Problem 1a (Geometric Jacobian)

The geometric Jacobian $J$ of a open-chain manipulator with $n$ revolute joints is a $(6 \times n)$ matrix, which in general is a function of the joint variables. It describes the relationship between the joint velocities $\dot{\theta}$ and the end-effector linear velocity $\dot{x}$ as well as angular velocity $\dot{\omega}$ according to

$$
\begin{bmatrix}
\dot{x} \\
\dot{\omega}
\end{bmatrix} = J(\theta) \dot{\theta}.
$$

(1)

The geometric Jacobian is given by

$$
J = 
\begin{bmatrix}
J_P \\
J_O
\end{bmatrix},
$$

(2)

where $J_P$ is the $(3 \times n)$ matrix relative to the contribution of the joint velocities $\dot{\theta}$ to the end-effector linear velocity $\dot{x}$ and $J_O$ is the $(3 \times n)$ matrix relative to the contribution of the joint velocities $\dot{\theta}$ to the end-effector angular velocity $\dot{\omega}$. The Jacobian $J$ in Eq. (2) can be partitioned into $(3 \times 1)$ column vectors according to

$$
\begin{bmatrix}
J_P \\
J_O
\end{bmatrix} =
\begin{bmatrix}
j_{P1} & \ldots & j_{Pn} \\
j_{O1} & \ldots & j_{On}
\end{bmatrix}.
$$

(3)

The term $\dot{\theta}_i j_{Pi}$ represents the contribution of single joint $i$ to the end-effector linear velocity, while the term $\dot{\theta}_i j_{Oi}$ represents the contribution of single joint $i$ to the end-effector angular velocity. In case of a open-chain manipulator with revolute joints these column vectors are given by

$$
\begin{bmatrix}
j_{P1} \\
j_{O1}
\end{bmatrix} =
\begin{bmatrix}
z_{i-1} \times (p_n - p_{i-1}) \\
z_{i-1}
\end{bmatrix},
$$

(4)

where $z_{i-1}$ is the unit vector of joint $i$ axis, $p_n$ the position of the end-effector, and $p_{i-1}$ the position of joint $i$. Thus, the generic formula for a geometric Jacobian of a system with $n$ revolute joints that only deals with postions is given by

$$
J =
\begin{bmatrix}
z_0 \times (p_n - p_0) & z_1 \times (p_n - p_1) & \ldots & z_{n-1} \times (p_n - p_{n-1})
\end{bmatrix}.
$$

(5)

The joint axis $z_i$ can be replaced by $\left[ \begin{array}{c} 0 \\ 0 \\ 1 \end{array} \right]^T$ since we consider a planar manipulator in which all axis are parallel to $z_0$. 


Problem 1b (Jacobian computation)

To eliminate the $p_i$ variables in Eq. (5) and make the Jacobian a function of only the joint angles $\theta_i$ and the link length $l_i$ we use the homogeneous transformation matrix

$$ A_i^{-1}(\theta_i) = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & l_i \cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) & 0 & l_i \sin(\theta_i) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . $$  

The homogeneous transformation matrix $A_i^{-1}(\theta_i)$ allows to compute the position $p_i$ of joint $i + 1$ from the position $p_{i-1}$ of joint $i$ and the joint angle $\theta_i$ at this joint. For example, the position $p_1$ of joint 2 is computed according to

$$ p_1 = A_0^1(\theta_1) p_0 . $$  

Moreover, the position $p_i$ of joint $i + 1$ is computed according to

$$ p_i = A_0^1(\theta_1) A_1^2(\theta_2) \cdots A_{i-1}^i(\theta_i) p_0 . $$  

Finally, the position of the end-effector $p$ is computed according to

$$ p_n = T(\theta)^0_n p_0 = A_0^1(\theta_1) A_1^2(\theta_2) \cdots A_{n-1}^n(\theta_n) p_0 . $$  

To obtain the Jacobian which is only depending on the joint angles $\theta_i$ and the link lengths $l_i$ we substitute the $p_i$ and $p_n$ in Eq. (5) with the $p_i$ from Eq. (8) and the $p_n$ from Eq. (9), respectively.

