CS 133a **November 17, 2023**

Problem Set 7

Problem 1 (Perfect Jacobian Condition Number) - 15 points:

part (a)

Generally we have $\vec{p}_1, \vec{p}_2, \vec{p}_{tip}$ (as shown below), which correspond to the positions of the first joint, second joint, and tip respectively.

$$\vec{p}_1 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$$

$$\vec{p}_2 = \begin{bmatrix} \ell_1 cos(\theta_1) & \ell_1 sin(\theta_1) & 0 \end{bmatrix}^T$$

$$\vec{p}_{tip} = \begin{bmatrix} \ell_1 cos(\theta_1) + \ell_2 cos(\theta_{12}) & \ell_1 sin(\theta_1) + \ell_2 sin(\theta_{12}) & 0 \end{bmatrix}^T$$

where $\theta_{12} = \theta_1 + \theta_2$. So we have:

$$\vec{e}_z \times (\vec{p}_{tip} - \vec{p}_1) = \begin{bmatrix} -(\ell_1 sin(\theta_1) + \ell_2 sin(\theta_{12})) \\ \ell_1 cos(\theta_1) + \ell_2 cos(\theta_{12}) \\ 0 \end{bmatrix}$$

$$\vec{e}_z \times (\vec{p}_{tip} - \vec{p}_2) = \begin{bmatrix} -\ell_2 sin(\theta_{12}) \\ \ell_2 cos(\theta_{12}) \\ 0 \end{bmatrix}$$

Since our task space is two dimensional and pertains to only (x,y), then we have:

$$J = [J_1 \ J_2] \begin{bmatrix} -(\ell_1 sin(\theta_1) + \ell_2 sin(\theta_{12})) & -\ell_2 sin(\theta_{12}) \\ \ell_1 cos(\theta_1) + \ell_2 cos(\theta_{12}) & \ell_2 cos(\theta_{12}) \end{bmatrix}$$

where J_1 and J_2 pertain to the first and second columns of J respectively.
Now, if J has a condition number of 1 then J_1 and J_2 must be orthogonal (as described in the problem). If J_1 and J_2 are orthogonal then we have:

$$J_1 = \begin{bmatrix} -(\ell_1 sin(\theta_1) + \ell_2 sin(\theta_{12})) \\ \ell_1 cos(\theta_1) + \ell_2 cos(\theta_{12}) \end{bmatrix} \Rightarrow J_2 = c \begin{bmatrix} -(\ell_1 cos(\theta_1) + \ell_2 cos(\theta_{12})) \\ -(\ell_1 sin(\theta_1) + \ell_2 sin(\theta_{12})) \end{bmatrix}$$

where $c \in \mathbb{R}$ (c is a scalar). However, since J_1 and J_2 must also have the same magnitude (as described by the problem), then there are only two solutions such that $c = \pm 1$. For the case $c = -1$ we have:

$$J_2 = \begin{bmatrix} \ell_1 cos(\theta_1) + \ell_2 cos(\theta_{12}) \\ \ell_1 sin(\theta_1) + \ell_2 sin(\theta_{12}) \end{bmatrix} = \begin{bmatrix} -\ell_2 sin(\theta_{12}) \\ \ell_2 cos(\theta_{12}) \end{bmatrix}$$

For the case $c = 1$ we have:

$$J_2 = \begin{bmatrix} -(\ell_1 cos(\theta_1) + \ell_2 cos(\theta_{12})) \\ -(\ell_1 sin(\theta_1) + \ell_2 sin(\theta_{12})) \end{bmatrix} = \begin{bmatrix} -\ell_2 sin(\theta_{12}) \\ \ell_2 cos(\theta_{12}) \end{bmatrix}$$

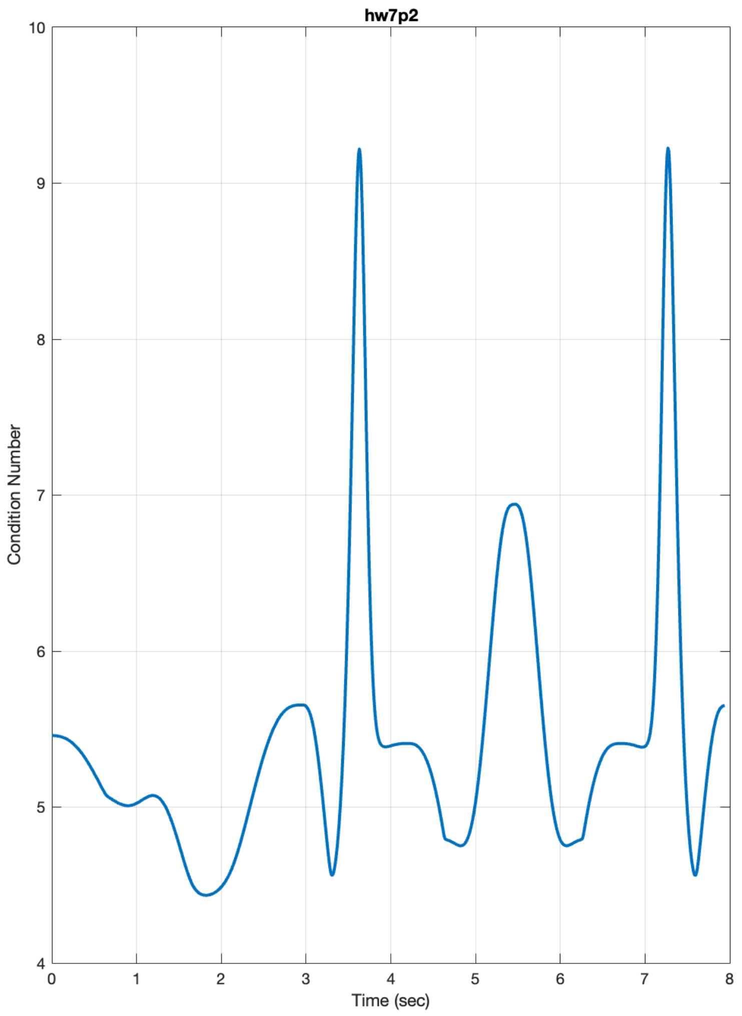
Therefore we must either have the following two equations below to be satisfied:

