

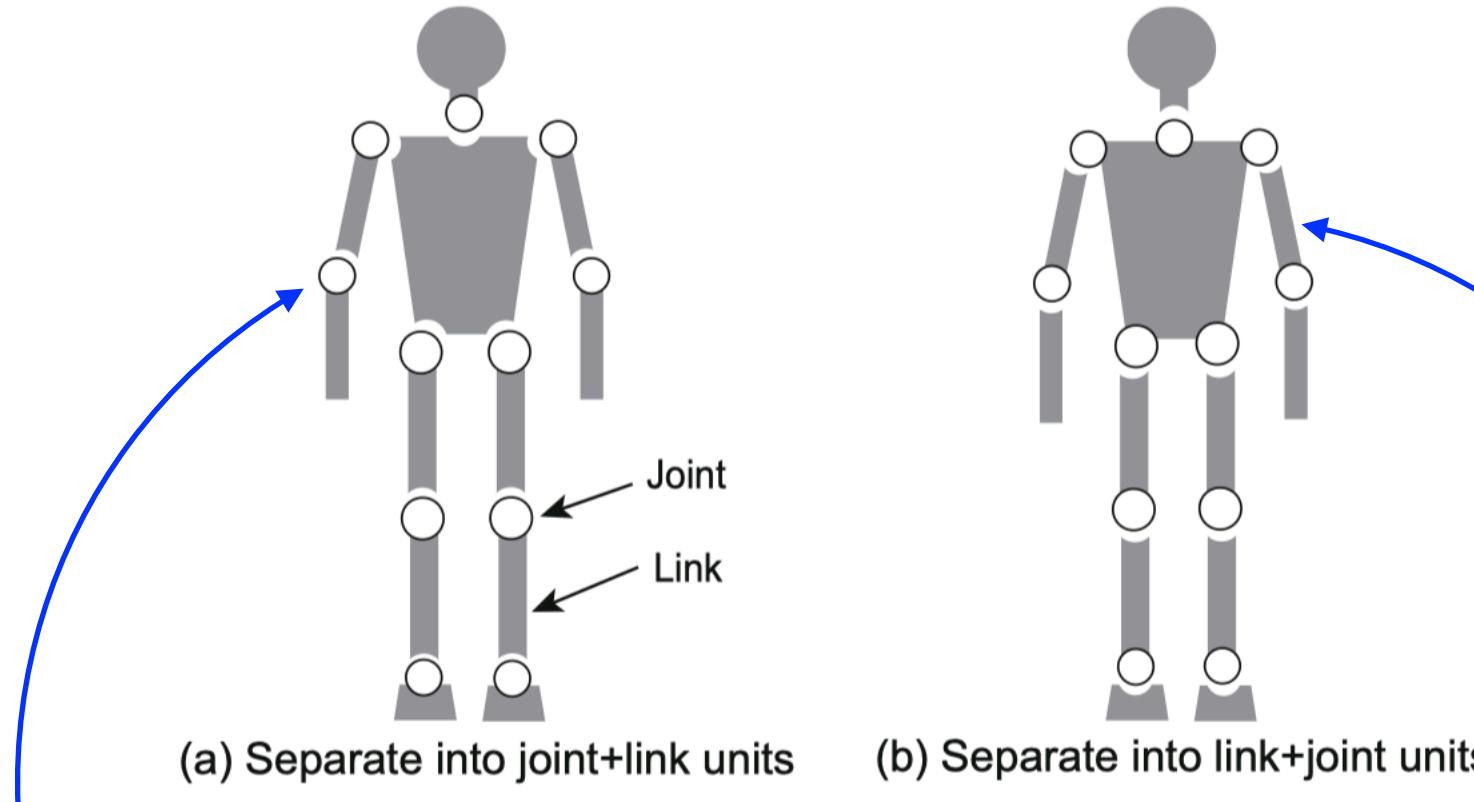
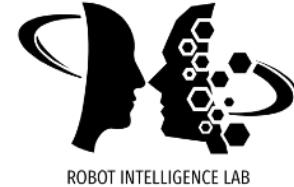


Intelligent Robotics

Kinematics

Sungjoon Choi, Korea University

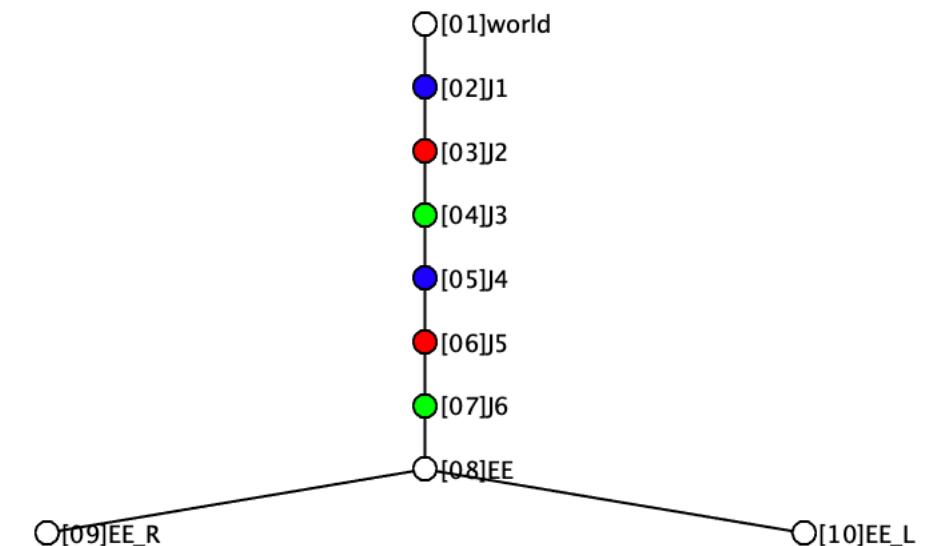
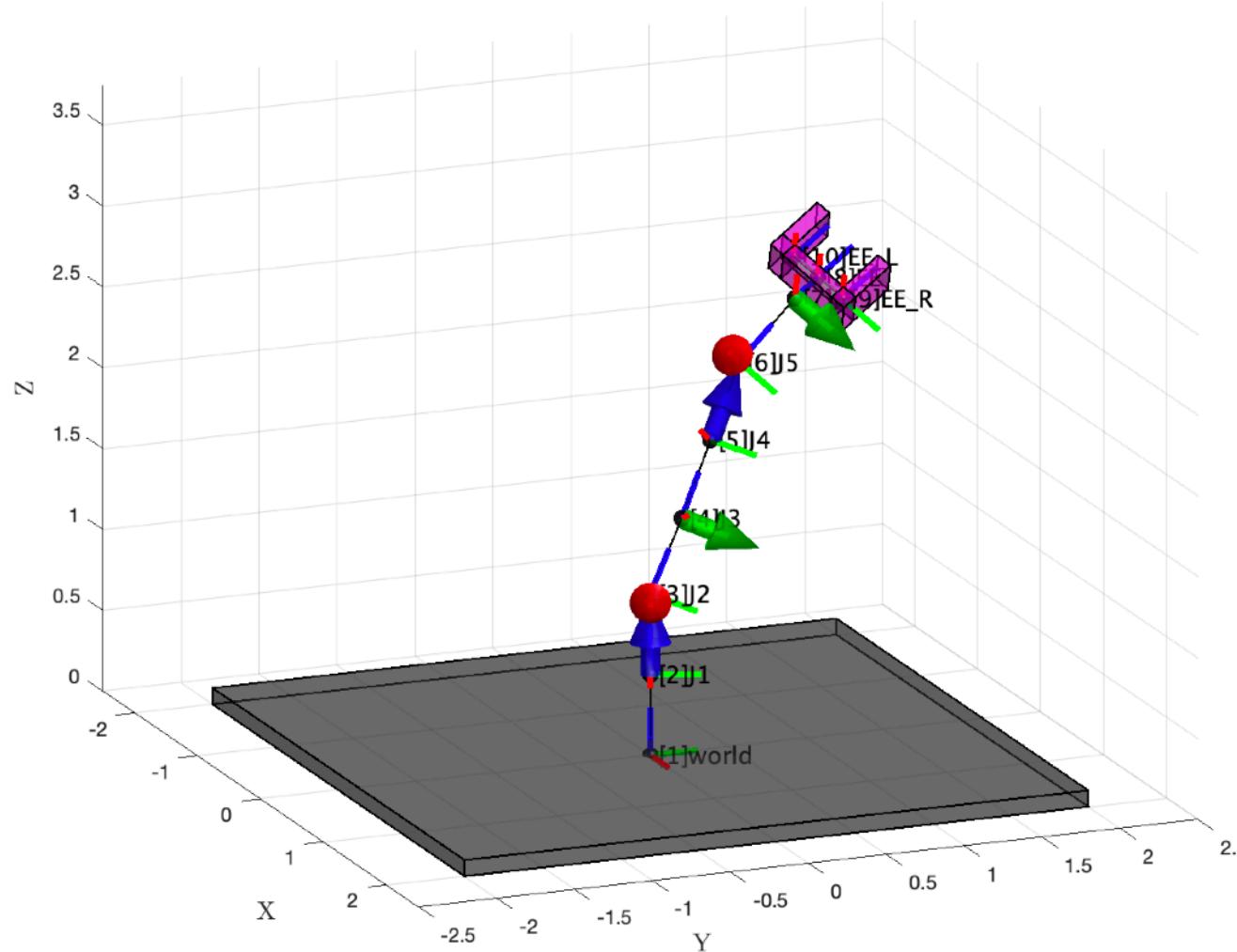
Kinematic Chain



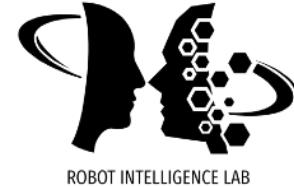
One **joint** contains a **link**, and it makes programming more convenient as kinematic chains are defined w.r.t **joints**. However, it cannot handle **joints** with multiple **links**.