Problem 1c (Jacobian computation)

The Jacobian $J$ for the planar manipulator with $n = 4$ degrees of freedom is computed in MATLAB as follows:

```matlab
% NOTE: insert your Jacobian calculation here
p0 = [0; 0];
p1 = forward_kinematics([theta(1)', links(1)']);
p2 = forward_kinematics([theta(1:2)', links(1:2)']);
p3 = forward_kinematics([theta(1:3)', links(1:3)']);
p4 = forward_kinematics([theta(1:4)', links(1:4)']);

J = [ -p4(2)+p0(2) -p4(2)+p1(2) -p4(2)+p2(2) -p4(2)+p3(2); ...
      p4(1)-p0(1) p4(1)-p1(1) p4(1)-p2(1) p4(1)-p3(1) ];
```
Problem 1d (Jacobian transpose method)

The general formula of the Jacobian transpose for inverse kinematics computations is given by

$$
\Delta \theta = J^T(\theta) K \Delta x .
$$

(10)

where $K$ is a symmetric positive definite matrix, which determines the convergence rate of the tracking error $\Delta x = x_{desired} - x_{actual}$. The matrix $K$ is usually a diagonal matrix. If all values on the diagonal are the same, the error in each task space dimension ($x, y$) are treated the same, and the matrix can be replaced by a scalar $\alpha$ resulting in

$$
\Delta \theta = \alpha J^T(\theta) \Delta x .
$$

(11)

The Jacobian transpose method is computed in MATLAB as follows.

```matlab
% NOTE: insert the required inverse kinematics methods at this location
alpha = 4.0;
theta_d = alpha * J' * xd;
```

The obtained task space tracking performance and the corresponding joint space trajectories are given below.

The method does not perform very well as it does not use the inverse of the Jacobian, but its transpose. However, the use of the method is justified in terms of virtual forces. Fig. 1(right) shows clearly that the method is non conservative.

---

Figure 1: Tracking performance of the Jacobian transpose method (left) and corresponding joint trajectories (right).

The method does not perform very well as it does not use the inverse of the Jacobian, but its transpose. However, the use of the method is justified in terms of virtual forces. Fig. 1(right) shows clearly that the method is non conservative.
Better tracking performance is achieved by choosing the value for $\alpha$ at each time step according to

$$\alpha = \frac{\langle \Delta x, JJ^T \Delta x \rangle}{\langle JJ^T \Delta x, JJ^T \Delta x \rangle}.$$  \hfill (12)

See http://math.ucsd.edu/~sbuss/ResearchWeb/ikmethods/iksurvey.pdf for more information. The corresponding MATLAB code, the resulting task space tracking performance and the corresponding joint space trajectories are given below.

% NOTE: insert the required inverse kinematics methods at this location
alpha = dot(xd, (J*J'*xd)) / dot((J*J'*xd), (J*J'*xd));
theta_d = alpha * J' * xd;

Figure 2: Tracking performance of the Jacobian transpose method with variable alpha (left) and corresponding joint trajectories (right).
Problem 1e (Pseudo-inverse method)

The general formula of the pseudo-inverse for inverse kinematics computations is

$$\Delta \theta = J^\dagger(\theta) \Delta x,$$

where

$$J^\dagger = J^T(JJ^T)^{-1}.$$  \hspace{1cm} (14)

The matrix \((JJ^T)\) may not be invertible due to singularities. Thus, a small number \((10^{-10})\) is added to its diagonal to ensure that the matrix remains full rank. This small number can be seen as a damping term as it prevents the inverse to have huge values which cause huge joint velocities. The corresponding MATLAB code and the resulting tracking performance as well as the joint trajectories are provided below.

```
% NOTE: insert the required inverse kinematics methods at this location
dampingTerm = 10^(-10);
pseudoInv = J' * inv(J * J' + (dampingTerm * eye(2)));
theta_d = pseudoInv * xd;
```

![Tracking performance using the pseudo-inverse method](image)

Figure 3: Tracking performance of the pseudo-inverse method (left) and corresponding joint trajectories (right).

