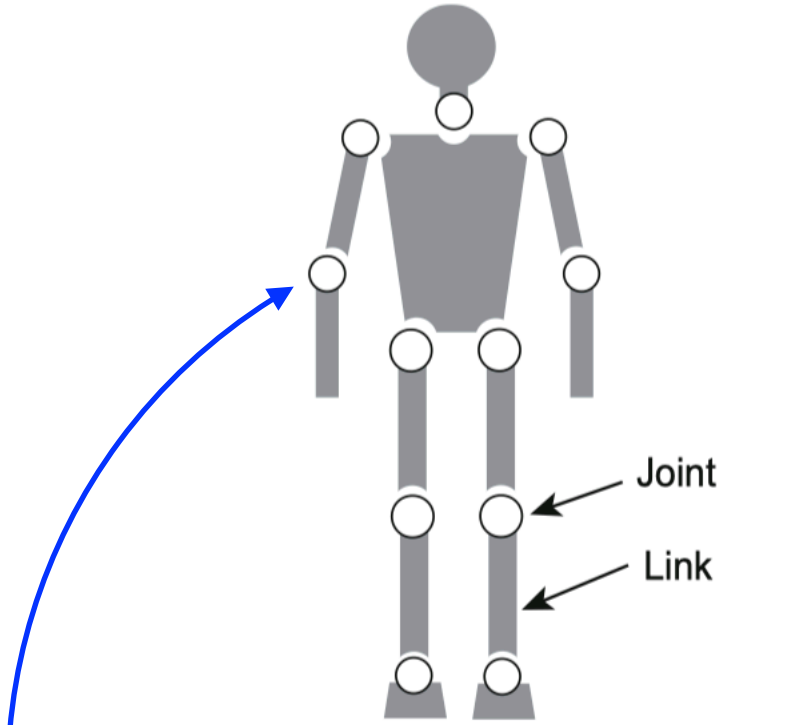
Intelligent Robotics

Kinematics

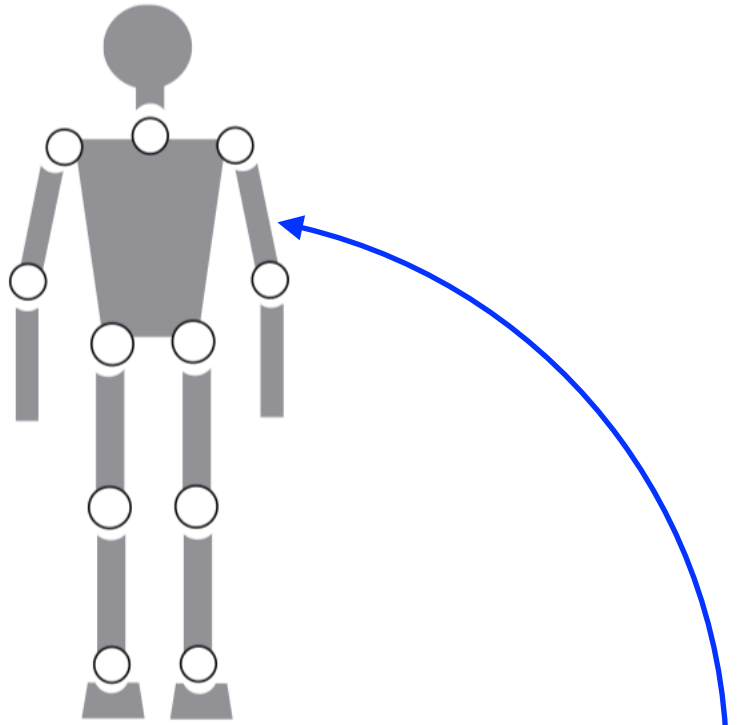
Sungjoon Choi, Korea University

Kinematic Chain

ROBOT INTELLIGENCE LAB



(a) Separate into joint+link units



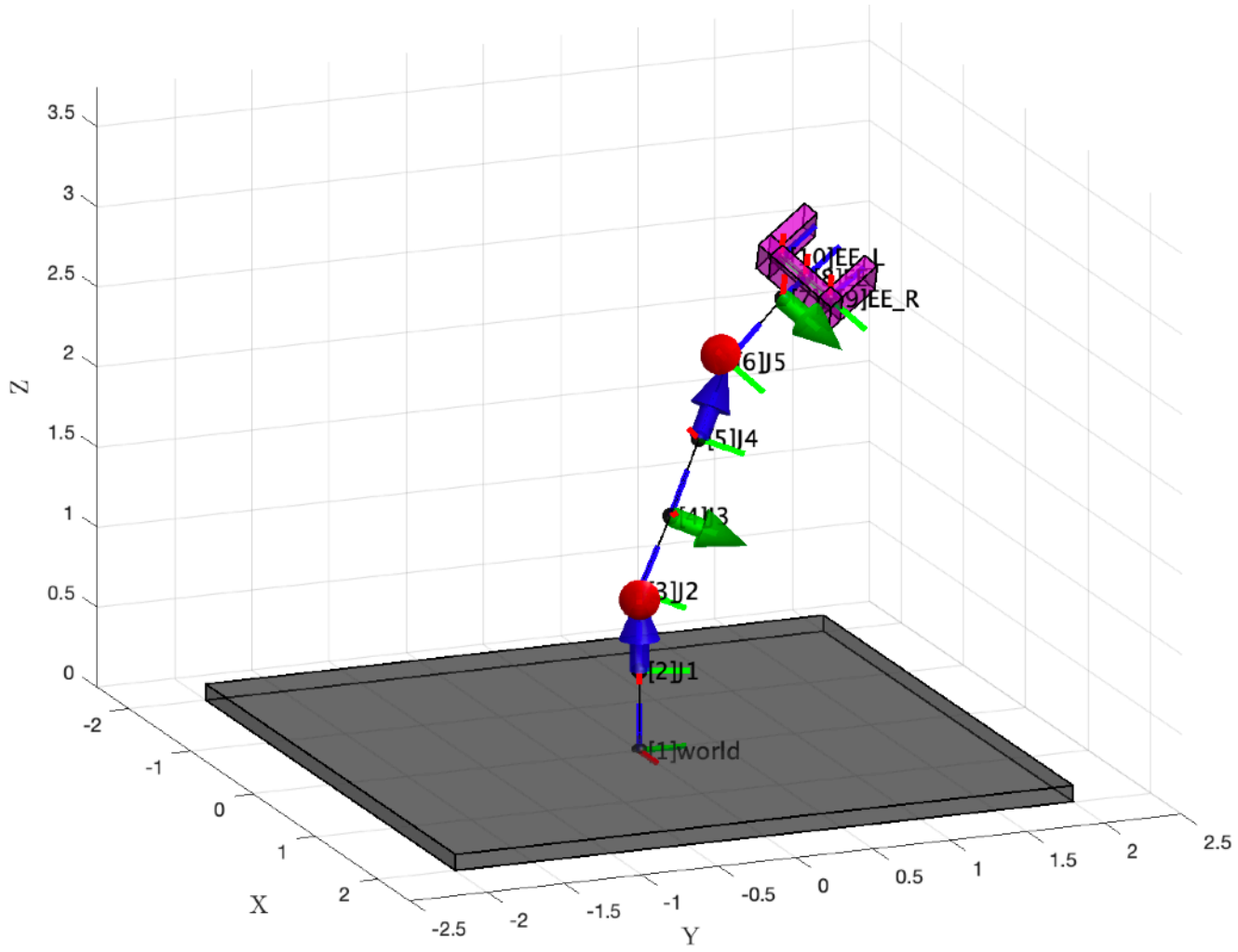
(b) Separate into link+joint units

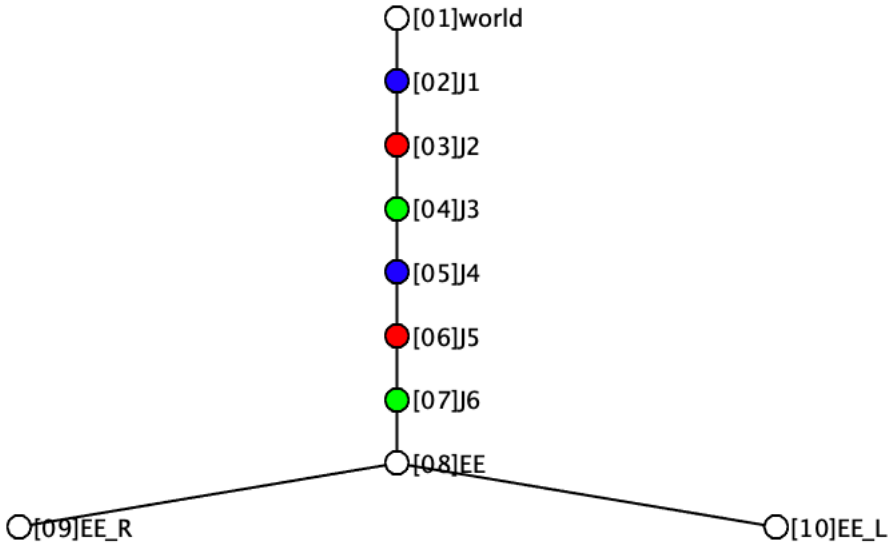
One **joint** contains a **link**, and it makes programming more convenient as kinematic chains are defined w.r.t **joints**. However, it cannot handle **joints** with multiple **links**.

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

Kinematic Chain

ROBOT INTELLIGENCE LAB





Kinematic Chain

ROBOT INTELLIGENCE LAB

```
struct with fields:

             name: 'kinematic_chain'
