CS 133a November 1, 2023

Problem Set 5

Problem 1 (3 DOF Jacobian - NumPy Warm-up) - 10 points:

Output for Test Case 1:

```
TEST CASE #1:
q:
 [[ 0.34906585]
 [ 0.6981317 ]
 [-0.52359878]]
fkin(q):
 [[-0.59882672]
 [ 1.64526289]
 [ 0.81643579]]
Jac(q):
 [[-1.64526289  0.27923749  0.05939117]
 [-0.59882672 -0.76719868 -0.16317591]
 [ 0.          1.7508522   0.98480775]]
```

Output for Test Case 2:

```
TEST CASE #2
q:
 [[0.52359878]
 [0.52359878]
 [1.04719755]]
fkin(q):
 [[-0.4330127]
 [ 0.75     ]
 [ 1.5      ]]
Jac(q):
 [[-0.75        0.75        0.5       ]
 [-0.4330127  -1.29903811 -0.8660254 ]
 [ 0.          0.8660254   0.        ]]
```

Code:

```
# Forward Kinematics
def fkin(q):
    #print("Put the forward kinematics here")
    theta_pan, theta_1, theta_2 = q[0,0], q[1,0],  q[2,0]
    x_tip = -np.sin(theta_pan) * (np.cos(theta_1) + np.cos(theta_1 + theta_2))
    y_tip = np.cos(theta_pan) * (np.cos(theta_1) + np.cos(theta_1 + theta_2))
    z_tip = np.sin(theta_1) + np.sin(theta_1 + theta_2)
    x = np.array([x_tip, y_tip, z_tip]).reshape(-1,1)
```

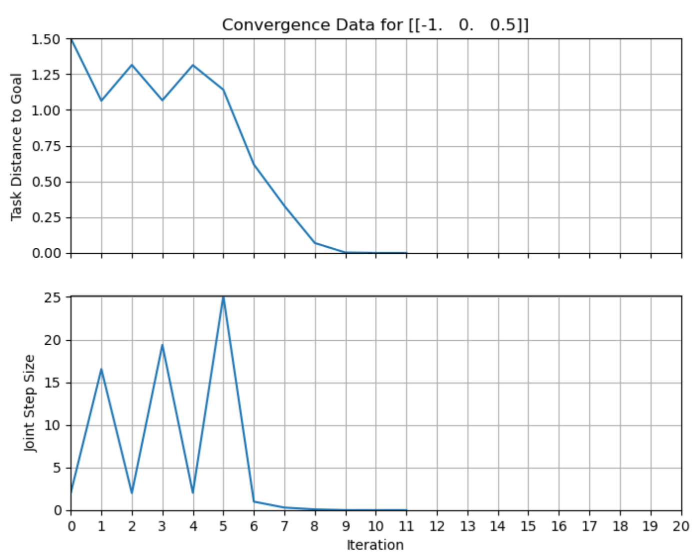
```
    # Return the tip position as a numpy 3x1 column vector.
    return x

# Jacobian
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
```

Problem 2 (Newton Raphson Algorithm) - 24 points:

xgoal	Report
$\begin{bmatrix}0.5\\1.0\\0.5\end{bmatrix}$	a) The algorithm converged b) 5 steps required for $\|\|x_{goal} - x(q(i))\|\| < 10^{-12}$ c) Final $q = \begin{bmatrix}-0.46364761\\1.33227263\\-1.82347658\end{bmatrix}$ d) Corresponds to an elbow up and front side solution. e) For the final q, all joints wrap around 360° 0 times.
$\begin{bmatrix}1.0\\0.5\\0.5\end{bmatrix}$	a) The algorithm converged b) 7 steps required for $\|\|x_{goal} - x(q(i))\|\| < 10^{-12}$ c) Final $q = \begin{bmatrix}-1.10714872\\1.33227263\\-1.82347658\end{bmatrix}$ d) Corresponds to an elbow up and front side solution e) For the final q, all joints wrap around 360° 0 times.
$\begin{bmatrix}2.0\\0.5\\0.5\end{bmatrix}$	a) The algorithm did not converge b) to e) no answer needed as the algorithm did not converge
$\begin{bmatrix}0\\-1\\0.5\end{bmatrix}$	a) The algorithm did not converge b) to e) no answer needed as the algorithm did not converge
$\begin{bmatrix}0\\-0.6\\0.5\end{bmatrix}$	a) The algorithm converged b) 14 steps required for $\|\|x_{goal} - x(q(i))\|\| < 10^{-12}$ c) Final $q = \begin{bmatrix}0\\1.27724626\\-3.94396908\end{bmatrix}$ d) Corresponds to an elbow down and back side solution e) For the final q, θ_{pan} and θ_1 wrap around 360° 0 times. θ_2 wraps around 360° 1 time (in the negative direction)

