p_{right})s_{p1} \qquad R_d(t) = Rot_y\left(\frac{-\pi}{2}s_{R1}\right)$$

$$v_d(t) = \frac{1}{2}(p_{high} - p_{right})\dot{s}_{p1} \qquad w_d(t) = e_y\left(\frac{-\pi}{2}\dot{s}_{R1}\right)$$

For $1.25 \leq t_1 < 2.50$ (from p_{high} to p_{left}):

$$(s_{p2}, \dot{s}_{p2}) = goto5(t_1 - 1.25, 1.25, -1.0, 1.0) \Rightarrow \qquad s_{p2}(t_1) = -1 + \frac{20}{1.25^2}(t_1 - 1.25)^3 - \frac{30}{1.25^4}(t_1 - 1.25)^4 + \frac{12}{1.25^5}(t_1 - 1.25)^5$$

$$(s_{R2}, \dot{s}_{R2}) = goto5(t_1 - 1.25, 1.25, 0, 1.0) \Rightarrow \qquad s_{R2}(t_1) = \frac{10}{1.25^2}(t_1 - 1.25)^3 - \frac{15}{1.25^4}(t_1 - 1.25)^4 + \frac{6}{1.25^5}(t_1 - 1.25)^5$$

$$p_d(t) = \frac{1}{2}(p_{left} + p_{high}) + \frac{1}{2}(p_{left} - p_{high})s_{p2} \qquad R_d(t) = Rot_y\left(\frac{-\pi}{2}\right) Rot_z\left(\frac{\pi}{2}s_{R2}\right)$$

$$v_d(t) = \frac{1}{2}(p_{left} - p_{high})\dot{s}_{p2} \qquad w_d(t) = Rot_y\left(\frac{-\pi}{2}\right) e_z\left(\frac{\pi}{2}\dot{s}_{R2}\right)$$

For $2.50 \leq t_1 < 3.75$ (from p_{left} to p_{high}):

$$(s_{p3}, \dot{s}_{p3}) = goto5(t_1 - 2.50, 1.25, -1.0, 1.0) \Rightarrow \quad s_{p3}(t_1) = -1 + \frac{20}{1.25^2}(t_1 - 2.50)^3 - \frac{30}{1.25^4}(t_1 - 2.50)^4 + \frac{12}{1.25^5}(t_1 - 2.50)^5$$

$$(s_{R3}, \dot{s}_{R3}) = goto5(t_1 - 2.50, 1.25, 1.0, 0) \Rightarrow \quad s_{R3}(t_1) = 1 - \frac{10}{1.25^2}(t_1 - 2.50)^3 + \frac{15}{1.25^4}(t_1 - 2.50)^4 - \frac{6}{1.25^5}(t_1 - 2.50)^5$$

$$p_d(t) = \frac{1}{2}(p_{high} + p_{left}) + \frac{1}{2}(p_{high} - p_{left})s_{p3} \qquad R_d(t) = Rot_y\left(\frac{-\pi}{2}\right) Rot_z\left(\frac{\pi}{2}s_{R3}\right)$$

$$v_d(t) = \frac{1}{2}(p_{high} - p_{left})\dot{s}_{p3} \qquad w_d(t) = Rot_y\left(\frac{-\pi}{2}\right) e_z\left(\frac{\pi}{2}\dot{s}_{R3}\right)$$

For $3.75 \leq t_1 < 5$ (from p_{high} to p_{right}):

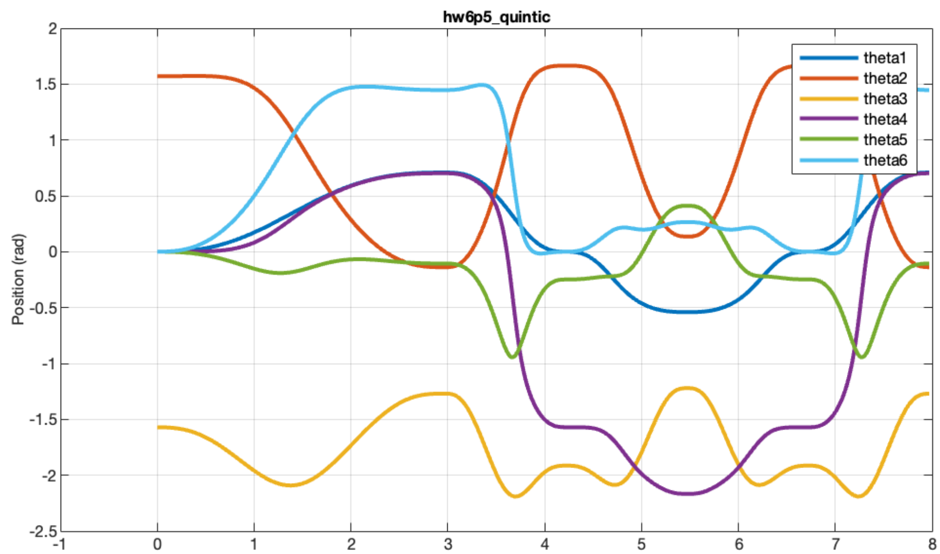
$$(s_{p4}, \dot{s}_{p4}) = goto5(t_1 - 3.75, 1.25, -1.0, 1.0) \Rightarrow \quad s_{p4}(t_1) = -1 + \frac{20}{1.25^2}(t_1 - 3.75)^3 - \frac{30}{1.25^4}(t_1 - 3.75)^4 + \frac{12}{1.25^5}(t_1 - 3.75)^5$$