               dt: 0.0100
            joint: [1×10 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: [1×4 struct]
       link_names: {'base_link'  'EE_link'  'EE_R_link'  'EE_L_link'}
           n_link: 4
```

Kinematic Chain (Joint)

ROBOT INTELLIGENCE LAB

```
>> chain.joint

ans =

  1×10 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)

ROBOT INTELLIGENCE LAB

```
>> chain.joint(1)

ans =

  struct with fields:

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

```
>> chain.joint(2)

ans =

  struct with fields:

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

```
>> chain.joint(3)

ans =

  struct with fields:

        name: 'J2'
           p: [3×1 double]
           R: [3×3 double]
           a: [3×1 double]
        type: 'revolute'
    p_offset: [3×1 double]
    R_offset: [3×3 double]
           q: -1.3464
          dq: 0
         ddq: 0
      q_diff: 0.0139
      q_prev: -1.3464
           v: [3×1 double]
          vo: [3×1 double]
           w: [3×1 double]
         dvo: [3×1 double]
          dw: [3×1 double]
           u: 0
       ext_f: [3×1 double]
      parent: 2
      childs: 4
    link_idx: []
       limit: [-Inf Inf]
```

Kinematic Chain (Link)

ROBOT INTELLIGENCE LAB

```
>> chain.link

ans =

  1×4 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)

ROBOT INTELLIGENCE LAB

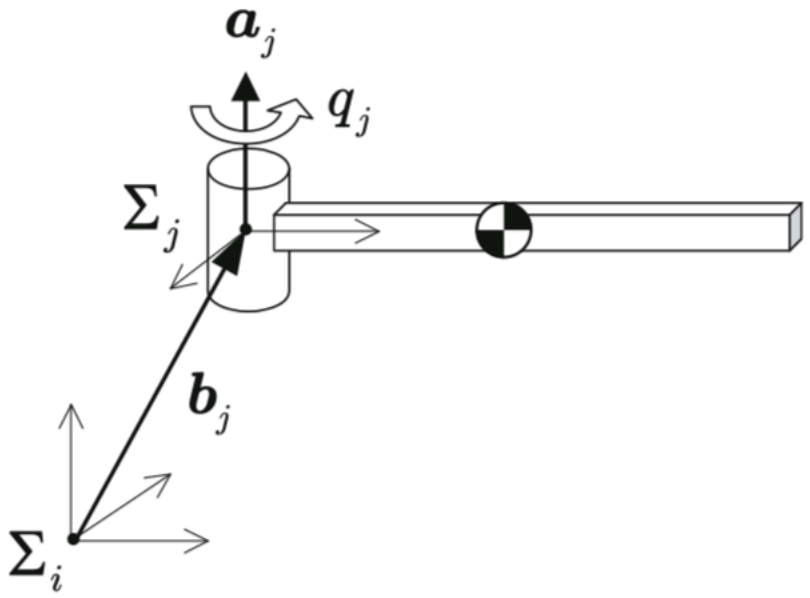
```
>> chain.link(1)

ans =

  struct with fields:

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