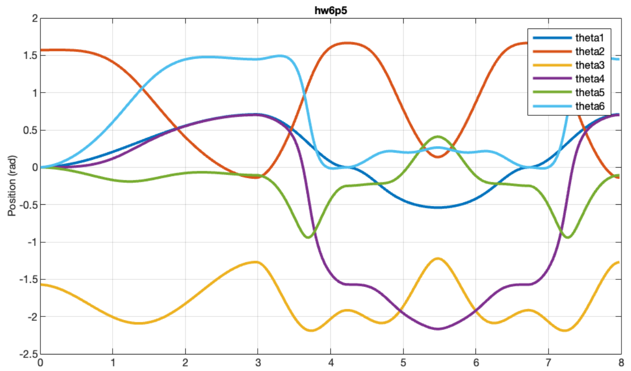
$$-\ell_2 sin(\theta_{12}) = \ell_1 cos(\theta_1) + \ell_2 cos(\theta_{12}) \qquad \ell_2 cos(\theta_{12}) = \ell_1 sin(\theta_1) + \ell_2 sin(\theta_{12})$$

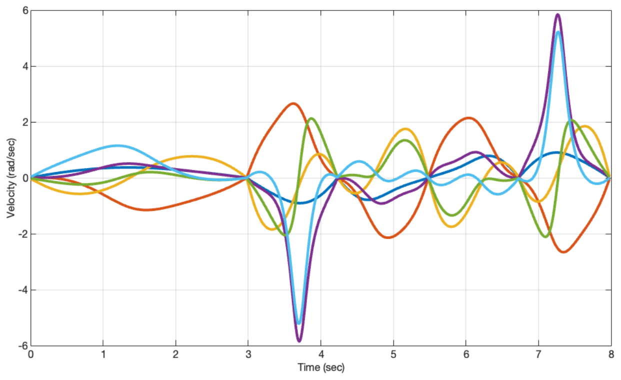
Or the following two below to be satisfied:

$$-\ell_2 sin(\theta_{12}) = -(\ell_1 cos(\theta_1) + \ell_2 cos(\theta_{12})) \qquad \ell_2 cos(\theta_{12}) = -(\ell_1 sin(\theta_1) + \ell_2 sin(\theta_{12}))$$

Problem 2 (Condition Numbers for the Touch and Go Movement) - 15 points:







Code for 2:

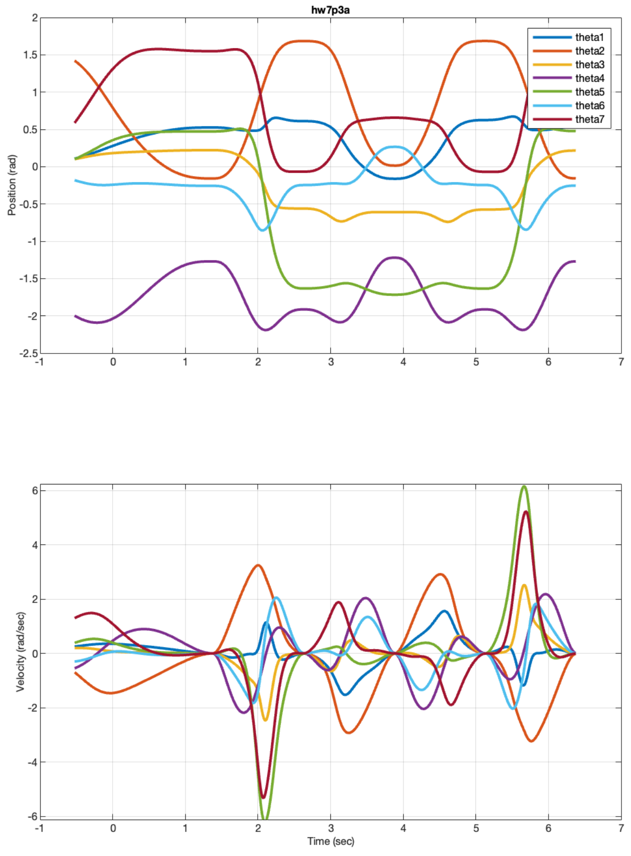
```
class Trajectory():
    # Initialization.
    def __init__(self, node):
        # Set up the kinematic chain object.
        self.chain = KinematicChain(node, 'world', 'tip', self.jointnames())

        # 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

        self.pub = node.create_publisher(Float64, '/condition', 10)


    # 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

        L = 0.4
        Jbar = np.diag([1/L, 1/L, 1/L, 1, 1, 1]) @ J
        condition = np.linalg.cond(Jbar)
        msg = Float64()
        msg.data = condition
```

```
    self.pub.publish(msg)

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

Problem 3 (Redundant 7 DOF Robot) – 33 points:

part (a)

The motion looks very similar to the motion of the 6R robot. The values of the joint angles over time have little change. The changes are noticeable between t=3 and t=5 seconds which is where theta3 (the new joint angle) deviates from zero the most. During this time, some of the other joint angles also change though the change is relatively small as stated (refer to the joint plot in problem 2 for comparison). Below is the new joint plot:

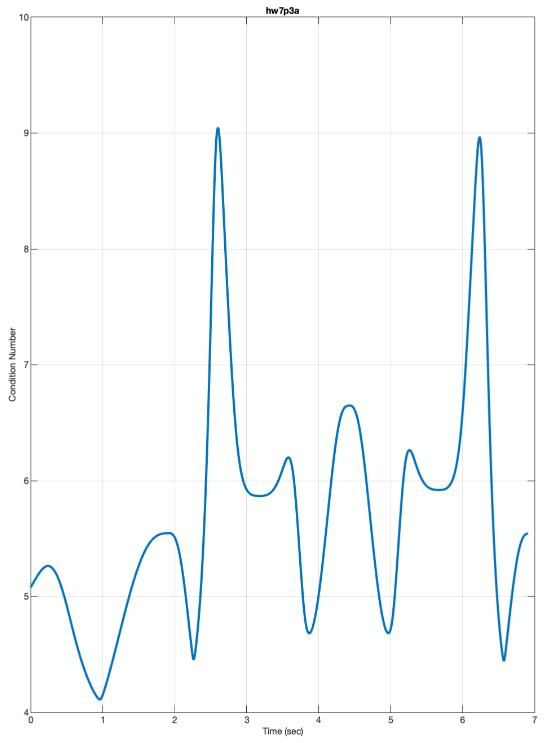


Inverse Kinematics Formula used for 3a:

$$\dot{q}(t_k) = J^{+}(q(t_{k-1}))\left(\begin{bmatrix}\dot{x}_d(t_k)\\ \omega_d(t_k)\end{bmatrix} + \lambda\begin{bmatrix}e(p_d(t_{k-1}), p(t_{k-1}))\\ e(R_d(t_{k-1}), R(t_{k-1}))\end{bmatrix}\right)$$

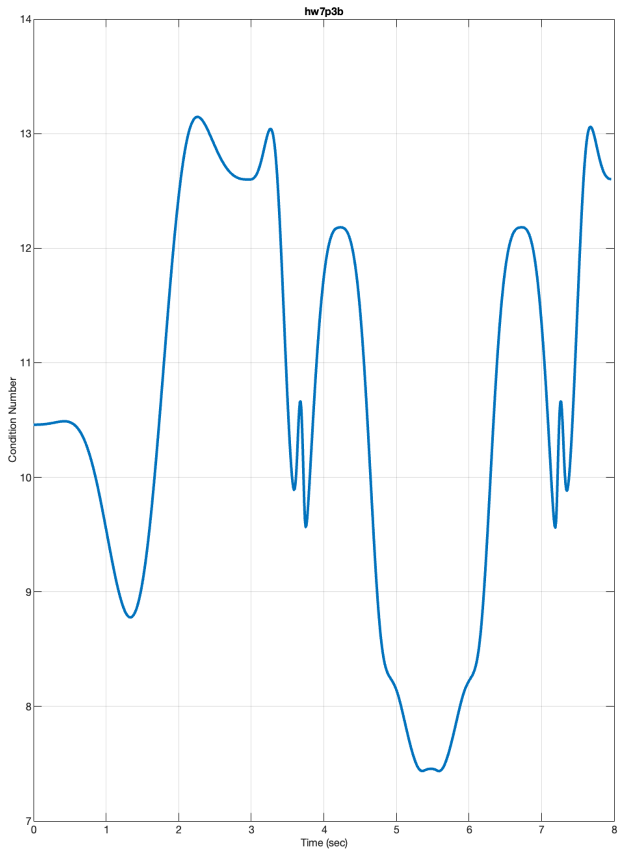
Inverse Kinematics Formula code for 3a:

```
# 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
```

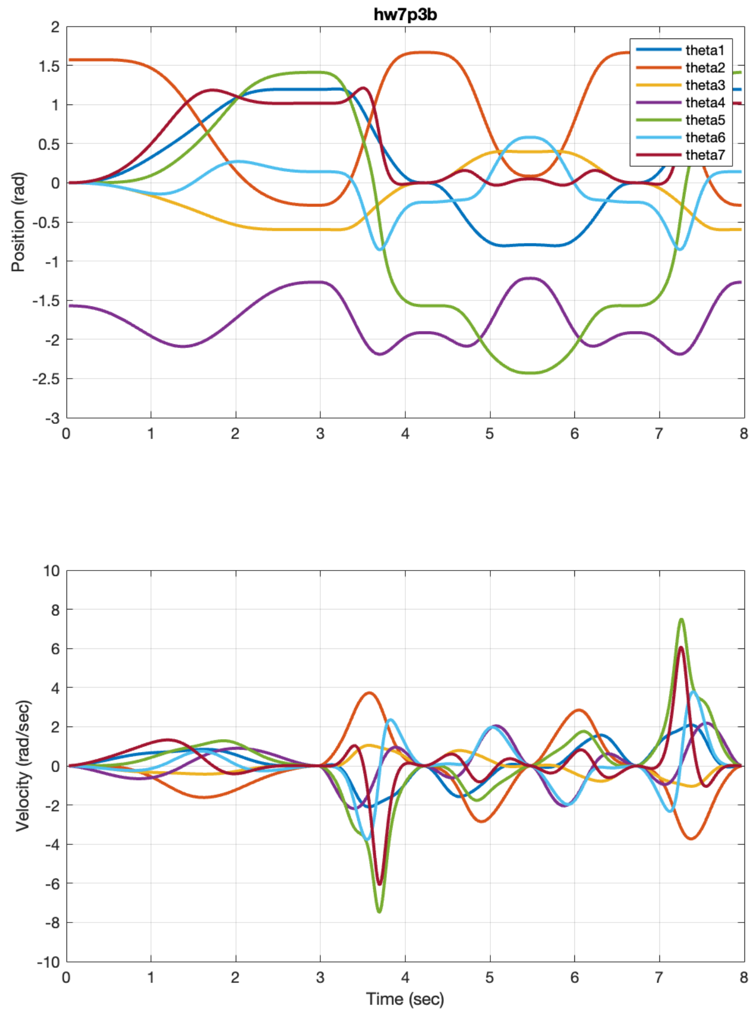


part (b)

Qualitatively, this should make the motion worse as we essentially trying to reduce the DOF by making θ_3 track $-\frac{1}{2}\theta_1$. Furthermore, comparing the condition numbers from 3a, we see that making θ_3 track $-\frac{1}{2}\theta_1$ leads to a higher condition number (at least for my trajectory):



Joint plot for 3b:



Inverse Kinematics Formula used for 3b:

$$\dot{q}(t_k) = J^{+}(q(t_{k-1}))\left(\begin{bmatrix}\dot{x}_d(t_k)\\ \omega_d(t_k)\\ \dot{x}_7^{des}\end{bmatrix} + \lambda\begin{bmatrix}e(p_d(t_{k-1}), p(t_{k-1}))\\ e(R_d(t_{k-1}), R(t_{k-1}))\\ e(x_7^{des}, x_7(t_{k-1}))\end{bmatrix}\right)$$

Note that $x_7^{des} = 0$ as given in the problem. Therefore, we also have $\dot{x}_7^{des} = 0$. Also note that a new row was added to the Jacobian. So the Jacobian has the form:

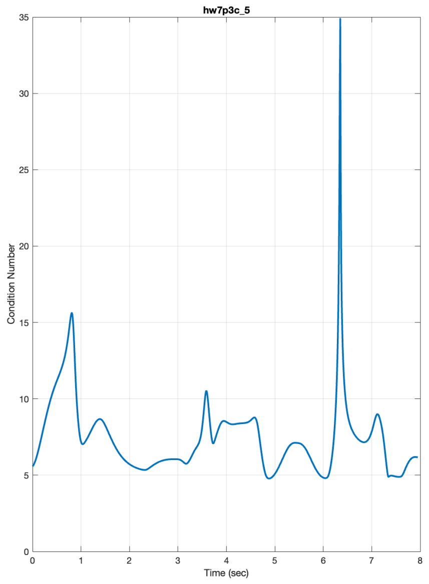
$$J = \begin{bmatrix} & & & J_v & & & \\ & & & J_\omega & & & \\ 0.5 & 0 & 1 & 0 & 0 & 0 & 0\end{bmatrix}$$

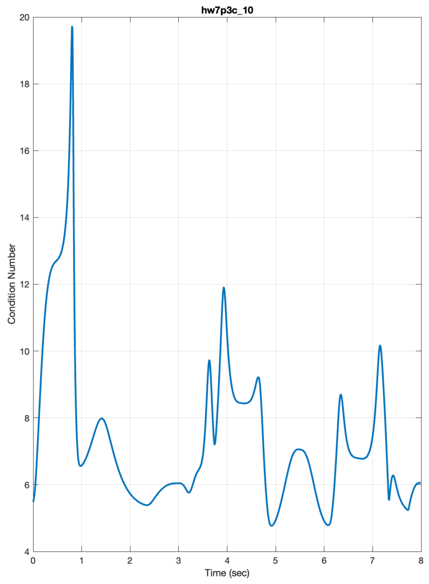
Inverse Kinematics code used for 3b:

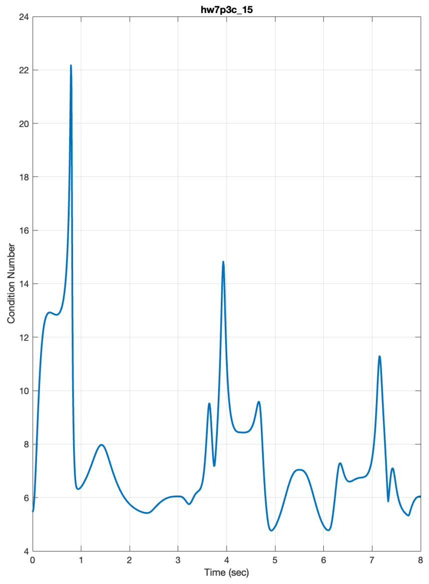
```
# Compute the old forward kinematics.
(ptip, R, Jv, Jw) = self.chain.fkin(self.qlast)

# compute the previous value of the 7th task
x7 = 0.5 * self.qlast[0,0] + self.qlast[2,0]

# Compute the errors
error_pos = ep(self.xd_last, ptip)
error_rot = eR(self.Rd_last, R)
error_seven = self.x7_d - x7 #error of the 7th task
error_seven = np.array([[error_seven]])
error = np.vstack((error_pos, error_rot, error_seven))