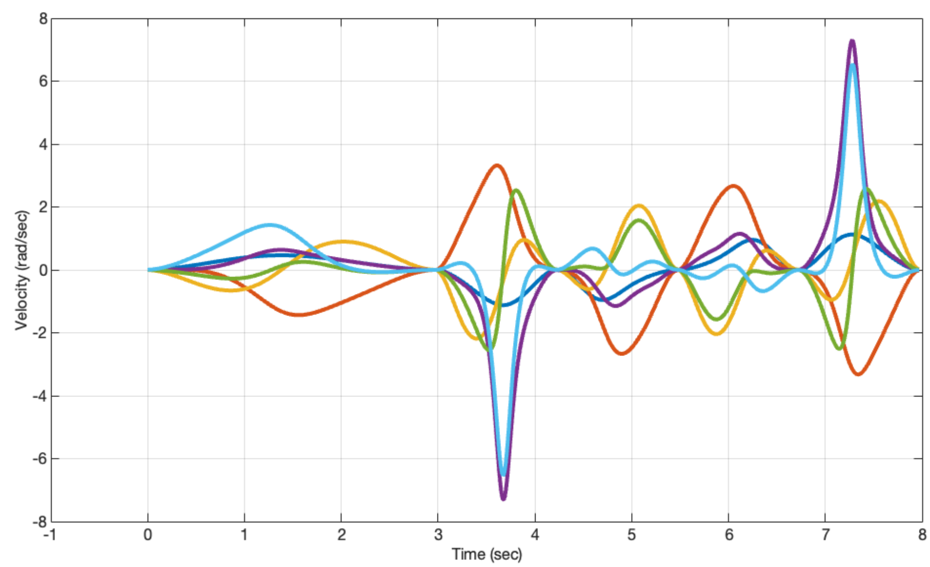
$$(s_{R4}, \dot{s}_{R4}) = goto5(t_1 - 3.75, 1.25, 1.0, 0) \Rightarrow \quad s_{R4}(t_1) = 1 - \frac{10}{1.25^2}(t_1 - 3.75)^3 + \frac{15}{1.25^4}(t_1 - 3.75)^4 - \frac{6}{1.25^5}(t_1 - 3.75)^5$$

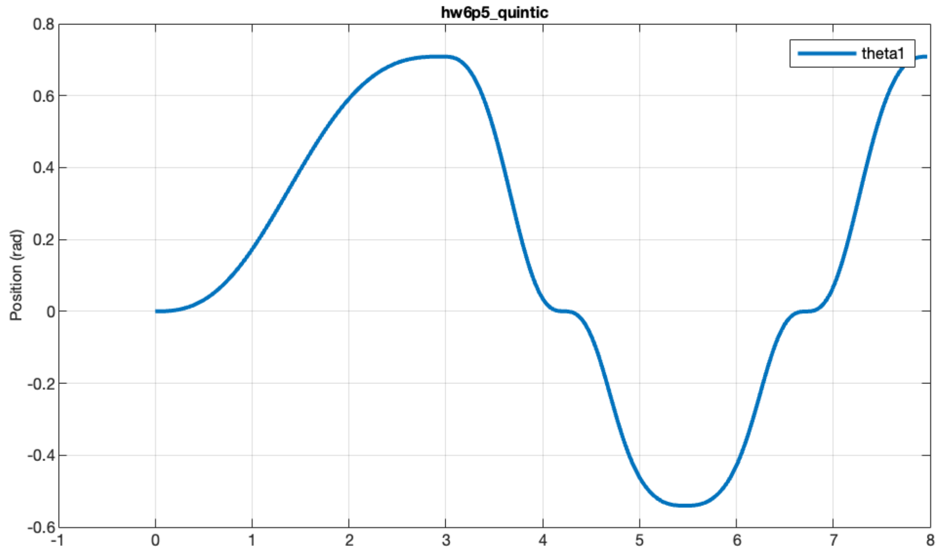
$$p_d(t) = \frac{1}{2}(p_{right} + p_{high}) + \frac{1}{2}(p_{right} - p_{high})s_{p4} \qquad R_d(t) = Rot_y\left(\frac{-\pi}{2}s_{R4}\right)$$