Multiple **joints** are attached to a single **link**. This becomes problematic when we often have kinematic chains without **links**.

Kinematic Chain



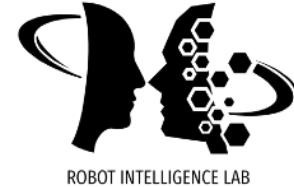
Kinematic Chain



struct with fields:

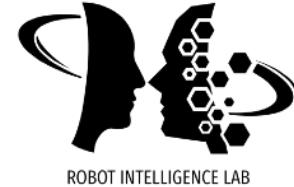
```
    name: 'kinematic_chain'  
        dt: 0.0100  
        joint: [1x10 struct]  
    joint_names: {'world'  'J1'  'J2'  'J3'  'J4'  'J5'  'J6'  'EE'  'EE_R'  'EE_L'}  
        n_joint: 10  
rev_joint_names: {'J1'  'J2'  'J3'  'J4'  'J5'  'J6'}  
        n_rev_joint: 6  
            link: [1x4 struct]  
link_names: {'base_link'  'EE_link'  'EE_R_link'  'EE_L_link'}  
        n_link: 4
```

Kinematic Chain (Joint)



```
>> chain.joint  
  
ans =  
  
    1x10 struct array with fields:  
  
        name  
        p  
        R  
        a  
        type  
        p_offset  
        R_offset  
        q  
        dq  
        ddq  
        q_diff  
        q_prev  
        v  
        vo  
        w  
        dvo  
        dw  
        u  
        ext_f  
        parent  
        childs  
        link_idx  
        limit
```

Kinematic Chain (Joint)



```
>> chain.joint(1)
```

```
ans =
```

struct with fields:

```
    name: 'world'  
    p: [3x1 double]  
    R: [3x3 double]  
    a: [3x1 double]  
    type: 'fixed'  
    p_offset: [3x1 double]  
    R_offset: [3x3 double]  
    q: 0  
    dq: 0  
    ddq: 0  
    q_diff: 0  
    q_prev: 0  
    v: [3x1 double]  
    vo: [3x1 double]  
    w: [3x1 double]  
    dvo: [3x1 double]  
    dw: [3x1 double]  
    u: 0  
    ext_f: [3x1 double]  
    parent: []  
    child: 2  
    link_idx: 1  
    limit: [-Inf Inf]
```

```
>> chain.joint(2)
```

```
ans =
```

struct with fields:

```
    name: 'J1'  
    p: [3x1 double]  
    R: [3x3 double]  
    a: [3x1 double]  
    type: 'revolute'  
    p_offset: [3x1 double]  
    R_offset: [3x3 double]  
    q: -1.3464  
    dq: 0  
    ddq: 0  
    q_diff: 0.0139  
    q_prev: -1.3464  
    v: [3x1 double]  
    vo: [3x1 double]  
    w: [3x1 double]  
    dvo: [3x1 double]  
    dw: [3x1 double]  
    u: 0  
    ext_f: [3x1 double]  
    parent: 1  
    child: 3  
    link_idx: []  
    limit: [-Inf Inf]
```

```
>> chain.joint(3)
```

```
ans =
```

struct with fields:

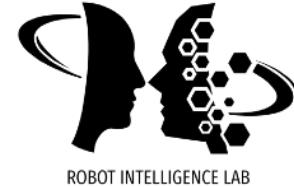
```
    name: 'J2'  
    p: [3x1 double]  
    R: [3x3 double]  
    a: [3x1 double]  
    type: 'revolute'  
    p_offset: [3x1 double]  
    R_offset: [3x3 double]  
    q: -1.3464  
    dq: 0  
    ddq: 0  
    q_diff: 0.0139  
    q_prev: -1.3464  
    v: [3x1 double]  
    vo: [3x1 double]  
    w: [3x1 double]  
    dvo: [3x1 double]  
    dw: [3x1 double]  
    u: 0  
    ext_f: [3x1 double]  
    parent: 2  
    child: 4  
    link_idx: []  
    limit: [-Inf Inf]
```

Kinematic Chain (Link)



```
>> chain.link  
  
ans =  
  
1x4 struct array with fields:  
  
    name  
    joint_idx  
    p_offset  
    R_offset  
    mesh_path  
    scale  
    fv  
    box  
    capsule  
    box_added  
    m  
    I_bar  
    com_bar  
    v  
    vo  
    w
```

Kinematic Chain (Link)



```
>> chain.link(1)

ans =

struct with fields:

    name: 'base_link'
    joint_idx: 1
    p_offset: [3x1 double]
    R_offset: [3x3 double]
    mesh_path: ''
    scale: [3x1 double]
    fv: ''
    box: ''
    capsule: ''
    box_added: [1x1 struct]
    m: 0
    I_bar: [3x3 double]
    com_bar: [3x1 double]
    v: [3x1 double]
    vo: [3x1 double]
    w: [3x1 double]
```

```
>> chain.link(2)

ans =

struct with fields:

    name: 'EE_link'
    joint_idx: 8
    p_offset: [3x1 double]
    R_offset: [3x3 double]
    mesh_path: ''
    scale: [3x1 double]
    fv: ''
    box: ''
    capsule: ''
    box_added: [1x1 struct]
    m: 0
    I_bar: [3x3 double]
    com_bar: [3x1 double]
    v: [3x1 double]
    vo: [3x1 double]
    w: [3x1 double]
```

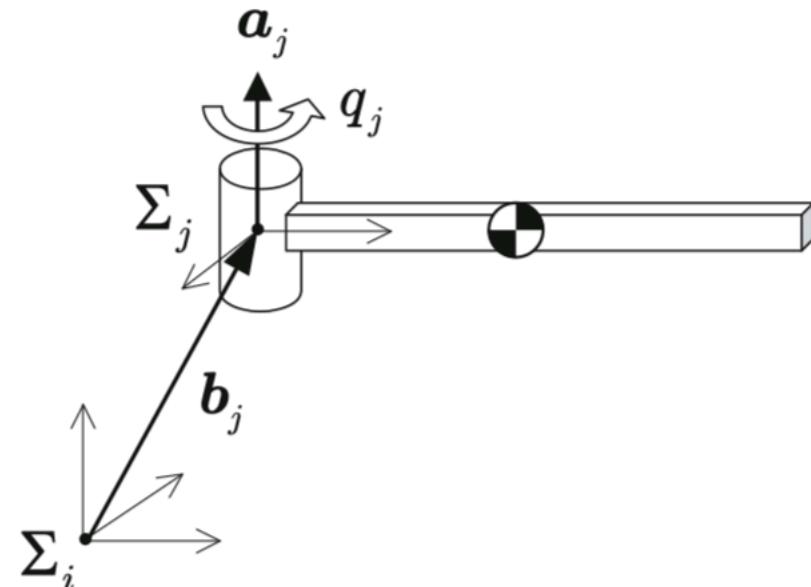
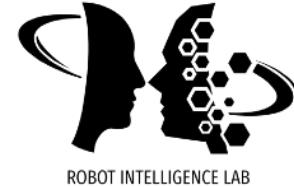
```
>> chain.link(3)

ans =

struct with fields:

    name: 'EE_R_link'
    joint_idx: 9
    p_offset: [3x1 double]
    R_offset: [3x3 double]
    mesh_path: ''
    scale: [3x1 double]
    fv: ''
    box: ''
    capsule: ''
    box_added: [1x1 struct]
    m: 0
    I_bar: [3x3 double]
    com_bar: [3x1 double]
    v: [3x1 double]
    vo: [3x1 double]
    w: [3x1 double]
```

Forward Kinematics



The homogeneous transform relative to the parent link is:

$${}^i\mathbf{T}_j = \begin{bmatrix} e^{\hat{a}_j q_j} & \mathbf{b}_j \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (2.56)$$

Next let us assume there are two links as shown in Fig. 2.22. We will assume that the absolute position and attitude of the parent link $\mathbf{p}_i, \mathbf{R}_i$ is known. Therefore, the homogeneous transform to Σ_i becomes:

$$\mathbf{T}_i = \begin{bmatrix} \mathbf{R}_i & \mathbf{p}_i \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (2.57)$$

From the chain rule of homogeneous transforms Σ_j is:

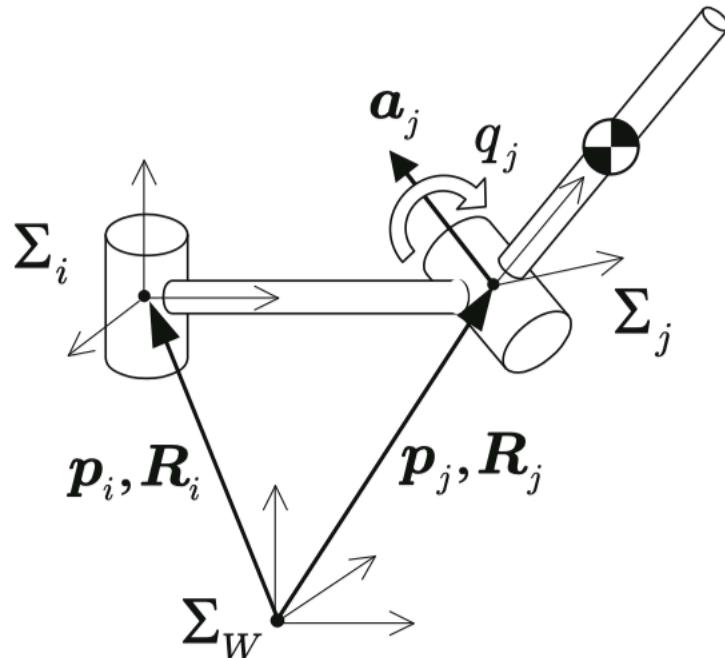
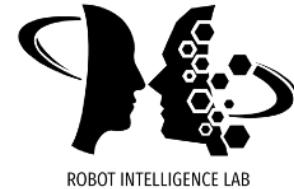
$$\mathbf{T}_j = \mathbf{T}_i {}^i\mathbf{T}_j. \quad (2.58)$$

From (2.56), (2.57) and (2.58) the absolute position (\mathbf{p}_j) and attitude (\mathbf{R}_j) of Σ_j can be calculated as being,

$$\mathbf{p}_j = \mathbf{p}_i + \mathbf{R}_i \mathbf{b}_j \quad (2.59)$$

$$\mathbf{R}_j = \mathbf{R}_i e^{\hat{a}_j q_j} \quad (2.60)$$

Forward Kinematics



```
function ForwardKinematics(j)
global uLINK

if j == 0 return; end
if j ~= 1
    i = uLINK(j).mother;
    uLINK(j).p = uLINK(i).R * uLINK(j).b + uLINK(i).p;
    uLINK(j).R = uLINK(i).R * Rodrigues(uLINK(j).a, uLINK(j).q);
end
ForwardKinematics(uLINK(j).sister);
ForwardKinematics(uLINK(j).child);
```

FK in Action



```
function chain = fk_chain(chain,idx_to)
%
% Forward kinematics
%
if isempty(idx_to)
    idx_to = get_topmost_idx(chain); % start from the topmost joint
end

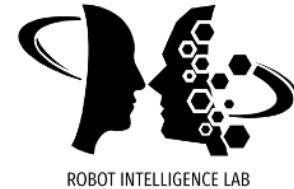
% Update p and R of joint
idx_fr = chain.joint(idx_to).parent;
if ~isempty(idx_fr)
    joint_fr = chain.joint(idx_fr);
    joint_to = chain.joint(idx_to);

        % update p
    chain.joint(idx_to).p = joint_fr.R*joint_to.p_offset + joint_fr.p;

        % update R
    if isfield(joint_to,'a') % with rotational axis
        q = joint_to.q;
        a = joint_to.a;
        chain.joint(idx_to).R = ...
            joint_fr.R*joint_to.R_offset*rodrigues(a,q);
    else % without rotational axis
        chain.joint(idx_to).R = joint_fr.R*joint_to.R_offset;
    end
end

% Recursive
for child = chain.joint(idx_to).childs
    chain = fk_chain(chain,child);
end
```

FK in Action



```
function chain = fk_chain(chain,idx_to)
%
% Forward kinematics
%
if isempty(idx_to)
    idx_to = get_topmost(chain);
end

% Update p and R of joints
idx_fr = chain.joint(idx_to).fr;
if ~isempty(idx_fr)
    joint_fr = chain.joint(idx_fr);
    joint_to = chain.joint(idx_to);

    % update p
    chain.joint(idx_to).p = joint_to.p + joint_fr.p;

    % update R
    if isfield(joint_to,'rot')
        q = joint_to.rot;
        a = joint_to.a;
        chain.joint(idx_to).R = rodrigues(a,q);
        joint_fr.R = rodrigues(a,q);
    else % without rotation
        chain.joint(idx_to).R = eye(3,3);
        joint_fr.R = eye(3,3);
    end
end

% Recursive
for child = chain.joint(idx_to).childs
    chain = fk_chain(chain,child);
end
```

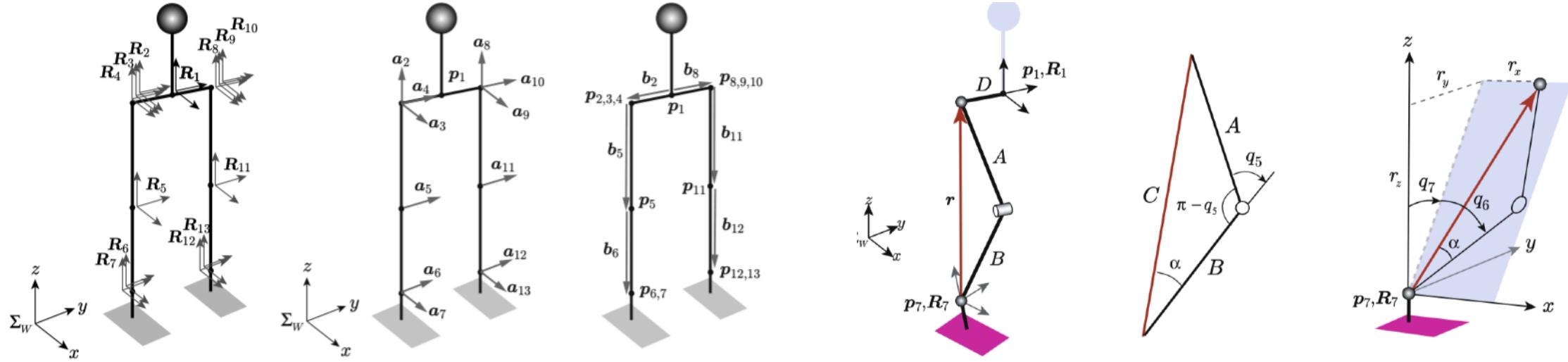
```
function R = rodrigues(a,q)
%
% Compute the rotation matrix from an angular velocity vector
% The reverse (R -> w) can be achieved by the w = r2w(R) function where w = a*q
%

if abs(norm(a)) < 1e-6
    R = eye(3,3);
    return;
end
if abs(norm(a)-1) > 1e-6
    warning('[rodrigues] a is not a unit vector (%.4e).\n',norm(a));
end

% Resolve some numerical issues
norm_a = norm(a);
a = a / norm_a;
q = q * norm_a;

% Get R
a_hat = skew(a);
R = eye(3,3) + a_hat*sin(q) + a_hat*a_hat*(1-cos(q));
```

(Analytic) Inverse Kinematics



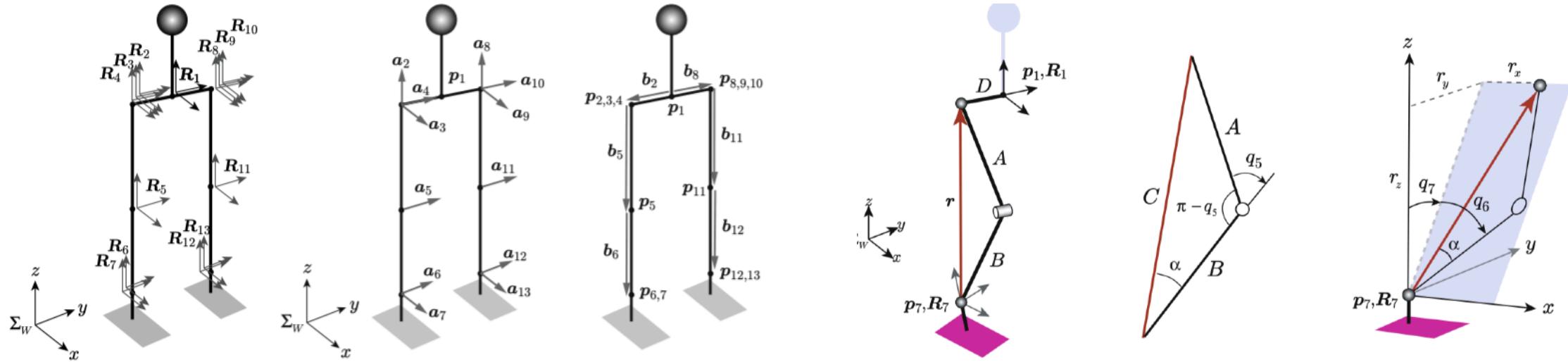
1. First compute the position of the hip

$$\mathbf{p}_2 = \mathbf{p}_1 + \mathbf{R}_1 \begin{bmatrix} 0 \\ D \\ 0 \end{bmatrix}.$$

2. Calculate the position of the crotch w.r.t. the ankle coordinate frame.

$$\mathbf{r} = \mathbf{R}_7^T (\mathbf{p}_2 - \mathbf{p}_7) \equiv [r_x \ r_y \ r_z]^T.$$