$\begin{bmatrix} 0.5 \\ -1 \\ 0.5 \end{bmatrix}$	a) The algorithm converged b) 8 steps required for $\lvert\lvert x_{goal} - x(q(i))\rvert\rvert < 10^{-12}$ c) Final $q = \begin{bmatrix} 28.73798149 \\ 1.33227263 \\ -1.82347658 \end{bmatrix}$ d) Corresponds to an elbow up and front side solution e) For the final q, θ_1 and θ_2 wrap around 360° 0 times. θ_{pan} wraps around 360° 5 times (in the positive direction)
$\begin{bmatrix} -1 \\ 0 \\ 0.5 \end{bmatrix}$	a) The algorithm converged b) 11 steps required for $\lvert\lvert x_{goal} - x(q(i))\rvert\rvert < 10^{-12}$ c) Final $q = \begin{bmatrix} 23.5619449 \\ -8.91082902 \\ -1.9551931 \end{bmatrix}$ d) Corresponds to an elbow up and back side solution e) For the final q, θ_2 wraps around 360° 0 times. θ_{pan} wraps around 360° 4 times (in the positive direction). θ_1 wraps around 360° 1 time in the negative direction

Last plot:



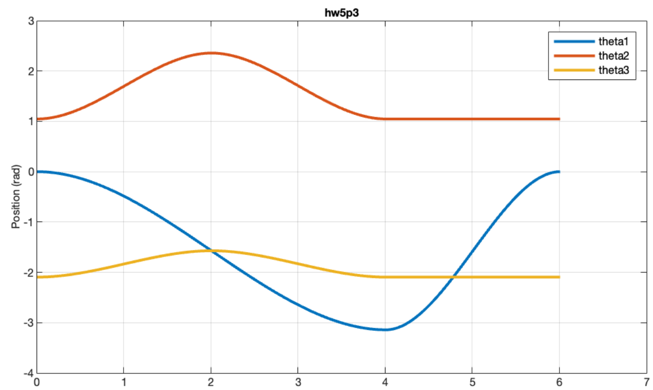
Relevant code:

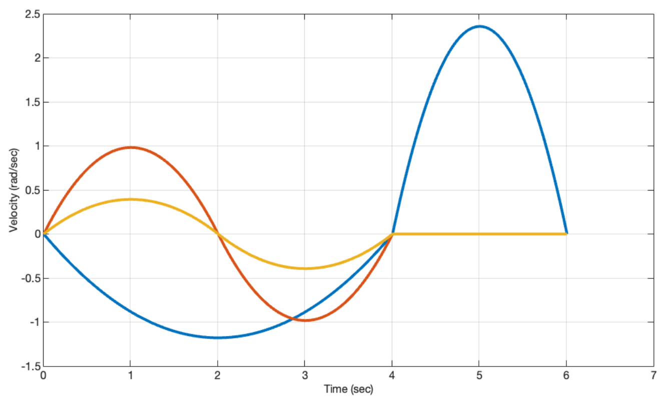
```
#  Newton Raphson
def newton_raphson(xgoal):
    # Collect the distance to goal and change in q every step!
    xdistance = []
    qstepsize = []

    # Set the initial joint value guess.
    q = np.array([0.0, np.pi/2, -np.pi/2]).reshape(3,1)
    min_error = 10e-12
    steps = 0
    converged = False

    # IMPLEMENT THE NEWTON-RAPHSON ALGORITHM!
    for i in range(0, 20):
        x_diff = xgoal - fkin(q)
        dist_error = np.linalg.norm(x_diff)
        xdistance.append(dist_error)
        q_next = q + np.matmul(np.linalg.inv(Jac(q)), x_diff)
        qstepsize.append(np.linalg.norm(q_next - q))
        if dist_error < min_error:
            converged = True
            print("Converged for {}".format(xgoal))
            print("Steps required: {}".format(steps))
            print("Final q = {}".format(q))
            elbow = "elbow down"
            side = "back side"
            if elbow_up(q):
                elbow = "elbow up"
            if front_side(q):
                side = "front side"
            print("Corresponds to an {} and {} solution".format(elbow, side))
            print("Wraps by 360 deg: {}".format(wraps(q)))
            print("-"*50)
            break
        q = q_next
        steps += 1

    if not converged:
        print("Did not converge for {}".format(xgoal))
        print("-" * 50)

    # Create a plot of x distances to goal and q step sizes, for N steps.
    N = 20
    xdistance = xdistance[:N+1]
    qstepsize = qstepsize[:N+1]

    fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)

    ax1.plot(range(len(xdistance)), xdistance)
    ax2.plot(range(len(qstepsize)), qstepsize)
```

```
ax1.set_title(f'Convergence Data for {xgoal.T}')
ax2.set_xlabel('Iteration')

ax1.set_ylabel('Task Distance to Goal')
ax1.set_ylim([0, max(xdistance)])
ax1.set_xlim([0, N])
ax1.set_xticks(range(N+1))
ax1.grid()

ax2.set_ylabel('Joint Step Size')
ax2.set_ylim([0, max(qstepsize)])
ax2.set_xlim([0, N])
ax2.set_xticks(range(N+1))
ax2.grid()

plt.show()
```