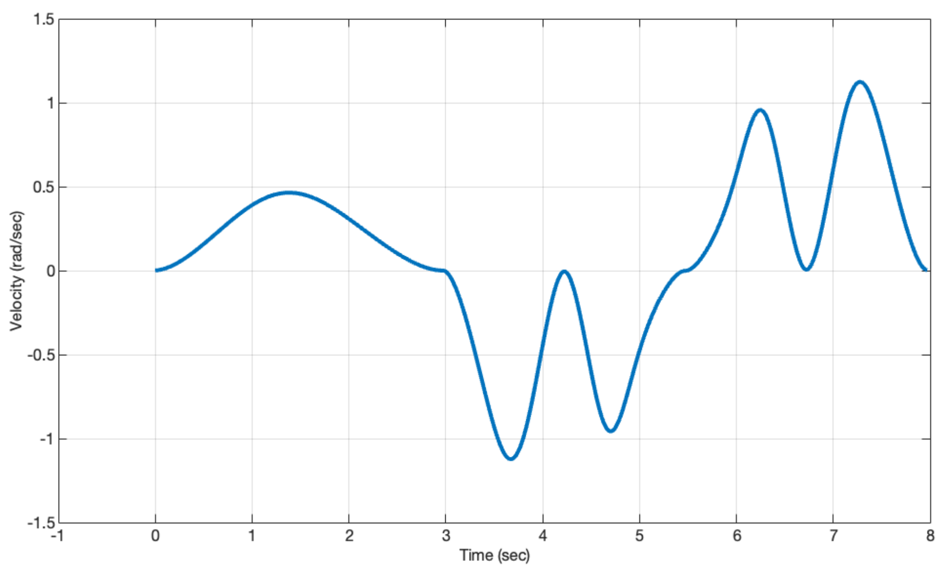
$$v_d(t) = \frac{1}{2}(p_{right} - p_{high})\dot{s}_{p4} \qquad w_d(t) = e_y\left(\frac{-\pi}{2}\dot{s}_{R4}\right)$$

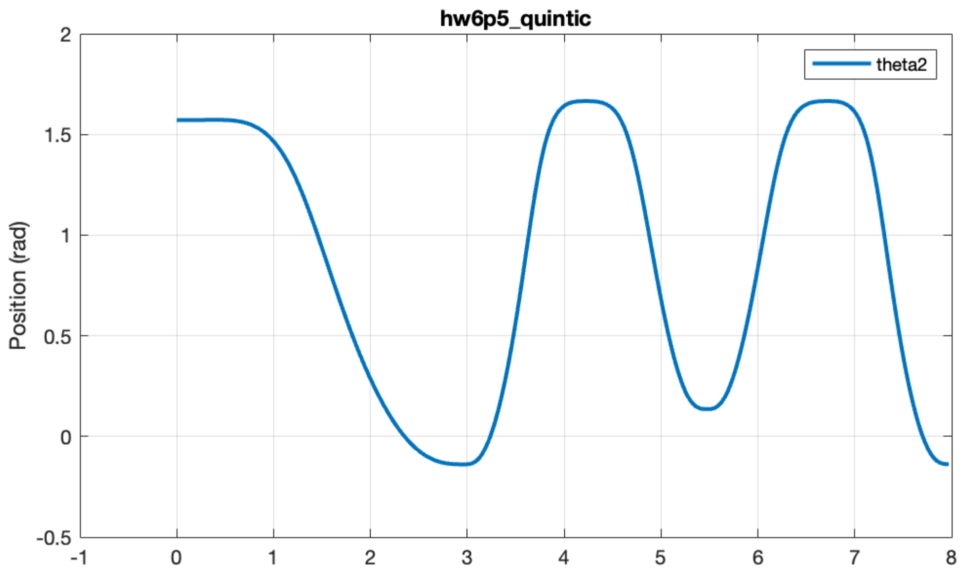
part (c)
Plots:

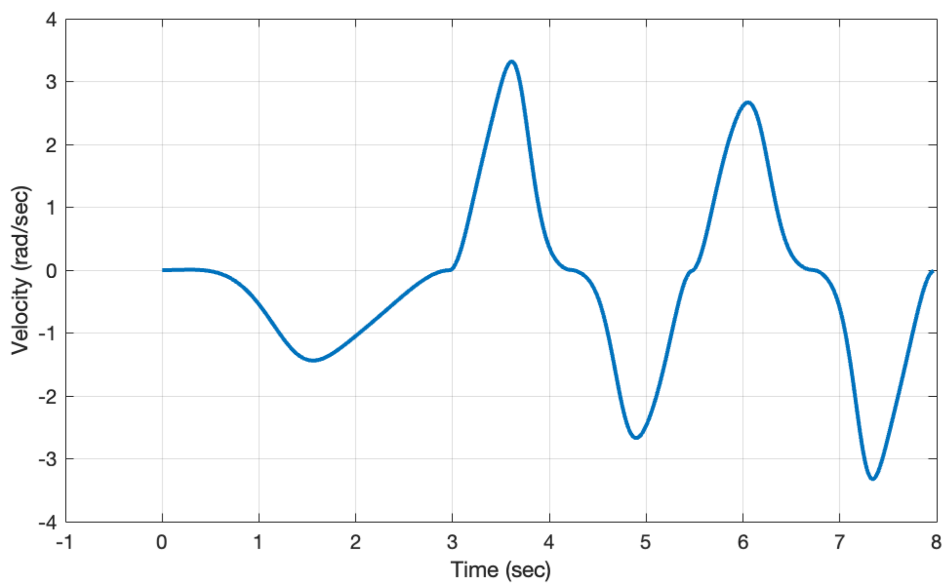

hw6p5_quintic
Position (rad)
theta1
theta2
theta3
theta4
theta5
theta6
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-1
0
1
2
3
4
5
6
7
8