(Analytic) Inverse Kinematics



3. The distance between the ankle and the hip

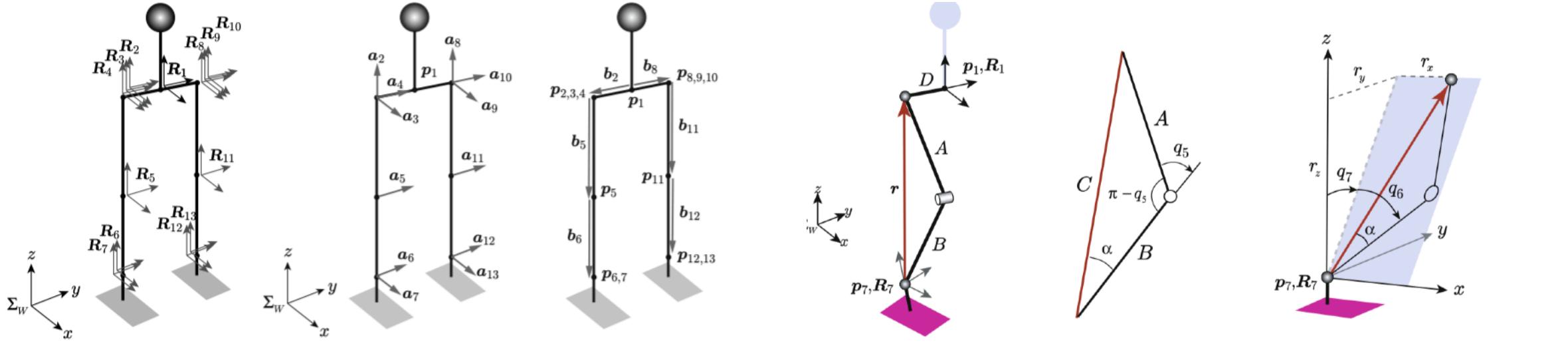
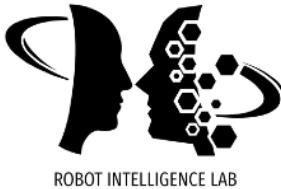
$$C = \sqrt{r_x^2 + r_y^2 + r_z^2}.$$

$$\cos(\pi - q_5) = \frac{A^2 + B^2 - C^2}{2AB}$$

4. The angle of the knees

$$q_5 = -\cos^{-1} \left(\frac{A^2 + B^2 - C^2}{2AB} \right) + \pi.$$

(Analytic) Inverse Kinematics



5. The angle of the lower end of the triangle α

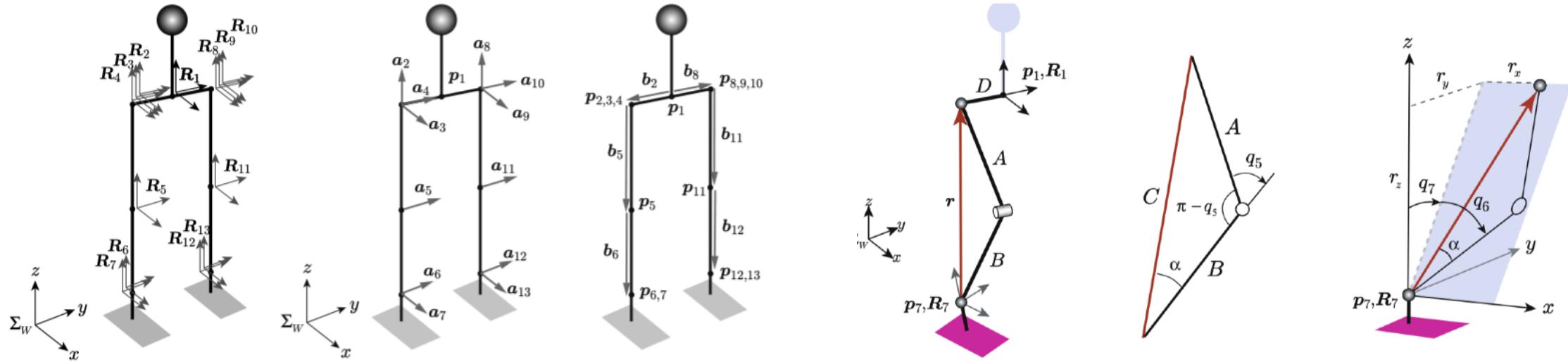
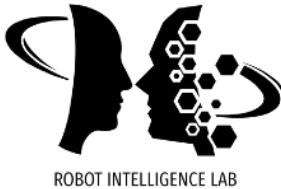
$$\alpha = \sin^{-1} \left(\frac{A \sin(\pi - q_5)}{C} \right).$$

6. The ankle roll and pitch angles

$$q_7 = \text{atan2}(r_y, r_z)$$

$$q_6 = -\text{atan2} \left(r_x, \text{sign}(r_z) \sqrt{r_y^2 + r_z^2} \right) - \alpha.$$

(Analytic) Inverse Kinematics



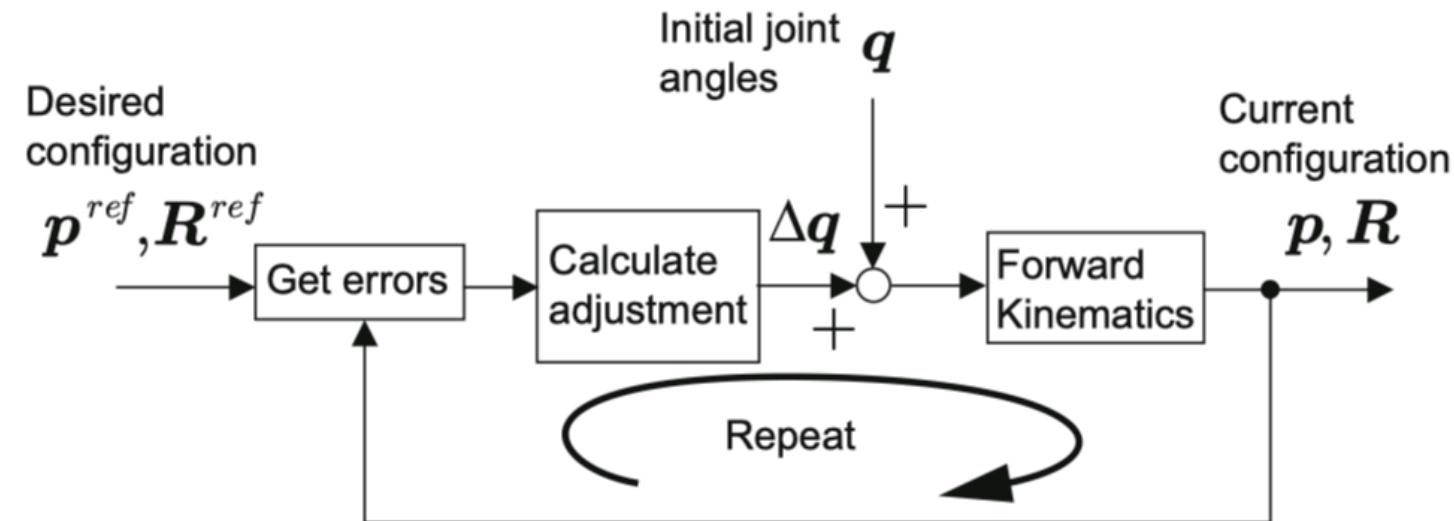
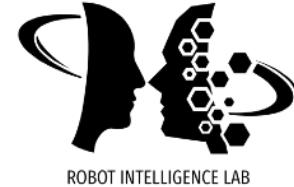
5. Other angles (right pelvis)