# compute qdot, added 7th task to v and J
v = np.vstack((vd,wd, self.x7_dot_d))
A = v + self.lam * error
J = np.vstack((Jv, Jw, self.J7))
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
```

part (c)

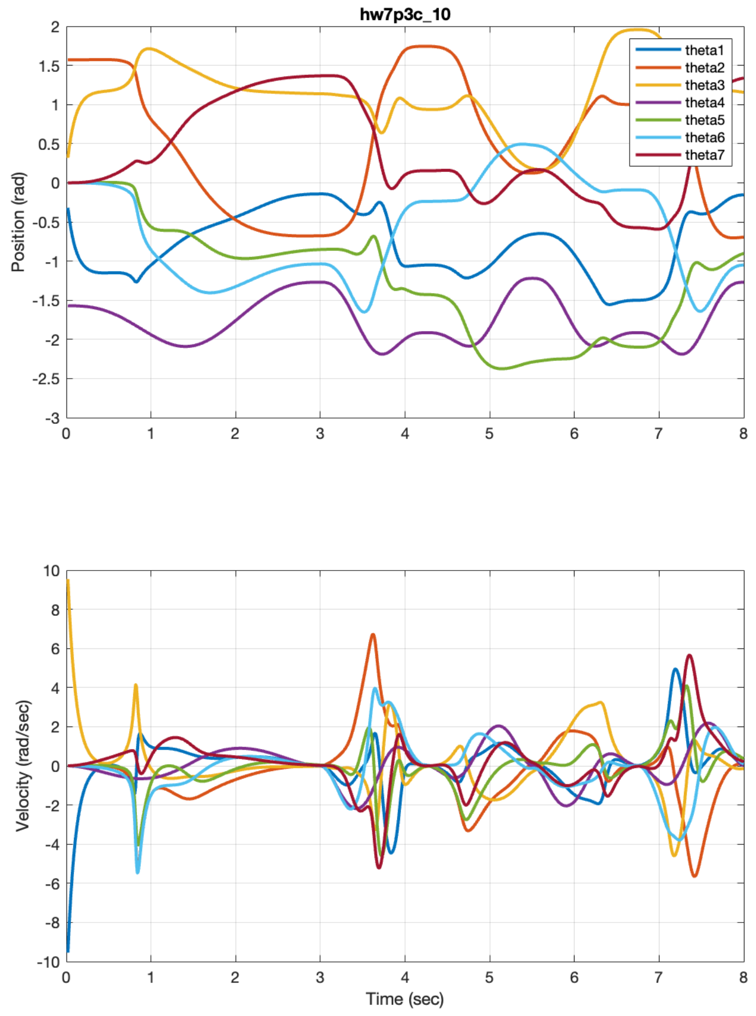
$\lambda_s = 10$ was a reasonable gain (λ_s refers to gain of the secondary task).. It was a good compromise between condition number and achieving the secondary task. For reference here are the condition numbers when $\lambda_s = 5, 10$ and 15.





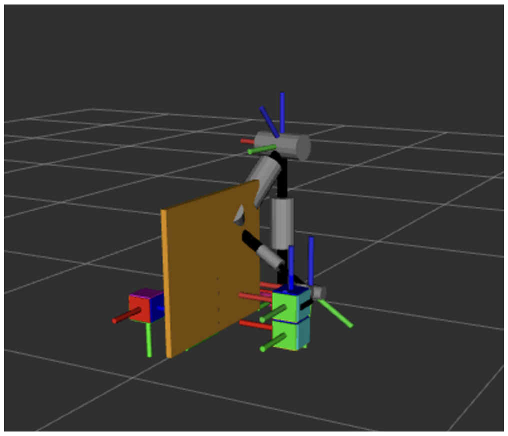


Joint plot for 3c:

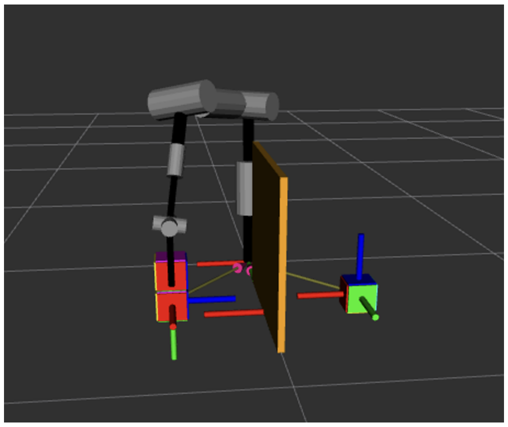


As shown above, when $\lambda_s = 10$, we see that (for the most part) the values of the joint angles stay in the ranges as give in the problem. An exception seems to by θ_5 where is goes below it range to about -2.4 radians. Note that changing the value of λ_s from 10 didn't seem to fix this exception.

At the right target we the configuration $q = \begin{bmatrix} -0.15 & -0.69 & 1.16 & -1.27 & -0.90 & -1.05 & 1.34 & \end{bmatrix}^T$. Here is an image:



At the left target we the configuration $q = \begin{bmatrix} -0.66 & 0.12 & 0.18 & -1.22 & -2.29 & 0.48 & 0.16 \end{bmatrix}^T$. Here is an image:



From the configuration above wee that the robot collides with the obstacle when it reached the target on the right. This is due to the secondary task. More specifically it is the desired range of θ_1 and θ_2 that leads to this. As we try to drive θ_1 to $\frac{-\pi}{4}$ and θ_2 , this makes the robot want to face towards the target on the left. Thus when it reaches the target on the right, it will need to bend its arm in a way that makes it collide with the obstacle. Furthermore, the secondary task does not account for the obstacle being there.

Inverse Kinematics Formula used for 3c (same as the one given in the generalized inverse notesf for secondary tasks)

$$\dot{q}(t_k) = J^+(q(t_{k-1}))\left(\begin{bmatrix}\dot{x}_d(t_k)\\ \omega_d(t_k)\end{bmatrix} + \lambda\begin{bmatrix}e(p_d(t_{k-1}), p(t_{k-1}))\\ e(R_d(t_{k-1}), R(t_{k-1}))\end{bmatrix}\right) + (I_N - J^+J)\dot{q}_{secondary}$$

where J is the Jacobian of the primary task, λ is the gain of the primary task, and N is the number of joints. $\dot{q}_{secondary}$ is definedd as:

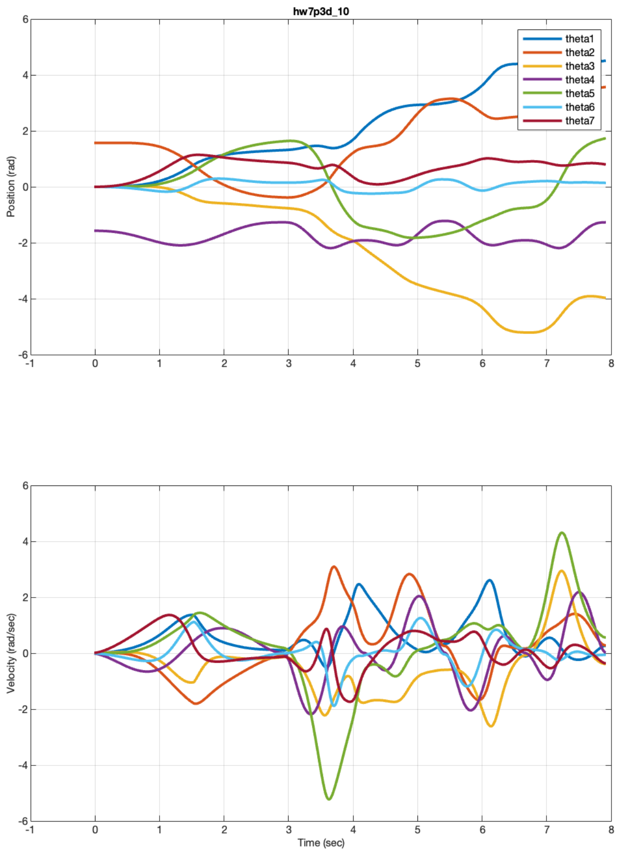
$$\dot{q}_{secondary} = \lambda_s(q_{goal} - q(t_{k-1}))$$

λ_s is the gain of the secondary task and $q_{goal} = [-\frac{\pi}{4}, -\frac{\pi}{4}, \frac{\pi}{2}, -\frac{\pi}{2}, 0, 0, 0]^T$

Inverse Kinematics Formula code used for 3c:

```
# 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))

# secondary qdot
qdot_s = self.lam_s * (self.q_goal_s - self.qlast)

(M, N) = J.shape
qdot = (np.linalg.pinv(J) @ A) + ( (np.eye(N) - np.linalg.pinv(J) @ J) @ qdot_s)

# 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
```

part (d)
Setting $c_{repulse} = 10$ seems to reasonable for my trajectory. When $c_{repulse}$ was low (around 5) the robot would collide with the obstacle, the same occurred when $c_{repulse}$ was high. In other words, when $c_{repulse}$ was high, overshooting (the velocity was adjusted too fast) occurs making the robot have a configuration that deviates more from the secondary task. And when $c_{repulse}$ was too low, the undershooting occurred (the velocity was not adjusted fast enough to prevent the robot from colliding with the obstacle).
Joint plot for 3d:



Inverse Kinematics Formula used for 3d

$$\dot{q}(t_k) = J^+(q(t_{k-1}))\left(\begin{bmatrix}\dot{x}_d(t_k)\\ \omega_d(t_k)\end{bmatrix} + \lambda\begin{bmatrix}e(p_d(t_{k-1}), p(t_{k-1}))\\ e(R_d(t_{k-1}), R(t_{k-1}))\end{bmatrix}\right) + (I_N - J^+J)\dot{q}_{secondary}$$

where J is the Jacobian of the primary task, λ is the gain of the primary task, and N is the number of joints. $\dot{q}_{secondary}$ is definedd as:

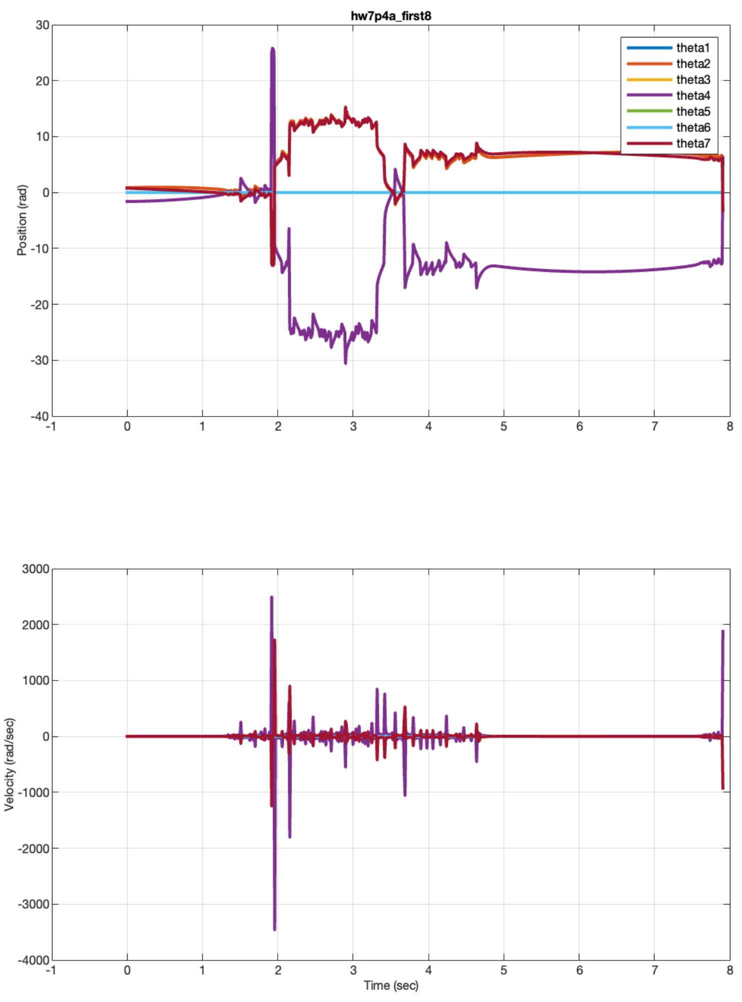
$$\dot{q}_{secondary} = \begin{bmatrix}\frac{c_{repulse}\cdot\theta_1}{\theta_1^2+\theta_2^2} & \frac{c_{repulse}\cdot max(|\theta_1|,\theta_2)}{\theta_1^2+\theta_2^2} & 0 & 0 & 0 & 0 & 0\end{bmatrix}^T$$

Inverse Kinematics code used for 3d