```
>> chain.link(2)

ans =

  struct with fields:

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

```
>> chain.link(3)

ans =

  struct with fields:

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

Forward Kinematics

ROBOT INTELLIGENCE LAB



The homogeneous transform relative to the parent link is:

$$ {}^{i}\boldsymbol{T}_{j} = \begin{bmatrix} e^{\widehat{a}_j q_j} & \boldsymbol{b}_j \\ 0\,0\,0 & 1 \end{bmatrix}. \tag{2.56} $$

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 $\boldsymbol{p}_i, \boldsymbol{R}_i$ is known. Therefore, the homogeneous transform to Σ_i becomes:

$$ \boldsymbol{T}_{i} = \begin{bmatrix} \boldsymbol{R}_i & \boldsymbol{p}_i \\ 0\,0\,0 & 1 \end{bmatrix}. \tag{2.57} $$

From the chain rule of homogeneous transforms Σ_j is:

$$ \boldsymbol{T}_{j} = \boldsymbol{T}_{i}\,{}^{i}\boldsymbol{T}_{j}. \tag{2.58} $$

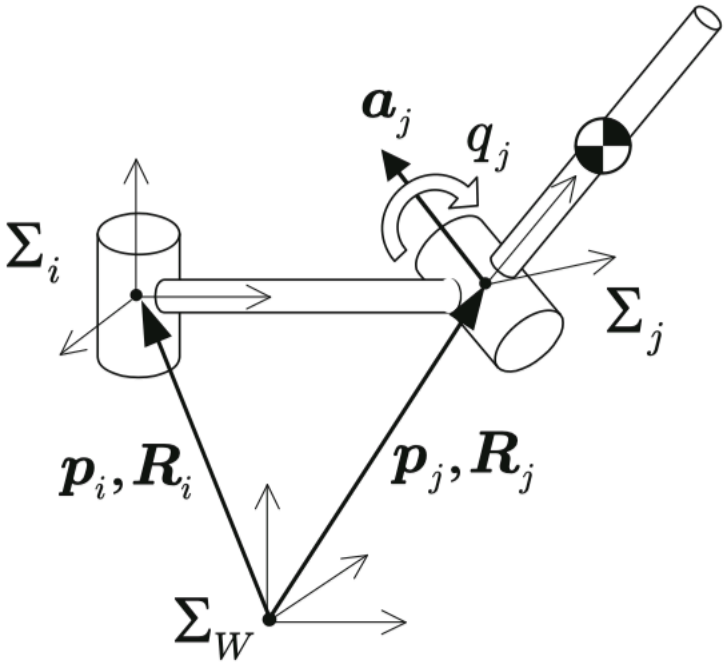
From (2.56), (2.57) and (2.58) the absolute position $(\boldsymbol{p}_j)$ and attitude $(\boldsymbol{R}_j)$ of Σ_j can be calculated as being,

$$ \boldsymbol{p}_j = \boldsymbol{p}_i + \boldsymbol{R}_i \boldsymbol{b}_j \tag{2.59} $$

$$ \boldsymbol{R}_j = \boldsymbol{R}_i e^{\widehat{a}_j q_j} \tag{2.60} $$

Forward Kinematics

ROBOT INTELLIGENCE LAB




```
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

ROBOT INTELLIGENCE LAB

```
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
```