$$q_2 = \text{atan2}(-R_{12}, R_{22})$$

$$q_3 = \text{atan2}(R_{32}, -R_{12}s_2 + R_{22}c_2)$$

$$q_4 = \text{atan2}(-R_{31}, R_{33}).$$

(Numerical) Inverse Kinematics



Jacobian Matrix



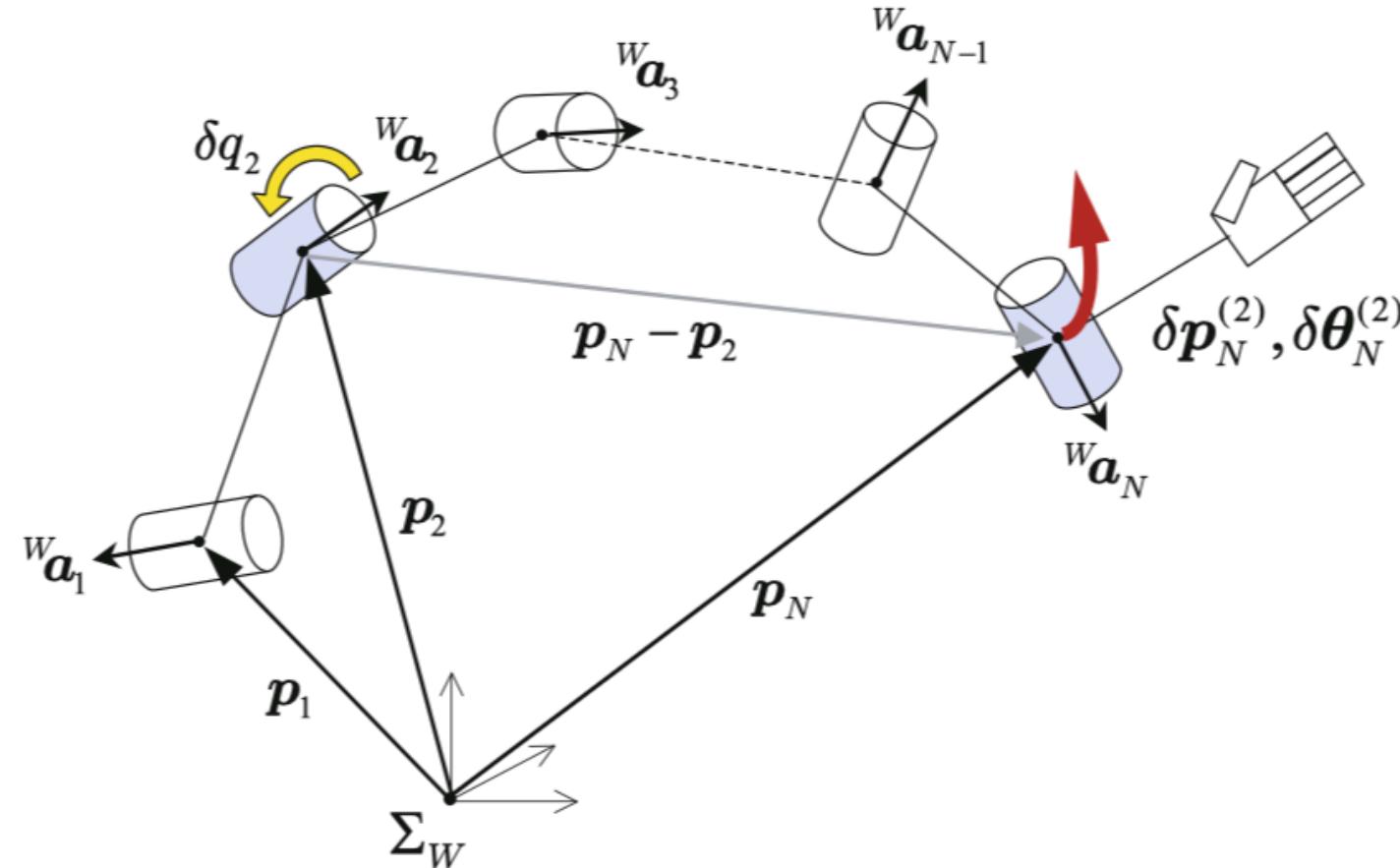
$$\begin{bmatrix} \delta p \\ \delta \theta \end{bmatrix} = \begin{bmatrix} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} & J_{16} \\ J_{21} & J_{22} & J_{23} & J_{24} & J_{25} & J_{26} \\ J_{31} & J_{32} & J_{33} & J_{34} & J_{35} & J_{36} \\ J_{41} & J_{42} & J_{43} & J_{44} & J_{45} & J_{46} \\ J_{51} & J_{52} & J_{53} & J_{54} & J_{55} & J_{56} \\ J_{61} & J_{62} & J_{63} & J_{64} & J_{65} & J_{66} \end{bmatrix} \delta q. \quad (2.69)$$

Here J_{ij} , ($i, j = 1 \dots 6$) are constants which are defined by the current position and attitude of the robots links. There are 6 because of the number of links in the leg. It is too much to write all the components each time so we will simplify it by

$$\begin{bmatrix} \delta p \\ \delta \theta \end{bmatrix} = J \delta q. \quad (2.70)$$

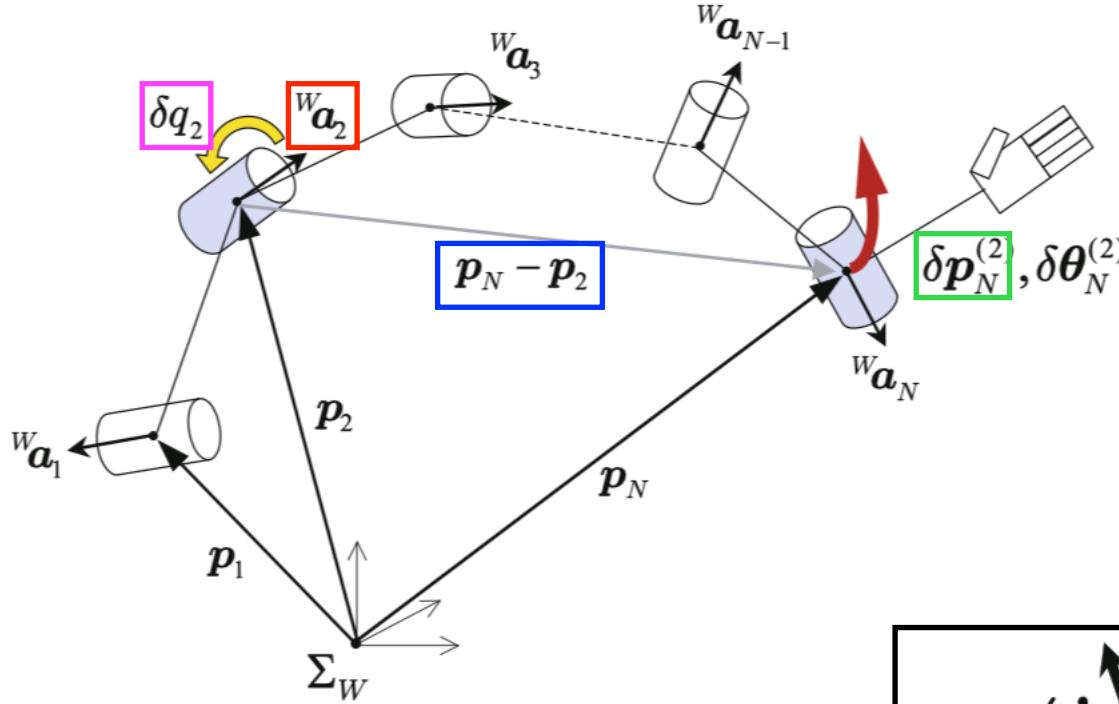
- A Jacobian matrix maps the joint velocity to the end-effector velocity.

Jacobian Matrix



- What happens to the end-effector pose (position and orientation) if we rotate the second joint with δq_2 ?

Jacobian Matrix

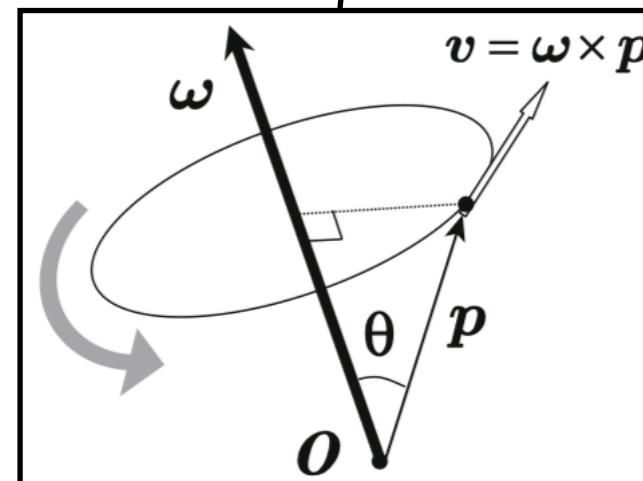


In this chain of links, let us assume we kept all the joints fixed except for the 2nd joint, which we turned by a small angle, δq_2 . The amount of the end effector (link N)'s position changed ($\delta \mathbf{p}_N^{(2)}$), and the amount the attitude of the end effector changed ($\delta \theta_N^{(2)}$) can be calculated by

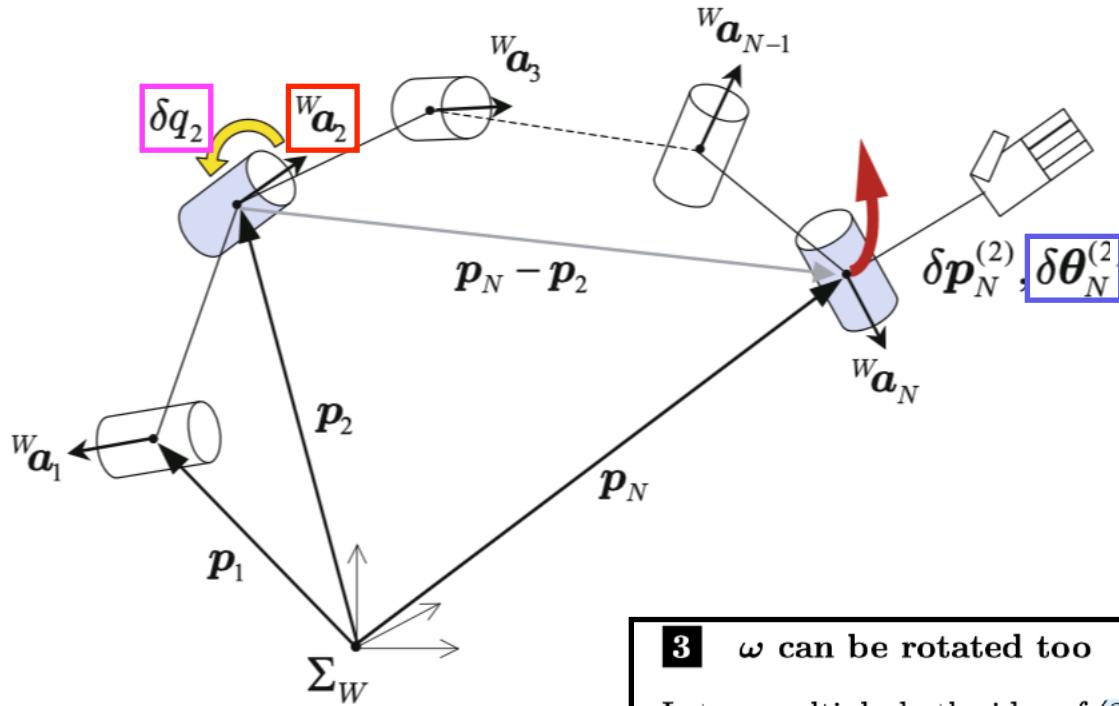
$$\begin{cases} \delta \mathbf{p}_N^{(2)} = \mathbf{w} \mathbf{a}_2 \times (\mathbf{p}_N - \mathbf{p}_2) \delta q_2 \\ \delta \theta_N^{(2)} = \mathbf{w} \mathbf{a}_2 \delta q_2 \end{cases}$$