The pseudo-inverse method performs pretty well as shown in Fig. 3(left). We can also see that the pseudo inverse method is not exactly non conservative, however, the joint trajectories shown in Fig. 3(right) indicate that the joint configuration at the beginning of each cycle do almost match.
Problem 1f (Pseudo-inverse with Null-space optimization method)

The general formula of the pseudo-inverse with Null-space optimization for inverse kinematics computations is

\[
\Delta \theta = J^\dagger(\theta) \Delta x + (I - J^\dagger J)(\theta_0 - \theta),
\]

where the pseudo inverse \( J^\dagger \) is computed as in Eq. (14), \( \theta \) is the actual joint configuration and \( \theta_0 \) is the desired posture for optimization. The same damping term as in Problem e) is used. The corresponding MATLAB code and the resulting tracking performance as well as the joint trajectories are provided below.

% NOTE: insert the required inverse kinematics methods at this location
dampingTerm = 10^(-10);
pseudoInv = J’ * inv(J * J’ + (dampingTerm * eye(2)));
theta0 = [0.0; 0.0; 0.0; 0.0];
theta_d = pseudoInv * xd + (eye(4) - pseudoInv * J) * (theta0-theta);

The pseudo-inverse with Null-space optimization method also performs pretty well as shown in Fig. 4(left). We can also see that the pseudo inverse method is non conservative. Also, the joint trajectories differ from Problem e) as a result of the Null-space optimization which pulls the joints to the zero position in the Null-space. We can see in Fig. 4(right) that especially joint \( \theta_3 \) and \( \theta_4 \) are closer to zero than in Fig. 3(right).
Problem 1g (Weighted pseudo-inverse with (unweighted) Null-space optimization method)

The weighted pseudo-inverse with (unweighted) Null-space optimization method is given by

$$ J^\dagger = W J^T (J W J^T)^{-1} , $$

where $W$ is a diagonal matrix with $W_{ii} = w_i$, for $i = 1, \ldots, 4$, and $w = [0.01, 0.1, 0.5, 1.0]^T$. The joint velocities are given by

$$ \Delta \theta = J^\dagger(\theta) \Delta x + (I - J^\dagger J)(\theta_0 - \theta) , $$

where $J^\dagger$ is the (unweighted) pseudo-inverse from Eq. (14). The corresponding MATLAB code and resulting tracking performance as well as corresponding joint trajectories are given below.

```matlab
% NOTE: insert the required inverse kinematics methods at this location

% dampingTerm = 10^(-10);  % damping term
w = [0.01; 0.1; 0.5; 1.0];  % weights
W = diag(w);  % diagonal matrix

% weighted PseudoInverse
weightedPseudoInv = W * J' * inv(J * W * J' + (dampingTerm * eye(2)));

% PseudoInverse
pseudoInv = J' * inv(J * J' + (dampingTerm * eye(2)));

% Initial joint positions
theta0 = [0.0; 0.0; 0.0; 0.0];

% Desired and actual joint positions
thetad = weightedPseudoInv * xd + (eye(4)-pseudoInv * J) * (theta0-theta);
```

Tracking performance using the weighted pseudo-inverse with (unweighted) null-space optimization method and corresponding joint trajectories are given below.

Figure 5: Tracking performance of the weighted pseudo-inverse method (left) and corresponding joint trajectories (right).

The weighted pseudo-inverse method performs well. The joint trajectories differ from those in Problem e) due to the weighting of the pseudo-inverse (see Problem i) for further explanations). The (unweighted) Null-space optimization term pulls each joint equally to its zero position whenever it does not effect the end-effector position, similar to Problem 1f).
**Problem 1h (Weighted pseudo-inverse method with Null-space optimization method)**

The weighted pseudo-inverse with Null-space optimization is given by

\[
\Delta \theta = J^\dagger(\theta) \Delta x + \left( I - J^\dagger J \right) (\theta_0 - \theta)
\]

where \(J^\dagger\) is computed using Eq. (16).

```matlab
% NOTE: insert the required inverse kinematics methods at this location
dampingTerm = 10^(-6);
w = [0.01; 0.1; 0.5; 1.0];
W = diag(w);

weightedPseudoInv = W * J' * inv(J * W * J' + (dampingTerm * eye(2)));
theta0 = [0.0; 0.0; 0.0; 0.0];

thetad = weightedPseudoInv * xd + (eye(4) - weightedPseudoInv * J) * (theta0-theta);
```

Figure 6: Tracking performance of the weighted pseudo-inverse with Null-space optimization method (left) and corresponding joint trajectories (right).