```
# 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))

# secondary qdot
theta1 = self.qlast[0,0]
theta2 = self.qlast[1,0]
max_theta = max(abs(theta1), theta2)
denom = theta1**2 + theta2**2
q_s = np.array([[theta1], [max_theta], [0], [0], [0], [0], [0]])
qdot_s = (self.crepulse/denom) * q_s

(M, N) = J.shape
qdot = (np.linalg.pinv(J) @ A) + ( (np.eye(N) - np.linalg.pinv(J) @ J) @ qdot_s)

# 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
```

Problem 4 (Handling the Singularity) – 33 points:

part (a)
As shown below, the velocity drastically increases, (essential leading to having close to an instantaneous change in both joint position and velocity).
Joint plot for 4a

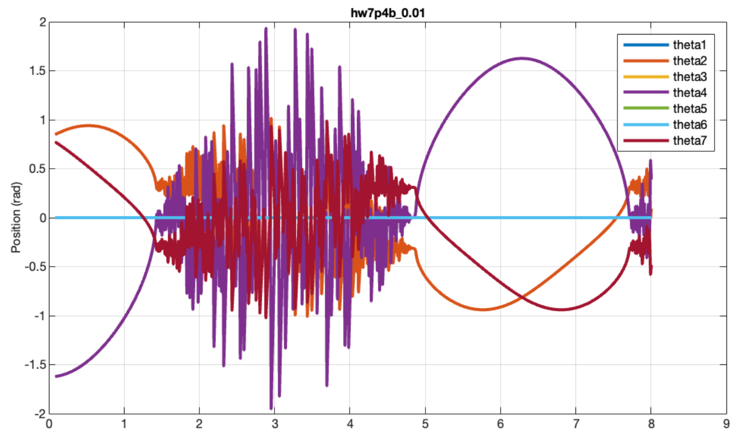


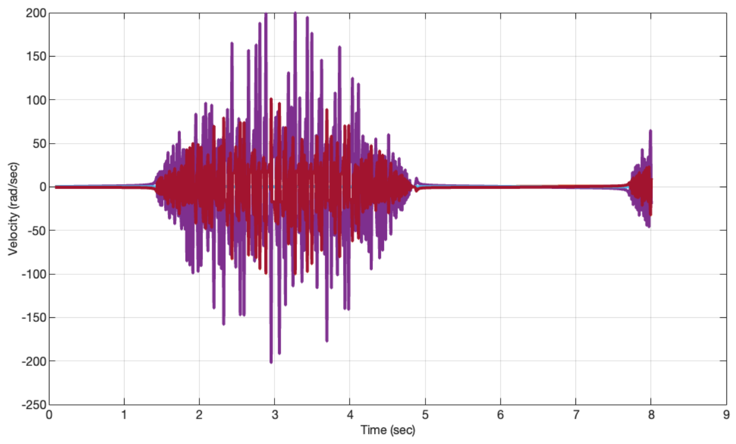
Inverse Kinematics code used for 4a