where $\mathbf{w} \mathbf{a}_2$ is the unit vector for the second joint axis with respect to the world frame

$$\mathbf{w} \mathbf{a}_2 = \mathbf{R}_1 \mathbf{a}_2.$$



Jacobian



3 ω can be rotated too

Let us multiply both sides of (2.17) using some rotation matrix \mathbf{R}

$$\mathbf{R}\boldsymbol{\omega} = \mathbf{R}\mathbf{a}\dot{q}. \quad (2.20)$$

If we introduce following new vectors,

$$\boldsymbol{\omega}' = \mathbf{R}\boldsymbol{\omega}, \quad \mathbf{a}' = \mathbf{R}\mathbf{a}$$

then we can rewrite (2.20) as

$$\boldsymbol{\omega}' = \mathbf{a}'\dot{q}.$$

In this chain of links, let us assume we kept all the joints fixed except for the 2nd joint, which we turned by a small angle, δq_2 . The amount of the end effector (link N)'s position changed ($\delta \mathbf{p}_N^{(2)}$), and the amount the attitude of the end effector changed ($\delta \theta_N^{(2)}$) can be calculated by

$$\begin{cases} \delta \mathbf{p}_N^{(2)} = {}^W\mathbf{a}_2 \times (\mathbf{p}_N - \mathbf{p}_2) \delta q_2 \\ \delta \theta_N^{(2)} = {}^W\mathbf{a}_2 \delta q_2 \end{cases}$$

where ${}^W\mathbf{a}_2$ is the unit vector for the second joint axis with respect to the world frame

$${}^W\mathbf{a}_2 = \mathbf{R}_1 \mathbf{a}_2.$$

Jacobian



If we apply the same procedure on all the links from the 1st to the N th and calculate their sum, we may obtain the change that occurs when all the joints are rotated by a small amount

$$\begin{cases} \delta\mathbf{p}_N^{(2)} = {}^W\mathbf{a}_2 \times (\mathbf{p}_N - \mathbf{p}_2) \delta q_2 \\ \delta\boldsymbol{\theta}_N^{(2)} = {}^W\mathbf{a}_2 \delta q_2 \end{cases} . \quad (2.72)$$

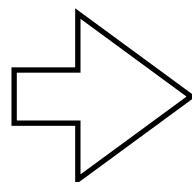
We can rewrite the above as a matrix to obtain

$$\begin{bmatrix} \delta\mathbf{p}_N \\ \delta\boldsymbol{\theta}_N \end{bmatrix} = \begin{bmatrix} {}^W\mathbf{a}_1 \times (\mathbf{p}_N - \mathbf{p}_1) & \dots & {}^W\mathbf{a}_{N-1} \times (\mathbf{p}_N - \mathbf{p}_{N-1}) & 0 \\ {}^W\mathbf{a}_1 & \dots & {}^W\mathbf{a}_{N-1} & {}^W\mathbf{a}_N \end{bmatrix} \begin{bmatrix} \delta q_1 \\ \delta q_2 \\ \vdots \\ \delta q_N \end{bmatrix} . \quad (2.73)$$

In other words, the Jacobian \mathbf{J} can be described by

$$\mathbf{J} \equiv \begin{bmatrix} {}^W\mathbf{a}_1 \times (\mathbf{p}_N - \mathbf{p}_1) & {}^W\mathbf{a}_2 \times (\mathbf{p}_N - \mathbf{p}_2) & \dots & {}^W\mathbf{a}_{N-1} \times (\mathbf{p}_N - \mathbf{p}_{N-1}) & 0 \\ {}^W\mathbf{a}_1 & {}^W\mathbf{a}_2 & \dots & {}^W\mathbf{a}_{N-1} & {}^W\mathbf{a}_N \end{bmatrix} . \quad (2.74)$$

$$\begin{cases} \delta\mathbf{p}_N^{(2)} = {}^W\mathbf{a}_2 \times (\mathbf{p}_N - \mathbf{p}_2) \delta q_2 \\ \delta\boldsymbol{\theta}_N^{(2)} = {}^W\mathbf{a}_2 \delta q_2 \end{cases}$$



(Numerical) Inverse Kinematics

$$\dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{q}}.$$

Here $\dot{\mathbf{x}}$ is a six dimension vector for the target velocity of the end-effector. Let us introduce an error vector \mathbf{e} since this equality will no longer be satisfied at singularity.

$$\mathbf{e} := \dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}}. \quad (2.78)$$

We can always make the error \mathbf{e} as small as possible, while it cannot reach zero in general. For this purpose, we define a cost function

$$E(\dot{\mathbf{q}}) = \frac{1}{2}\mathbf{e}^T\mathbf{e}. \quad (2.79)$$

If $\dot{\mathbf{q}}$ minimizes $E(\dot{\mathbf{q}})$, we will have,

$$\frac{\partial E(\dot{\mathbf{q}})}{\partial \dot{\mathbf{q}}} = 0. \quad (2.80)$$

By substituting (2.78) and (2.79), we get

$$\frac{\partial E(\dot{\mathbf{q}})}{\partial \dot{\mathbf{q}}} = -\mathbf{J}^T\dot{\mathbf{x}} + \mathbf{J}^T\mathbf{J}\dot{\mathbf{q}} = 0. \quad (2.81)$$

Therefore, $\dot{\mathbf{q}}$ to minimize $E(\dot{\mathbf{q}})$ is obtained by

$$\dot{\mathbf{q}} = (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T\dot{\mathbf{x}}. \quad (2.82)$$

Unfortunately, we cannot use this at singularity because,

$$\det(\mathbf{J}^T\mathbf{J}) = \det(\mathbf{J}^T)\det(\mathbf{J}) = 0.$$

This means the inverse at the right side of (2.82) is not solvable.

IK with Singularity Robustness

Now, we slightly modify the cost function.

$$E(\dot{\mathbf{q}}) = \frac{1}{2} \mathbf{e}^T \mathbf{e} + \frac{\lambda}{2} \dot{\mathbf{q}}^T \dot{\mathbf{q}} \quad (2.83)$$

This evaluates the joint speed magnitude as well as the end-effector speed error. The joint speed is accounted by increasing the positive scalar λ . Again, by substituting (2.78) and (2.79), we get

$$\frac{\partial E(\dot{\mathbf{q}})}{\partial \dot{\mathbf{q}}} = -\mathbf{J}^T \dot{\mathbf{x}} + (\mathbf{J}^T \mathbf{J} + \lambda \mathbf{E}) \dot{\mathbf{q}} = 0, \quad (2.84)$$

where \mathbf{E} is an identity matrix of the same size of $\mathbf{J}^T \mathbf{J}$. The cost $E(\dot{\mathbf{q}})$ is minimized by

$$\dot{\mathbf{q}} = (\mathbf{J}^T \mathbf{J} + \lambda \mathbf{E})^{-1} \mathbf{J}^T \dot{\mathbf{x}}. \quad (2.85)$$