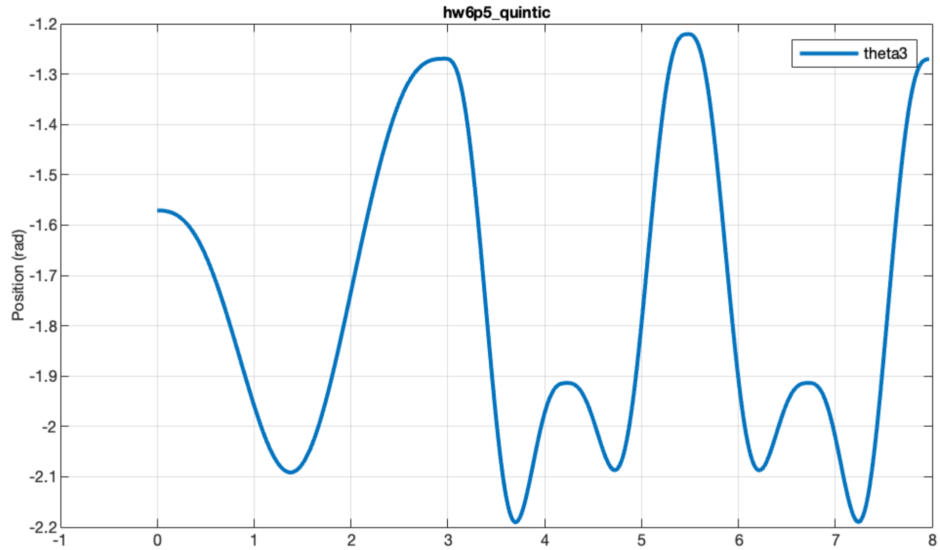

Velocity (rad/sec)
Time (sec)
8
6
4
2
0
-2
-4
-6
-8
-1
0
1
2
3
4
5
6
7
8

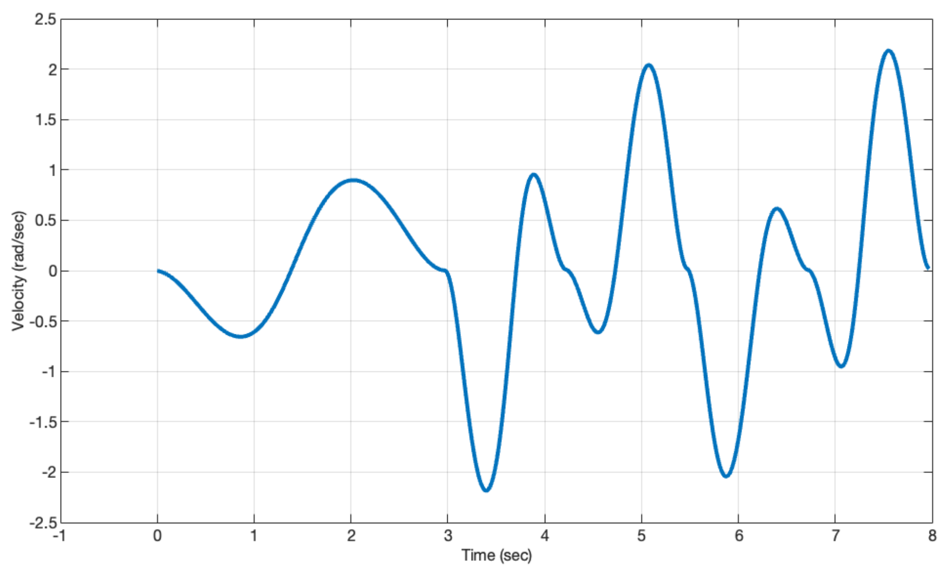

hw6p5_quintic
theta1
Position (rad)
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-1
0
1
2
3
4
5
6
7
8

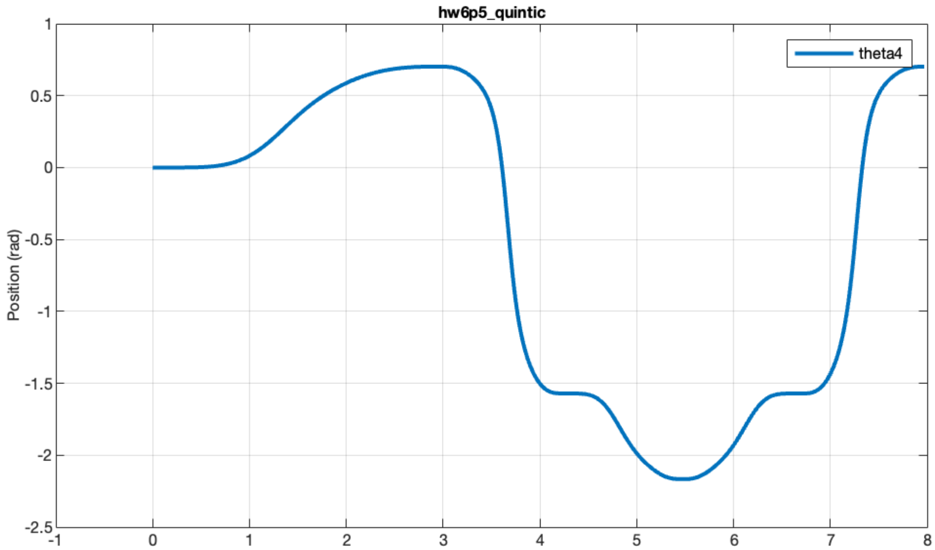

Velocity (rad/sec)
1.5
1
0.5
0
-0.5
-1
-1.5
-1
0
1
2
3
4
5
6
7
8
Time (sec)