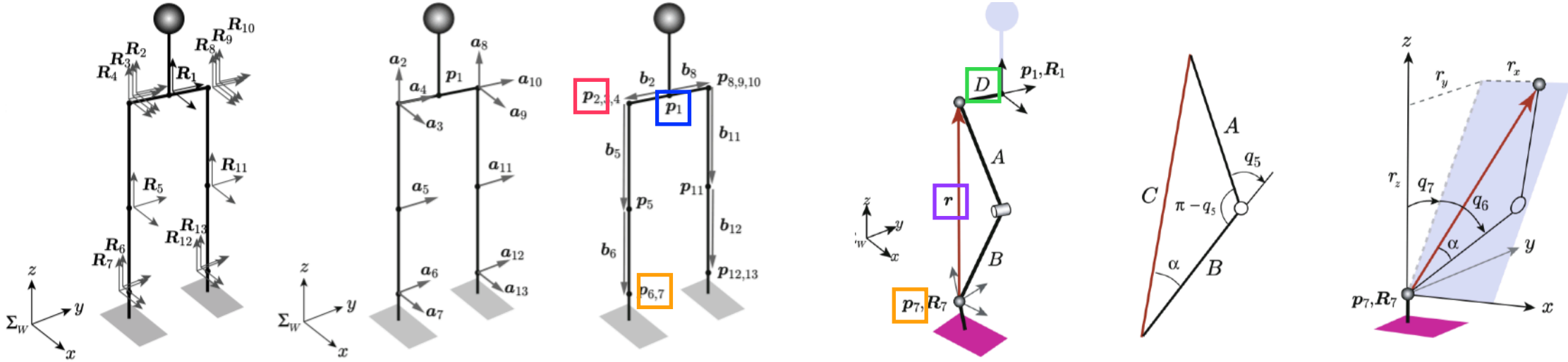
FK in Action

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

    % update R
    if isfield(joint_to,'a') % with rotational ax
        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
```

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

ROBOT INTELLIGENCE LAB



1. First compute the position of the hip

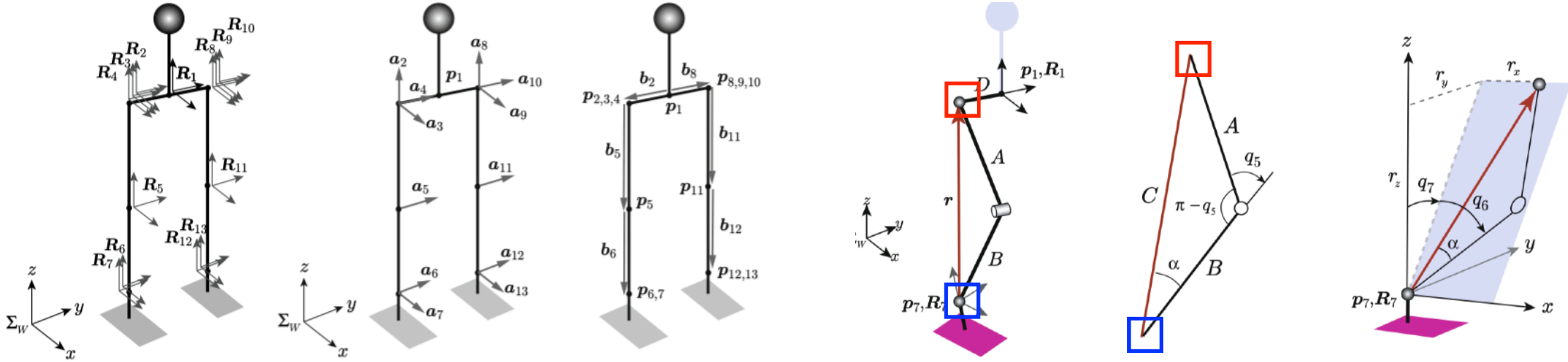
$$\boxed{\boldsymbol{p}_2} = \boxed{\boldsymbol{p}_1} + \boldsymbol{R}_1 \begin{bmatrix} 0 \\ \boxed{D} \\ 0 \end{bmatrix}.$$

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

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

(Analytic) Inverse Kinematics

ROBOT INTELLIGENCE LAB