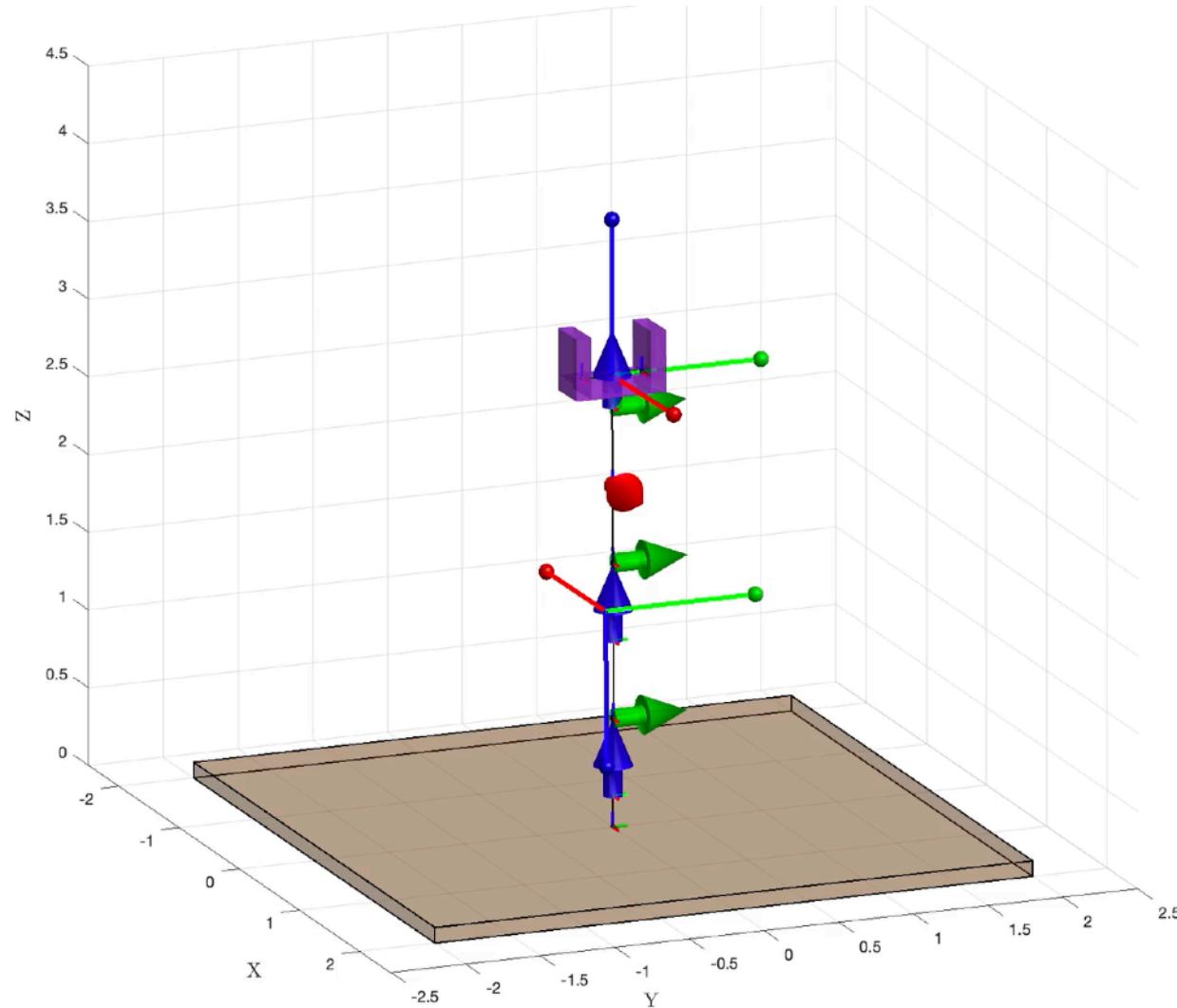
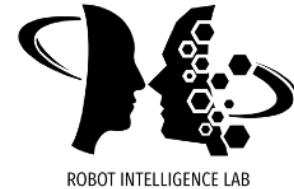
Even at singularity where $\det(\mathbf{J}) = 0$, we can always solve the above equation by using proper λ . Let us define a new matrix.

$$\begin{aligned} \dot{\mathbf{q}} &= \mathbf{J}^{\# \lambda} \dot{\mathbf{x}} \\ \mathbf{J}^{\# \lambda} &:= (\mathbf{J}^T \mathbf{J} + \lambda \mathbf{E})^{-1} \mathbf{J}^T. \end{aligned} \quad (2.86)$$

The matrix $\mathbf{J}^{\# \lambda}$ is called an SR inverse²¹.

*SR: singularity-Robust

IK in Action



IK in Action



$$J \equiv \begin{bmatrix} \mathbf{W}\mathbf{a}_1 \times (\mathbf{p}_N - \mathbf{p}_1) & \mathbf{W}\mathbf{a}_2 \times (\mathbf{p}_N - \mathbf{p}_2) & \dots & \mathbf{W}\mathbf{a}_{N-1} \times (\mathbf{p}_N - \mathbf{p}_{N-1}) & 0 \\ \mathbf{W}\mathbf{a}_1 & \mathbf{W}\mathbf{a}_2 & \dots & \mathbf{W}\mathbf{a}_{N-1} & \mathbf{W}\mathbf{a}_N \end{bmatrix}.$$

```
% Get the current target joint position
p_trgt_curr = chain.joint(idx_cell(chain.joint_names,joint_name_trgt)).p;
R_trgt_curr = chain.joint(idx_cell(chain.joint_names,joint_name_trgt)).R;

% Get the list of indices from root joint to the target joint
joint_idxs_route = get_idx_route(chain,joint_name_trgt);

% Intersect 'joint_idxs_route' with 'joint_idxs_to_ctrl' to get actual using indices
joint_idxs_use = intersect(joint_idxs_route,joint_idxs_to_ctrl);
n_use = length(joint_idxs_use);

% Compute the Jacobian matrix (2.74)
n_ctrl = length(joint_names_to_ctrl);
J = zeros(6,n_ctrl);
for i_idx = 1:n_ctrl % along the joint route
    joint_idx_to_ctrl = joint_idxs_to_ctrl(i_idx);
    parent = chain.joint(joint_idx_to_ctrl).parent; % parent joint index
    p_joint_ctrl = chain.joint(joint_idx_to_ctrl).p; % joint position
    R_offset = chain.joint(joint_idx_to_ctrl).R_offset; % current joint's rotation offset
    % Rotation axis in {W}
    a = chain.joint(parent).R * R_offset * chain.joint(joint_idx_to_ctrl).a;
    % 'idx_append': which column to append
    joint_name_append = chain.joint_names{joint_idx_to_ctrl};
    idx_append = idx_cell(joint_names_to_ctrl,joint_name_append); % which column to append
    J(:,idx_append) = [...
        cv(cross(a,p_trgt_curr-p_joint_ctrl));... % position part
        cv(a)... % orientation part (simply rotation axis in {W})
    ];
end
```

IK in Action



Now, we slightly modify the cost function.

$$E(\dot{\mathbf{q}}) = \frac{1}{2}\mathbf{e}^T\mathbf{e} + \frac{\lambda}{2}\dot{\mathbf{q}}^T\dot{\mathbf{q}} \quad (2.83)$$

This evaluates the joint speed magnitude as well as the end-effector speed error. The joint speed is accounted by increasing the positive scalar λ . Again, by substituting (2.78) and (2.79), we get

$$\frac{\partial E(\dot{\mathbf{q}})}{\partial \dot{\mathbf{q}}} = -\mathbf{J}^T\dot{\mathbf{x}} + (\mathbf{J}^T\mathbf{J} + \lambda\mathbf{E})\dot{\mathbf{q}} = 0, \quad (2.84)$$

where \mathbf{E} is an identity matrix of the same size of $\mathbf{J}^T\mathbf{J}$. The cost $E(\dot{\mathbf{q}})$ is minimized by

$$\dot{\mathbf{q}} = (\mathbf{J}^T\mathbf{J} + \lambda\mathbf{E})^{-1}\mathbf{J}^T\dot{\mathbf{x}}. \quad (2.85)$$

Even at singularity where $\det(\mathbf{J}) = 0$, we can always solve the above equation by using proper λ . Let us define a new matrix.

$$\begin{aligned} \dot{\mathbf{q}} &= \mathbf{J}^{\# \lambda}\dot{\mathbf{x}} \\ \mathbf{J}^{\# \lambda} &:= (\mathbf{J}^T\mathbf{J} + \lambda\mathbf{E})^{-1}\mathbf{J}^T. \end{aligned} \quad (2.86)$$

The matrix $\mathbf{J}^{\# \lambda}$ is called an SR inverse²¹.

```
% Compute the error
p_err_weight = 1;
w_err_weight = 1;
[p_trgt_goal,R_trgt_goal] = t2pr(T_trgt_goal);
p_err = p_trgt_goal-p_trgt_curr;
w_err = R_trgt_curr * r2w(R_trgt_curr' * R_trgt_goal);
ik_err = [p_err_weight*p_err; w_err_weight*w_err];
ik_err_avg = mean(abs(ik_err));

% Compute dq
lambda = 0.1*ik_err_avg+1e-4; % damping term
dq = ((J'*J + lambda*eye(n_ctrl,n_ctrl)) \ J'*ik_err);
step_size = 1.0;
dq = trim_scale(step_size*dq,10*D2R);

% Update
q = q + dq;
chain = update_chain_q(chain,chain.rev_joint_names,q);
```