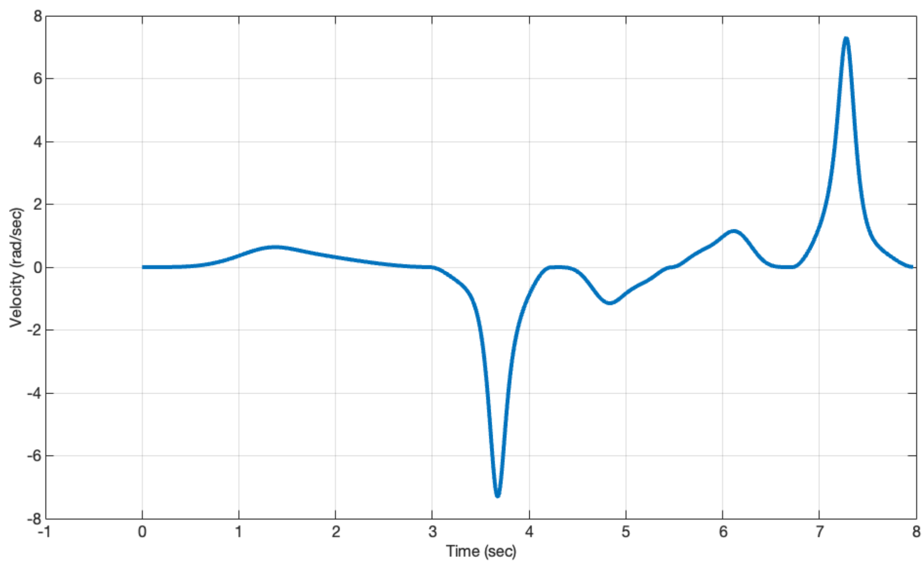

hw6p5_quintic
theta2
Position (rad)
2
1.5
1
0.5
0
-0.5
-1
0
1
2
3
4
5
6
7
8

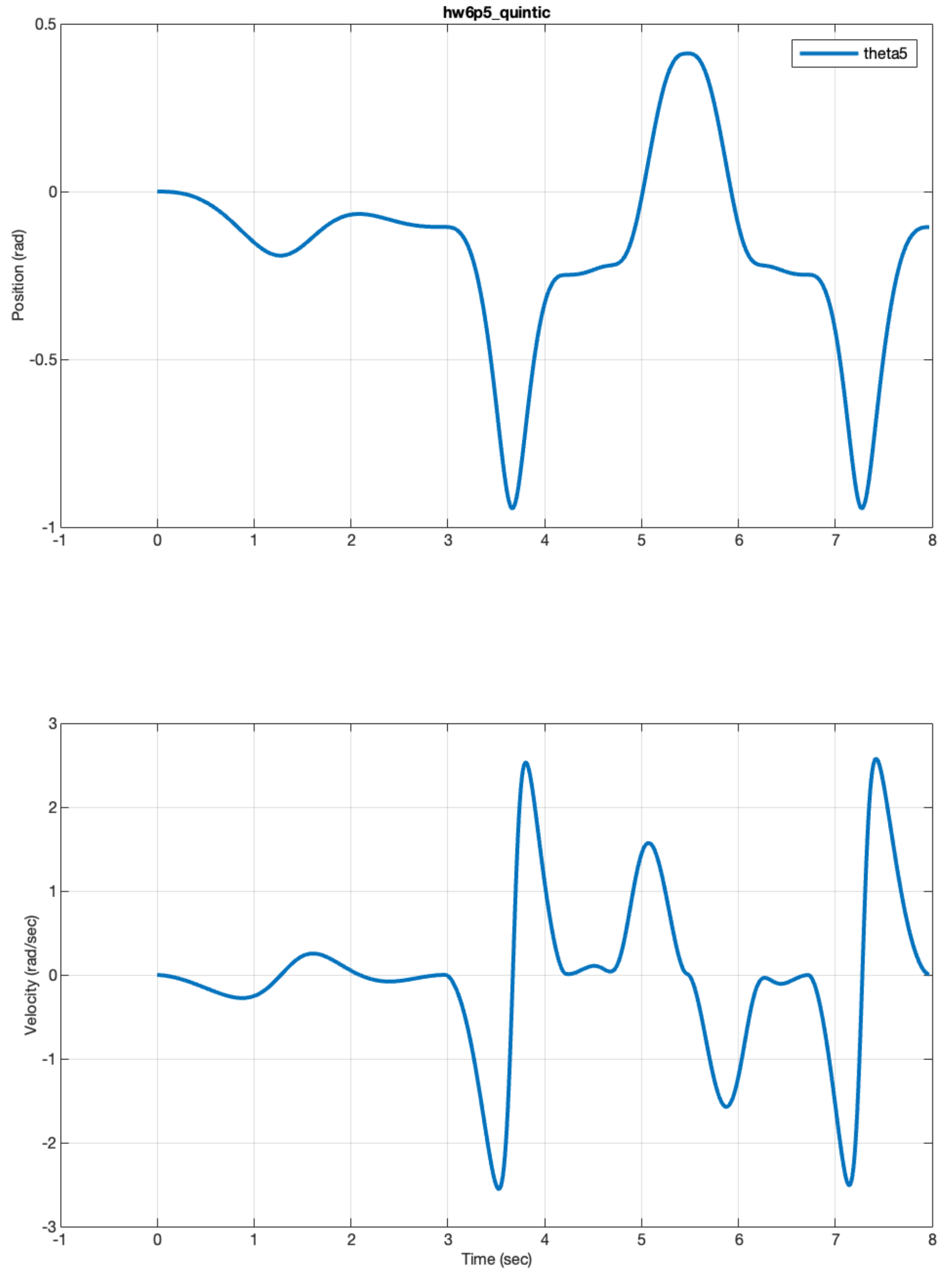

Velocity (rad/sec)
4
3
2
1
0
-1
-2
-3
-4
-1
0
1
2
3
4
5
6
7
8
Time (sec)