3. The distance between the **ankle** and the **hip**

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

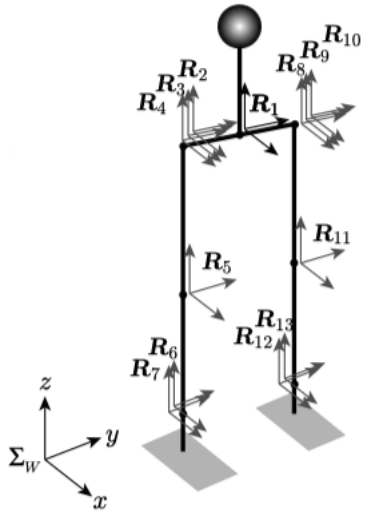
$$\cos(\pi - q_5) = \frac{A^2 + B^2 - C^2}{2AB}$$

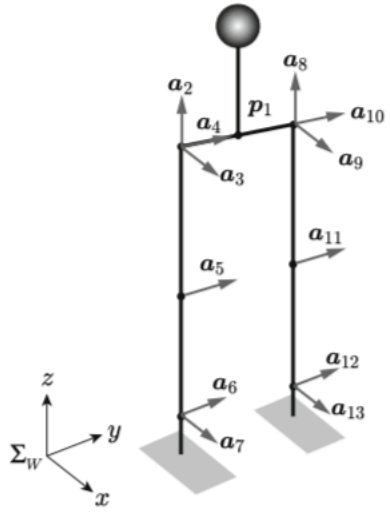
4. The angle of the knees

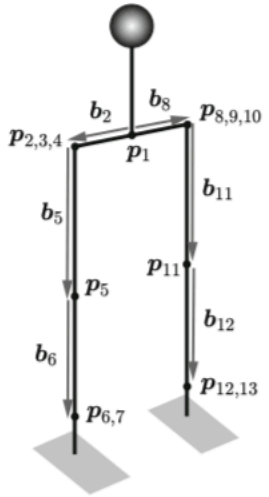
$$q_5 = -\cos^{-1}\left(\frac{A^2 + B^2 - C^2}{2AB}\right) + \pi.$$

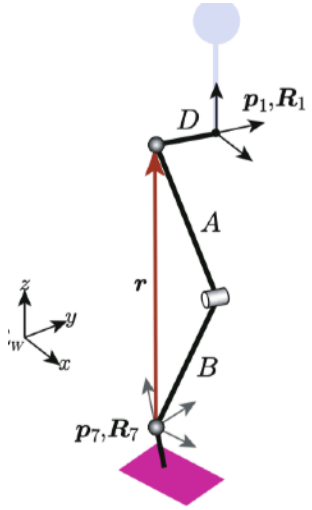
(Analytic) Inverse Kinematics

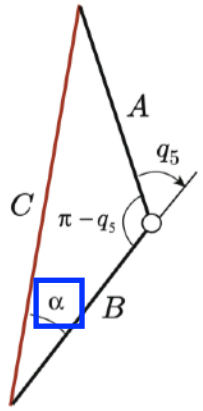
ROBOT INTELLIGENCE LAB

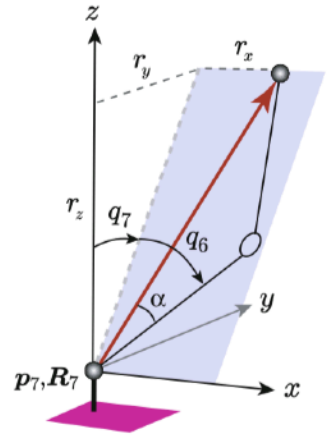












$$C\sin(\alpha) = A\sin(\pi - q_5)$$

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