```
# 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
```

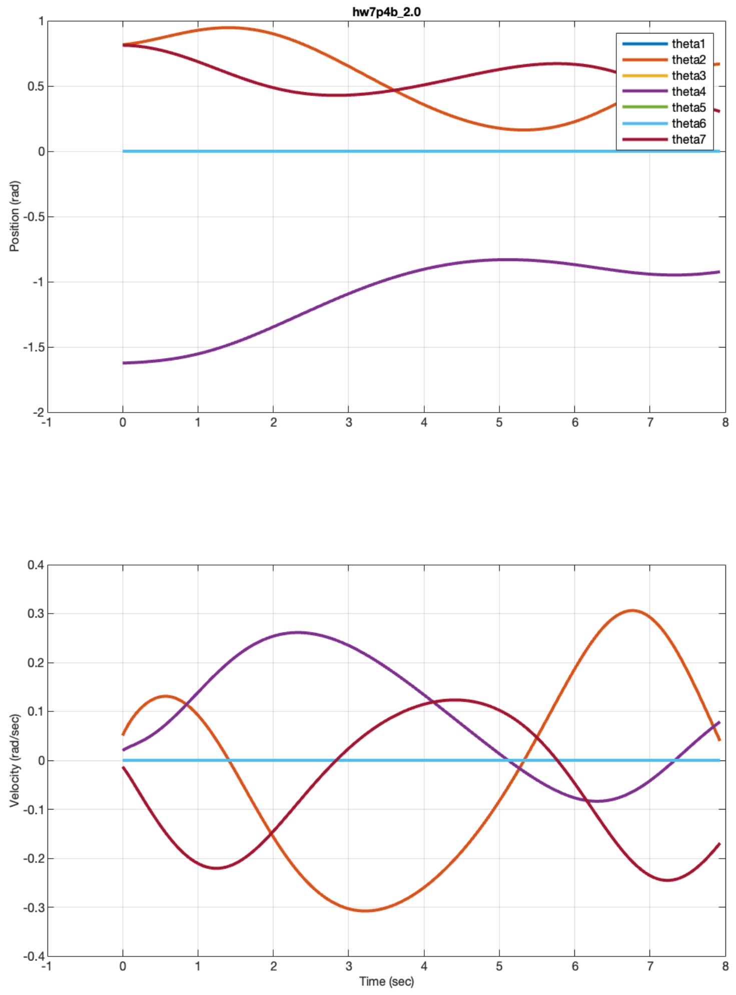
part (b)
When γ is too small, there is still instantaneous changes in both joint position and velocity. When γ is too big, the joint become more "stiff". The joint velocity and position vary less. A reasonable γ would be 0.10.

Plot when γ is too low ($\gamma = 0.01$)





Plot when γ is too big ($\gamma = 2.0$)



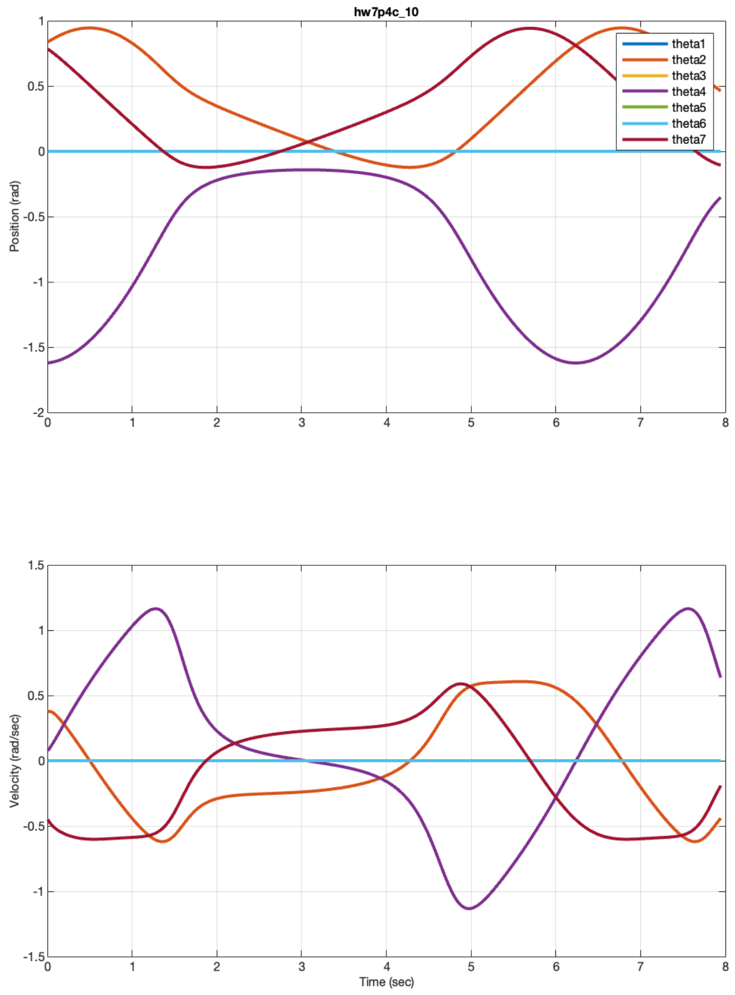
Inverse Kinematics code used for 4b

```
# 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))

#get weighted psuedo inverse
JT = np.transpose(J)
JW_pinv = JT @ np.linalg.inv(J @ JT + (self.gamma**2) * np.eye(6))
qdot = JW_pinv @ 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
```

part (c)

$$\dot{q}_{secondary} = \begin{bmatrix}0 & 0 & 0 & \lambda_s(\frac{-\pi}{4} - \theta_4(t)) & 0 & 0 & 0\end{bmatrix}^T$$

Joint data with $\lambda_s = 10$



Inverse Kinematics used for 4c

```
# 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))
```

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

theta4 = self.qlast[3,0]
qdot_s = np.array([0,0,0, self.lam_s * (-pi/2 - theta4) ,0,0,0]).reshape((-1,1))

#get weighted psuedo inverse
JT = np.transpose(J)
JW_pinv = JT @ np.linalg.inv(J @ JT + (self.gamma**2) * np.eye(6))

# compute qdot
qdot = JW_pinv @ A  + ((np.eye(7) - JW_pinv @ J) @ qdot_s)

# 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
```

Problem 5 (Time Spent) - 4 points:

I spent about 6 hours on the set. I think I over thought number 1, so I spent more time on it than was intended. For the most part, I thought the course structure was fine (in terms of problem sets, quizzes, and final project). The only improvement I can think of is trying to release sets a bit sooner (sometimes the sets were release a little later than expected).