IK in Action



```
% Initialize a kinematic chain
chain = init_chain('name','kinematic_chain');
% Add joint to the chain
chain = add_joint_to_chain(chain,'name','world');
chain = add_joint_to_chain(chain,'name','J1','parent_name','world',...
    'p_offset',cv([0,0,0.5]),'a',cv([0,0,1]));
chain = add_joint_to_chain(chain,'name','J2','parent_name','J1',...
    'p_offset',cv([0,0,0.5]),'a',cv([1,0,0]));
chain = add_joint_to_chain(chain,'name','J3','parent_name','J2',...
    'p_offset',cv([0,0,0.5]),'a',cv([0,1,0]));
chain = add_joint_to_chain(chain,'name','J4','parent_name','J3',...
    'p_offset',cv([0,0,0.5]),'a',cv([0,0,1]));
chain = add_joint_to_chain(chain,'name','J5','parent_name','J4',...
    'p_offset',cv([0,0,0.5]),'a',cv([1,0,0]));
chain = add_joint_to_chain(chain,'name','J6','parent_name','J5',...
    'p_offset',cv([0,0,0.5]),'a',cv([0,1,0]));
chain = add_joint_to_chain(chain,'name','J7','parent_name','J6',...
    'p_offset',cv([0,0,0.5]),'a',cv([1,0,0]));
chain = add_joint_to_chain(chain,'name','EE','parent_name','J7',...
    'p_offset',cv([0,0,0.2]),'a',cv([0,0,0]));
chain = add_joint_to_chain(chain,'name','EE_R','parent_name','EE',...
    'p_offset',cv([0,0.2,0]),'a',cv([0,0,0]));
chain = add_joint_to_chain(chain,'name','EE_L','parent_name','EE',...
    'p_offset',cv([0,-0.2,0]),'a',cv([0,0,0]));
% Add link to the chain
box_added = struct('xyz_min',[ -2,-2,0 ],'xyz_len',[ 4,4,0.1 ],...
    'p_offset',cv([0,0,0]),'R_offset',rpy2r([0,0,0]*D2R),...
    'color',0.3*[1,1,1],'alpha',0.5,'ec','k');
chain = add_link_to_chain(chain,'name','base_link','joint_name','world','box_added',box_added);
box_added = struct('xyz_min',[ -0.15,-0.3,-0.1 ],'xyz_len',[ 0.3,0.6,0.1 ],...
    'p_offset',cv([0,0,0]),'R_offset',rpy2r([0,0,0]*D2R),...
    'color','m','alpha',0.5,'ec','k');
chain = add_link_to_chain(chain,'name','EE_link','joint_name','EE','box_added',box_added);
box_added = struct('xyz_min',[ -0.15,0,-0.1 ],'xyz_len',[ 0.3,0.1,0.4 ],...
    'p_offset',cv([0,0,0]),'R_offset',rpy2r([0,0,0]*D2R),...
    'color','m','alpha',0.5,'ec','k');
chain = add_link_to_chain(chain,'name','EE_R_link','joint_name','EE_R','box_added',box_added);
box_added = struct('xyz_min',[ -0.15,-0.1,-0.1 ],'xyz_len',[ 0.3,0.1,0.4 ],...
    'p_offset',cv([0,0,0]),'R_offset',rpy2r([0,0,0]*D2R),...
    'color','m','alpha',0.5,'ec','k');
chain = add_link_to_chain(chain,'name','EE_L_link','joint_name','EE_L','box_added',box_added);

% Initialize the kinematic chain
q = 1e-6*randn(chain.n_rev_joint,1);
chain = update_chain_q(chain,chain.rev_joint_names,q);

% Joint names to control
joint_names_to_ctrl = chain.rev_joint_names;
```

```
% Get the current target joint position
p_trgt_curr = chain.joint(idx_cell(chain.joint_names,joint_name_trgt)).p;
R_trgt_curr = chain.joint(idx_cell(chain.joint_names,joint_name_trgt)).R;

% Get the list of indices from root joint to the target joint
joint_idxs_route = get_idx_route(chain,joint_name_trgt);

% Intersect 'joint_idxs_route' with 'joint_idxs_to_control' to get actual using indices
joint_idxs_to_ctrl = idxs_cell(chain.joint_names,joint_names_to_ctrl);
joint_idxs_use = intersect(joint_idxs_route,joint_idxs_to_ctrl);
n_use = length(joint_idxs_use);

% Compute the Jacobian matrix (2.74)
n_ctrl = length(joint_names_to_ctrl);
J = zeros(6,n_ctrl);
for i_idx = 1:n_ctrl % along the joint route
    joint_idx_to_ctrl = joint_idxs_to_ctrl(i_idx);
    parent = chain.joint(joint_idx_to_ctrl).parent; % parent joint index
    p_joint_ctrl = chain.joint(joint_idx_to_ctrl).p; % joint position
    R_offset = chain.joint(joint_idx_to_ctrl).R_offset; % current joint's rotation offset

    % Rotation axis in the world coordinate
    a = chain.joint(parent).R * R_offset * chain.joint(joint_idx_to_ctrl).a;

    % 'idx_append': which column to append
    joint_name_append = chain.joint_names{joint_idx_to_ctrl};
    idx_append = idx_cell(joint_names_to_ctrl,joint_name_append); % which column to append
    J(:,idx_append) = [...
        cv(cross(a,p_trgt_curr-p_joint_ctrl));... % position part
        cv(a)... % orientation part (simply rotation axis in {W})
    ];
end

% Compute the error
p_err_weight = 1;
w_err_weight = 1;
[p_trgt_goal,R_trgt_goal] = t2pr(T_trgt_goal);
p_err = p_trgt_goal - p_trgt_curr;
w_err = R_trgt_curr * r2w(R_trgt_curr'*R_trgt_goal);
ik_err = [p_err_weight*p_err; w_err_weight*w_err];
ik_err_avg = mean(abs(ik_err));

% Compute dq
lambda = 0.1*ik_err_avg+1e-4; % damping term proportional to the error
dq = (J'*J + lambda*eye(n_ctrl,n_ctrl)) \ J' * ik_err;
step_size = 1.0;
dq = trim_scale(step_size*dq,10*D2R);

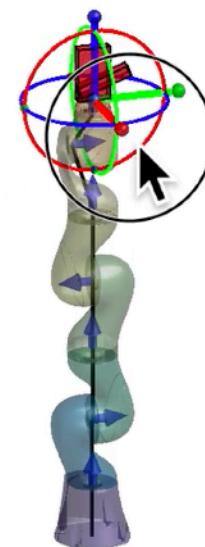
% Update
q = get_q_chain(chain,joint_names_to_ctrl);
q = q + dq;
chain = update_chain_q(chain,joint_names_to_ctrl,q);
```

IK with Interactive Markers

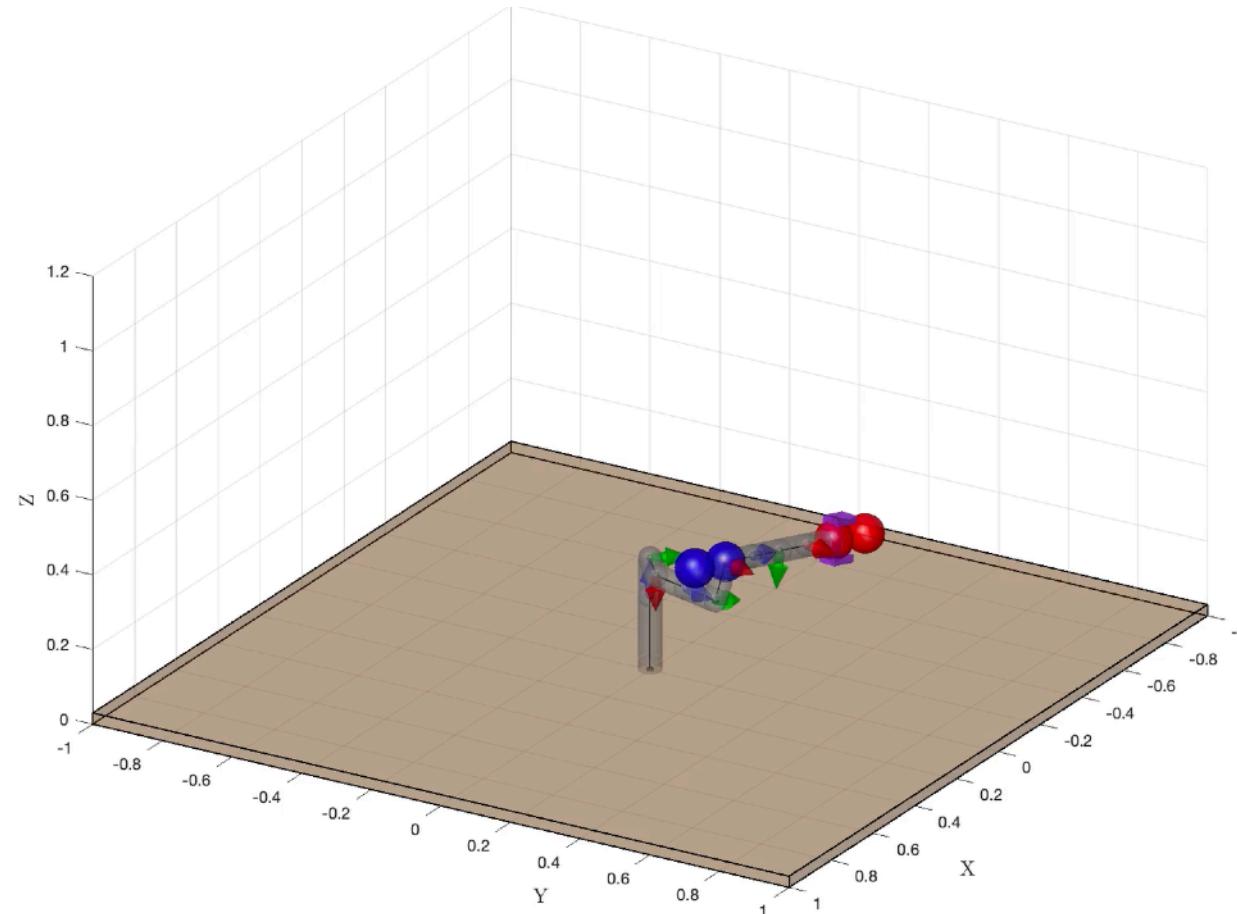


[Consider Joint Limit] Tick:[4495] IK error:[0.011]

(Press [t]:toggle [q]:quit)



Homework



Implement IK with two different target joints.
(Hint: Augmented Jacobian Method)

Thank You



ROBOT INTELLIGENCE LAB