Problem 3 (3 DOF Joint Movement - NumPy/Spline/ROS Warm-up) - 14 points:

Plots (note that theta1 in the plot refers to the pan angle):





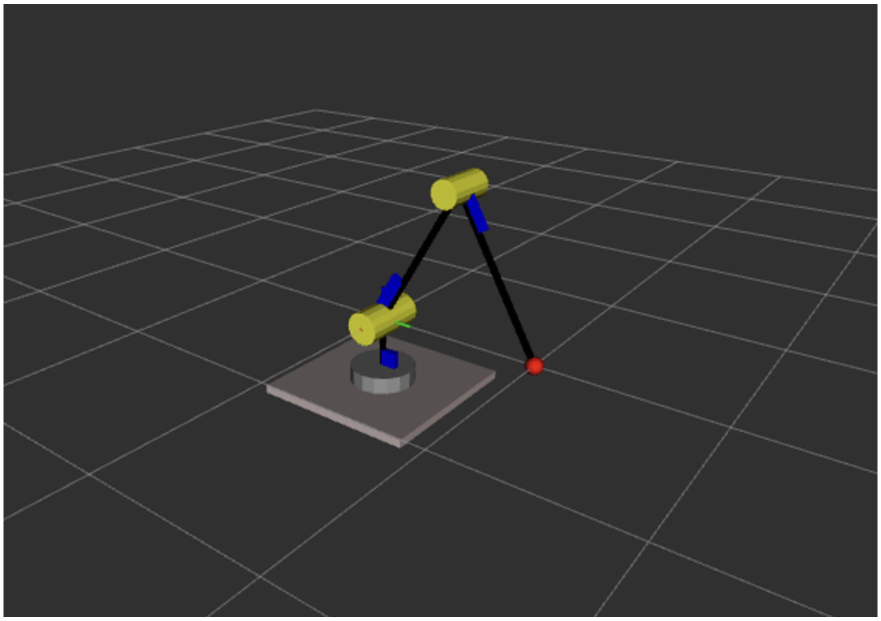
Using the goto function, we have that the velocity at q_B is $\begin{bmatrix} -1.17809725 \\ 0 \\ 0 \end{bmatrix}$ More specifically this is the output (velocity) from goto(2.0, 4.0, self.qA, self.qC).