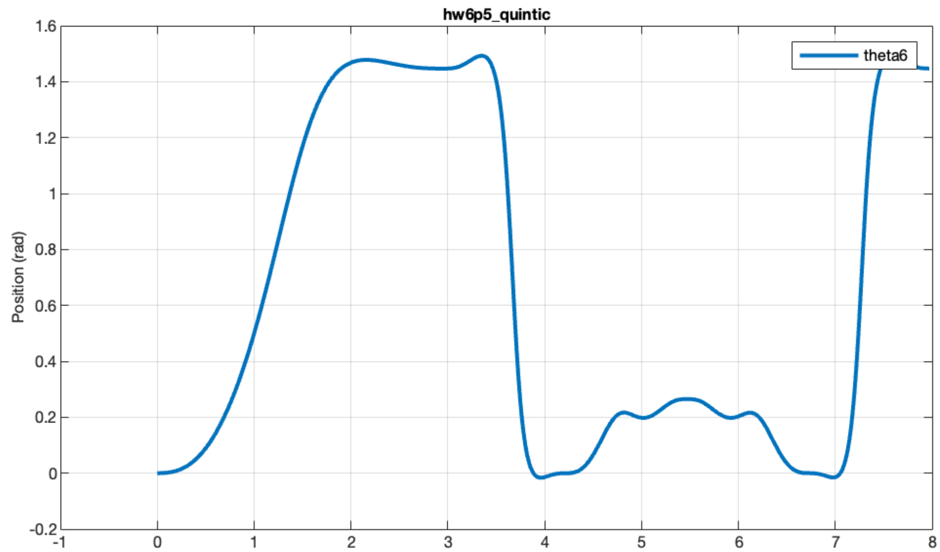

hw6p5_quintic
theta3
Position (rad)
-1.2
-1.3
-1.4
-1.5
-1.6
-1.7
-1.8
-1.9
-2
-2.1
-2.2
-1
0
1
2
3
4
5
6
7
8

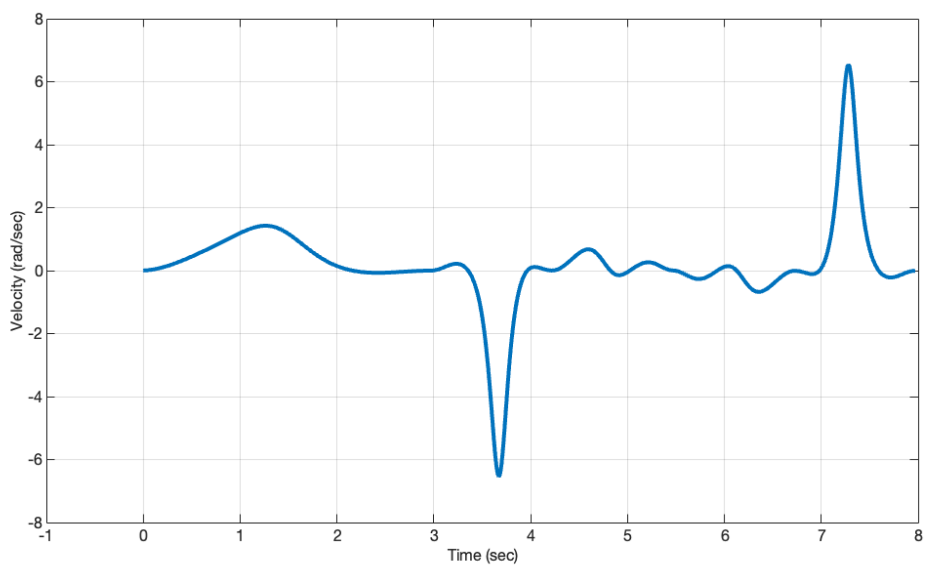

Velocity (rad/sec)
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-1
0
1
2
3
4
5
6
7
8
Time (sec)


hw6p5_quintic
theta4
Position (rad)
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-1
0
1
2
3
4
5
6
7
8


Velocity (rad/sec)
8
6
4
2
0
-2
-4
-6
-8
-1
0
1
2
3
4
5
6
7
8
Time (sec)


hw6p5_quintic
theta5
Position (rad)
Velocity (rad/sec)
Time (sec)


hw6p5_quintic
theta6
Position (rad)
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
-1
0
1
2
3
4
5
6
7
8