The weighted pseudo-inverse with Null-space optimization method performs well, except of a certain part of the trajectory as shown in Fig. 6(left). The reason for that is that joint \(\theta_1\) is weighted with 0.01 and therefore does not contribute much to the solution. As a result, the arm does not rotate around the origin \(p_0\). Thus, for a short moment, the desired velocity \(\Delta x\) is not achievable due to a singular configuration. The reason why this singular configuration did not occur in Problem 1g) is because the Null-space optimization term used there is unweighted. Thus, the arm stretches in the Null-space of the end-effector (due to the choice of the default posture) causing joint \(\theta_1\) to rotate around the origin \(p_0\).
Problem 1i (Pseudo-inverse solution for inverse kinematics with changed differential kinematics)

Let \( \tilde{\theta} \) be the scaled joint position, \( \dot{\tilde{\theta}} \) the scaled joint velocities and both vectors are scaled by the same diagonal matrix \( S \), i.e. \( \tilde{\theta} = S\theta \), and \( \dot{\tilde{\theta}} = S\dot{\theta} \). The pseudo-inverse with this changed differential kinematics is obtained by replacing the Jacobian \( J \) in Eq. (14) with \( JS \), which leads to

\[
J^\dagger = (JS)(JSJ^T)^{-1}
\]

The solution for inverse kinematics is then obtained by

\[
\dot{\hat{\theta}} = J^\dagger(\theta) \dot{x} .
\]

To compute the scaled joint velocities \( \tilde{\theta} \), Eq. (19) is multiplied by \( S \) from the left leading to

\[
\dot{\tilde{\theta}} = S\dot{\hat{\theta}} = SSJ^\dagger(\theta) \dot{x}
\]

\[
= SSJ^T(JSSJ^T)^{-1} \dot{x}
\]

\[
= WJ^T(JWJ^T)^{-1} \dot{x}
\]

where \( W = SS \). Thus, we obtain a weighted pseudo-inverse method as in Problem g). This weighted pseudo-inverse computes the solutions \( \dot{\tilde{\theta}} \) that satisfy the linear equation \( \dot{x} = J\dot{\tilde{\theta}} \) and minimize the quadratic cost function of joint velocities

\[
g(\dot{\tilde{\theta}}) = \frac{1}{2}\dot{\tilde{\theta}}^T W^{-1} \dot{\tilde{\theta}} .
\]

If \( W \) is the identity matrix \( I \) the solution simplifies to the pseudo-inverse method described in Problem e) which minimizes the cost function

\[
g(\dot{\tilde{\theta}}) = \frac{1}{2}\dot{\tilde{\theta}}^T \dot{\tilde{\theta}} .
\]

If \( W \) is a scaled identity matrix \( \beta I \) (\( \beta \in \mathbb{R} \)), the result would also not differ from the one in Problem e) since minimizing the cost function in Eq. (22) does not differ from minimizing

\[
g(\dot{\tilde{\theta}}) = \frac{1}{2}\dot{\theta}^T (\beta I)^{-1} \dot{\theta} .
\]

However, if \( W \) has different values on the diagonal the result would be different as the obtained solution would minimize a different cost function. Thus, the scaling of joint position and velocities is important for inverse kinematics if they are scaled differently. The same property holds for a system which is perfectly calibrated, but uses different units. Assuming that \( \tilde{\theta} \) is given in radians and \( \theta \) is given in degrees, the scaling matrix \( S \) becomes \( \beta I \), where \( \beta = \frac{\pi}{180} \). Such a scaling would not affect the solution. However, if the system would also contain prismatic joints which do not undergo the same scaling the result would differ.