Code for problem 3:

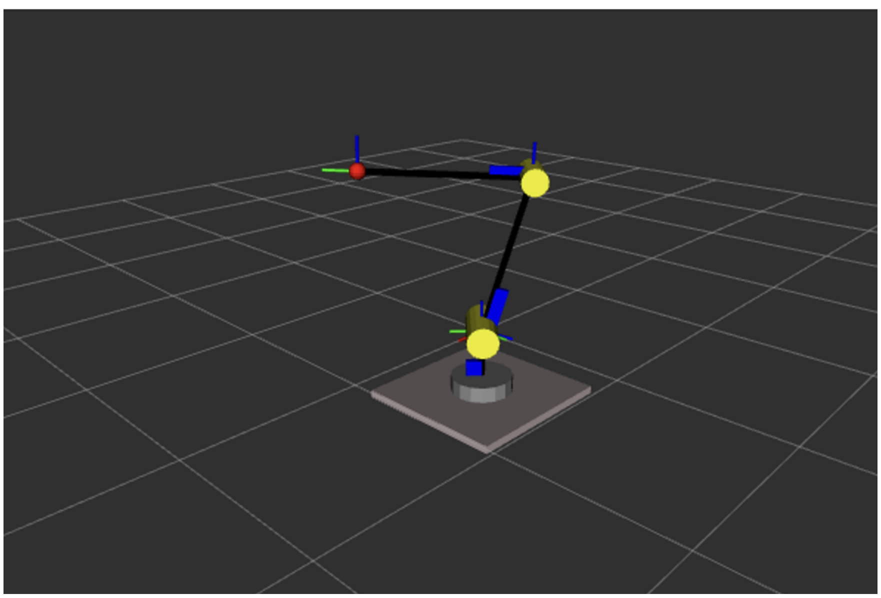
```
class Trajectory():
    # Initialization.
    def __init__(self, node):
        # Define the three joint positions.
        self.qA = np.radians(np.array([   0,  60, -120])).reshape(3,1)
        self.qB = np.radians(np.array([ -90, 135,  -90])).reshape(3,1)
        self.qC = np.radians(np.array([-180,  60, -120])).reshape(3,1)

        # get the velocity at qB
        (q, qdot) = goto(2.0, 4.0, self.qA, self.qC)
        self.omega_B = qdot

        # zero vector
        self.zero_vec = np.array([0, 0, 0]).reshape(3,1)

    # Declare the joint names.
    def jointnames(self):
        # Return a list of joint names
        return ['theta1', 'theta2', 'theta3']

    # Evaluate at the given time.
    def evaluate(self, t, dt):
        # stop after first cycle
        # uncomment to keep going indefinitely
        if (t > 6.0):
            return None

        # First modulo the time by 4 seconds
        t = fmod(t, 6.0)

        # Compute the joint values.
        if (t < 2.0):
            (q, qdot) = spline(t, 2.0, self.qA, self.qB,self.zero_vec,self.omega_B)
        elif(t < 4.0) :
            (q, qdot) = spline(t-2.0, 2.0, self.qB, self.qC,self.omega_B,self.zero_vec)
        else:
            (q, qdot) = spline(t-4.0, 2.0, self.qC, self.qA,self.zero_vec,self.zero_vec)

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

Problem 4 (3 DOF Tip Movement - Inverse Jacobian) - 24 points:

Robot at point A:

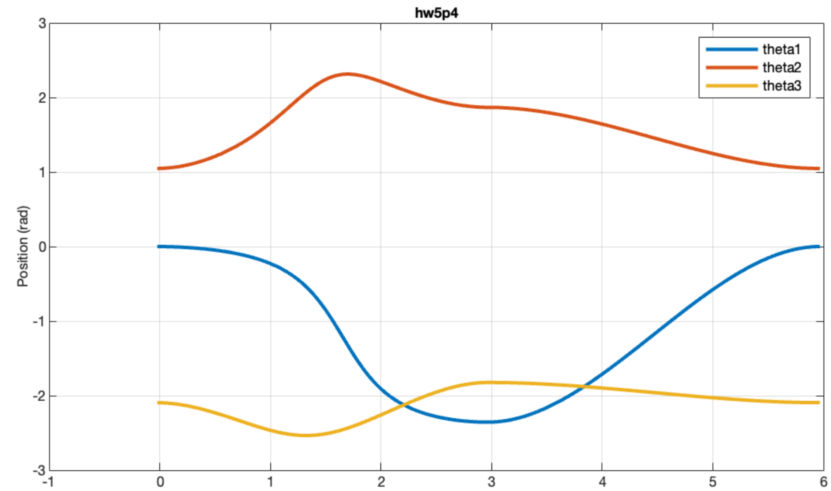


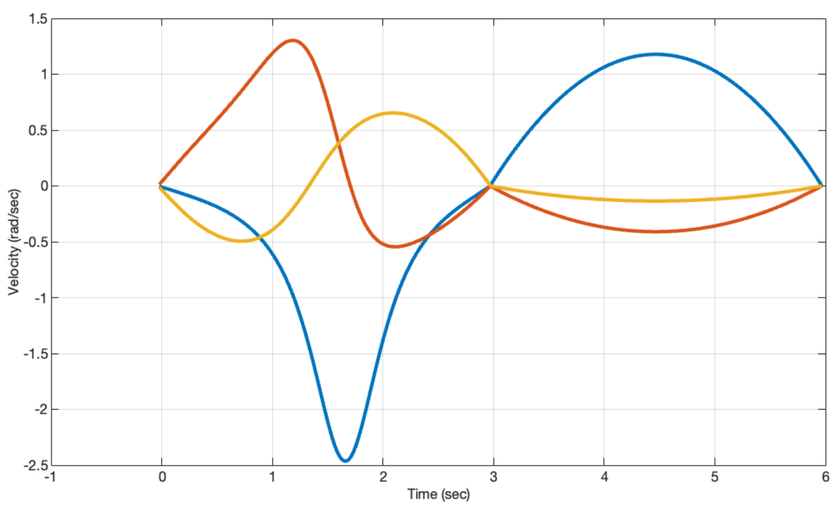
Robot at point D:



Numerically we have $q_D \approx \begin{bmatrix} -2.35603426 \\ 1.86729854 \\ -1.82381187 \end{bmatrix}$

Plots for problem 4:





Code for problem 4:

```
class Trajectory():
    # Initialization.
    def __init__(self, node):
        # Define the known tip/joint positions.
        self.qA = np.radians(np.array([ 0, 60, -120])).reshape(3,1)
        self.xA = fkin(self.qA)

        self.qD = None
        self.xD = np.array([0.5, -0.5, 1.0]).reshape(3,1)

        # Select the leg duration.
        self.T = 3.0

        # Initialize the parameters and anything stored between cycles!
        self.l = 20 #lambda
        self.q_prev = self.qA
        self.zero_vec = np.array([0, 0, 0]).reshape(3,1)

    # Declare the joint names.
    def jointnames(self):
        # Return a list of joint names
        return ['theta1', 'theta2', 'theta3']
```

Code for problem 4 (continued):

```
# Evaluate at the given time.
def evaluate(self, t, dt):
    # End after one cycle.
    # uncomment to keep going indefinitely
    if (t > 2*self.T):
        return None

    # First modulo the time by 2 legs.
    t = fmod(t, 2*self.T)

    # reset after every cycle
    if (t < 0.01):
        print('reset')
        self.q_prev = self.qA
        q = self.q_prev
        qdot = self.zero_vec
        return (q.flatten().tolist(), qdot.flatten().tolist())

    # COMPUTE THE MOTION.
    # from A to D
    if (t < 3.0):
        # desired position
        xd_prev, vd_prev = spline(t-dt, self.T, self.xA, self.xD, self.zero_vec, self.zero_vec)
        xd, vd = spline(t, self.T, self.xA, self.xD, self.zero_vec, self.zero_vec)
        error = xd_prev - fkin(self.q_prev)
        xr_dot = vd + self.l * error
        qdot = np.matmul(np.linalg.inv(Jac(self.q_prev)), xr_dot)
        q = self.q_prev + dt * qdot
        self.q_prev = q
        print(fkin(q).flatten().tolist())

    # from D to A
    else:
        # self.q_prev is the joint config for point D
        (q, qdot) = spline(t-3.0, self.T, self.q_prev, self.qA, self.zero_vec, self.zero_vec)
        return (q.flatten().tolist(), qdot.flatten().tolist())

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

Problem 5 (General Kinematic Chain Calculations) - 24 points:

Output from code (all test cases passed):

```
q:
 [[ 0.349]
 [ 0.698]
 [-0.524]]
ptip(q):
 [[-0.599]
 [ 1.645]
 [ 0.816]]
Rtip(q):
 [[ 0.94  -0.337  0.059]
 [ 0.342  0.925 -0.163]
 [ 0.     0.174  0.985]]
Jv(q):
 [[-1.645  0.279  0.059]
 [-0.599 -0.767 -0.163]
 [ 0.     1.751  0.985]]
Jw(q):
 [[0.    0.94  0.94 ]
 [0.    0.342 0.342]
 [1.    0.    0.   ]]
```

```
q:
 [[0.524]
 [0.524]
 [1.047]]
ptip(q):
 [[-0.433]
 [ 0.75 ]
 [ 1.5  ]]
Rtip(q):
 [[ 0.866 -0.     0.5  ]
 [ 0.5    0.    -0.866]
 [ 0.     1.     0.   ]]
Jv(q):
 [[-0.75   0.75   0.5  ]
 [-0.433 -1.299 -0.866]
 [ 0.     0.866  0.   ]]
Jw(q):
 [[0.    0.866 0.866]
 [0.    0.5   0.5  ]
 [1.    0.    0.   ]]
```

```
q:
 [[-0.785]
 [ 1.309]
 [ 2.094]]
ptip(q):
 [[-0.5  ]
```

```
 [-0.5   ]
 [ 0.707]]
Rtip(q):
 [[ 0.707 -0.683  0.183]
 [-0.707 -0.683  0.183]
 [ 0.    -0.259 -0.966]]
Jv(q):
 [[ 0.5   -0.5    0.183]
 [-0.5   -0.5    0.183]
 [ 0.    -0.707 -0.966]]
Jw(q):
 [[ 0.     0.707  0.707]
 [ 0.    -0.707 -0.707]
 [ 1.     0.     0.   ]]
```

Code:

```
def fkin(self, q):
# Check the number of joints
if (len(q) != self.dofs):
    self.error("Number of joint angles (%d) does not chain (%d)",
               len(q), self.dofs)

# Clear any data from past invocations (just to be safe).
for s in self.steps:
    s.clear()

# Initialize the T matrix to walk up the chain, w.r.t. world frame!
T = np.eye(4)

# Walk the chain, one step at a time.  Record the T transform
# w.r.t. world for each step.
for s in self.steps:
    #  s.Tshift     Transform w.r.t. the previous frame
    #  s.elocal     Joint axis in the local frame
    #  s.dof        Joint number
    #  q[s.dof]     Joint position (angle for revolute, displacement for linear)

    # Take action based on the joint type.
    if s.type is Joint.REVOLUTE:
        # first do the fixed shift, then do rotation
        T = np.matmul(T, s.Tshift)
        R = Rote(s.elocal, q[s.dof])
        T = np.matmul(T, T_from_Rp(R, pzero()))

    elif s.type is Joint.LINEAR:
        # first do the fixed shift, then do translation
        T = np.matmul(T, s.Tshift)
        p = s.elocal * q[s.dof]
        T = np.matmul(T, T_from_Rp(Reye(), p))
    else:
```

```
        T = np.matmul(T, s.Tshift)

    # Store the info (w.r.t. world frame) into the step.
    s.T = T
    s.p = p_from_T(T)
    s.R = R_from_T(T)
    s.e = R_from_T(T) @ s.elocal

# Collect the tip information w.r.t. world!
ptip = p_from_T(T)
Rtip = R_from_T(T)

# Re-walk up the chain to fill in the Jacobians.
Jv = np.zeros((3,self.dofs))
Jw = np.zeros((3,self.dofs))
for s in self.steps:
    #  s.p      Position w.r.t. world
    #  s.e      Joint axis w.r.t. world

    # Take action based on the joint type.
    if s.type is Joint.REVOLUTE:
        # Revolute is a rotation:
        Jv[:,s.dof:s.dof+1] = cross(s.e, (ptip - s.p))
        Jw[:,s.dof:s.dof+1] = s.e
    elif s.type is Joint.LINEAR:
        # Linear is a translation:
        Jv[:,s.dof:s.dof+1] = s.e
        Jw[:,s.dof:s.dof+1] = pzero()

# Return the info
return (ptip, Rtip, Jv, Jw)
```

Problem 6 (Time Spent) - 4 points:

I spent about 6.5 hours on this problem set. I did not have any particular bottlenecks.