Velocity (rad/sec)
8
6
4
2
0
-2
-4
-6
-8
-1
0
1
2
3
4
5
6
7
8
Time (sec)

part (c)
Code

```
class Trajectory():
    # Initialization.
    def __init__(self, node):
        # Set up the kinematic chain object.
        self.chain = KinematicChain(node, 'world', 'tip', self.jointnames())

        # Define the various points.
        self.q0 = np.radians(np.array([0, 90, -90, 0, 0, 0]).reshape((-1,1)))
        self.p0 = np.array([0.0, 0.55, 1.0]).reshape((-1,1))
        self.R0 = Reye()

        self.pleft  = np.array([0.3, 0.5, 0.15]).reshape((-1,1))
        self.phigh = np.array([0.0, 0.5, 0.9]).reshape((-1,1))
        self.pright = np.array([-0.3, 0.5, 0.15]).reshape((-1,1))

        # Initialize the current/starting joint position.
        self.qlast  = self.q0
        self.xd_last = self.p0
        self.Rd_last = self.R0
        self.lam = 20


    # Declare the joint names.
    def jointnames(self):
        # Return a list of joint names FOR THE EXPECTED URDF!
        return ['theta1', 'theta2', 'theta3', 'theta4', 'theta5', 'theta6']

    # Evaluate at the given time.  This was last called (dt) ago.
    def evaluate(self, t, dt):
        #if t >= 8.0:
        #    return None

        if t < 3.0:
            # Goes to from p0 to pright:
            (s0, s0dot) = goto5(t, 3.0, 0.0, 1.0)

            pd = self.p0 + (self.pright - self.p0) * s0
            vd =           (self.pright - self.p0) * s0dot

            Rd = Reye()
            wd = np.array([[0],[0],[0]])

        else:
            t1 = (t-3) % 5.0
            if t1 < 1.25:
                # from pright to phigh
                (sp, spdot) = goto5(t1, 1.25, -1.0, 1.0)
                (sR, sRdot) = goto5(t1, 1.25, 0, 1.0)
```

```
            # Use the path variables to compute the trajectory.
            pd = 0.5*(self.phigh+self.pright) + 0.5*(self.phigh-self.pright) * sp
            vd =                               + 0.5*(self.phigh-self.pright) * spdot

            Rd = Roty(-pi/2 * sR)
            wd = ey() * (-pi/2 * sRdot)
        elif t1 < 2.50:
            # from phigh to pleft
            (sp, spdot) = goto5(t1-1.25, 1.25, -1.0, 1.0)
            (sR, sRdot) = goto5(t1-1.25, 1.25, 0, 1.0)

            # Use the path variables to compute the trajectory.
            pd = 0.5*(self.pleft+self.phigh) + 0.5*(self.pleft-self.phigh) * sp
            vd =                             + 0.5*(self.pleft-self.phigh) * spdot

            Rd = Roty(-pi/2) @ Rotz(pi/2 * sR)
            wd = (Roty(-pi/2) @ ez()) * (pi/2 * sRdot)

        elif t1 < 3.75:
            # from pleft to phigh
            (sp, spdot) = goto5(t1-2.50, 1.25, -1.0, 1.0)
            (sR, sRdot) = goto5(t1-2.50, 1.25, 1.0, 0)

            # Use the path variables to compute the trajectory.
            pd = 0.5*(self.phigh+self.pleft) + 0.5*(self.phigh-self.pleft) * sp
            vd =                             + 0.5*(self.phigh-self.pleft) * spdot

            Rd = Roty(-pi/2) @ Rotz(pi/2 * sR)
            wd = (Roty(-pi/2) @ ez()) * (pi/2 * sRdot)
        else:
            # from phigh to pright
            (sp, spdot) = goto5(t1-3.75, 1.25, -1.0, 1.0)
            (sR, sRdot) = goto5(t1-3.75, 1.25, 1.0, 0)

            # Use the path variables to compute the trajectory.
            pd = 0.5*(self.pright+self.phigh) + 0.5*(self.pright-self.phigh) * sp
            vd =                              + 0.5*(self.pright-self.phigh) * spdot

            Rd = Roty(-pi/2 * sR)
            wd = ey() * (-pi/2 * sRdot)


    # Compute the old forward kinematics.
    (ptip, R, Jv, Jw) = self.chain.fkin(self.qlast)

    # Compute the errors
    error_pos = ep(self.xd_last, ptip)
    error_rot = eR(self.Rd_last, R)
    error = np.vstack((error_pos, error_rot))

    # compute qdot
```

```
v = np.vstack((vd,wd))
A = v + self.lam * error
J = np.vstack((Jv, Jw))
qdot = np.linalg.pinv(J) @ A

# Integrate the joint position.
q = self.qlast + dt * qdot

# Save the data needed next cycle.
self.qlast = q
self.xd_last = pd
self.Rd_last = Rd

# Return the position and velocity as python lists.
return (q.flatten().tolist(), qdot.flatten().tolist())
```

Problem 6 (Time Spent) - 4 points:

I spent about 6.5 hours on this problem set. About 2 hours on P1 and P2, and 4.5 hours on P3-P5. I did not encounter any particular difficulties, just took some time to understand the material (generalized inverses).