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

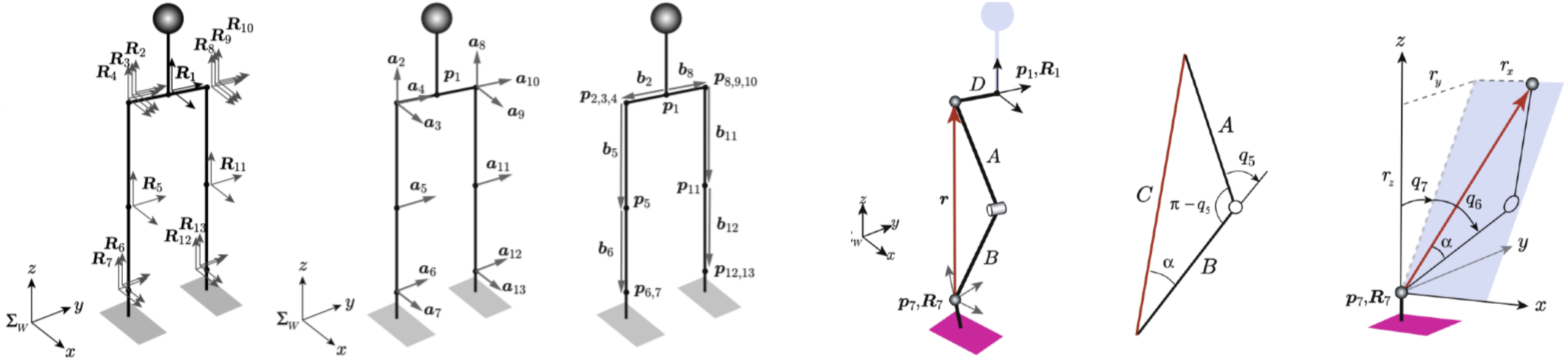
6. The ankle roll and pitch angles

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

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

(Analytic) Inverse Kinematics

ROBOT INTELLIGENCE LAB

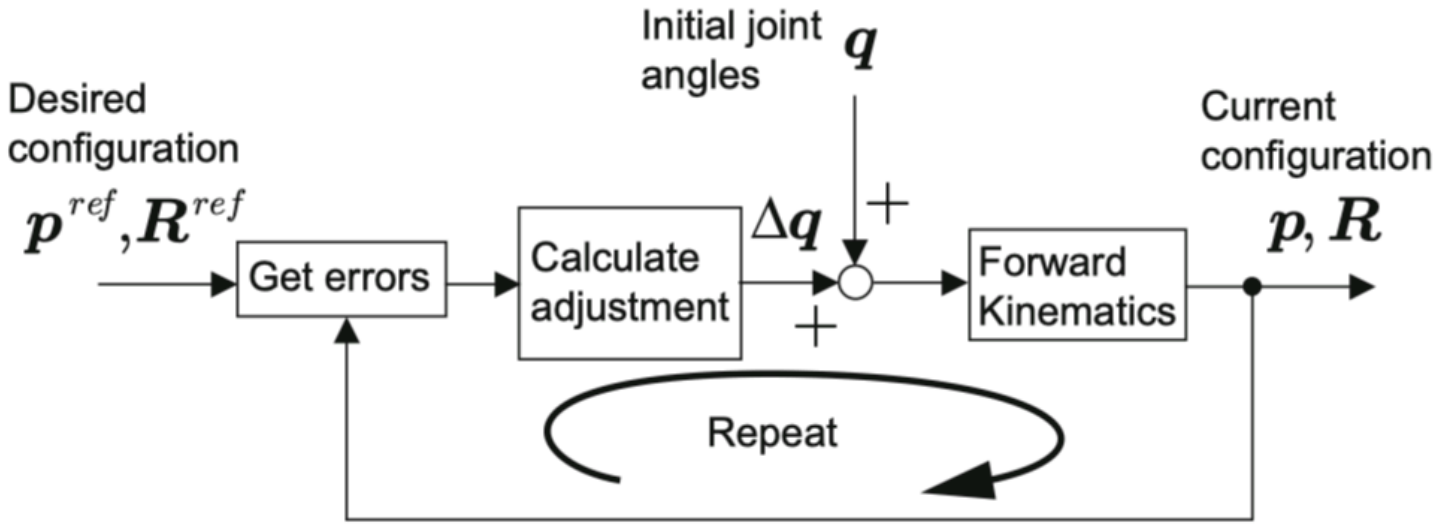


5. Other angles (right pelvis)

$$q_2 = \text{atan2}(-R_{12}, R_{22})$$
$$q_3 = \text{atan2}(R_{32}, -R_{12}s_2 + R_{22}c_2)$$
$$q_4 = \text{atan2}(-R_{31}, R_{33}).$$

(Numerical) Inverse Kinematics

ROBOT INTELLIGENCE LAB



Jacobian Matrix

ROBOT INTELLIGENCE LAB

position information

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

rotation information

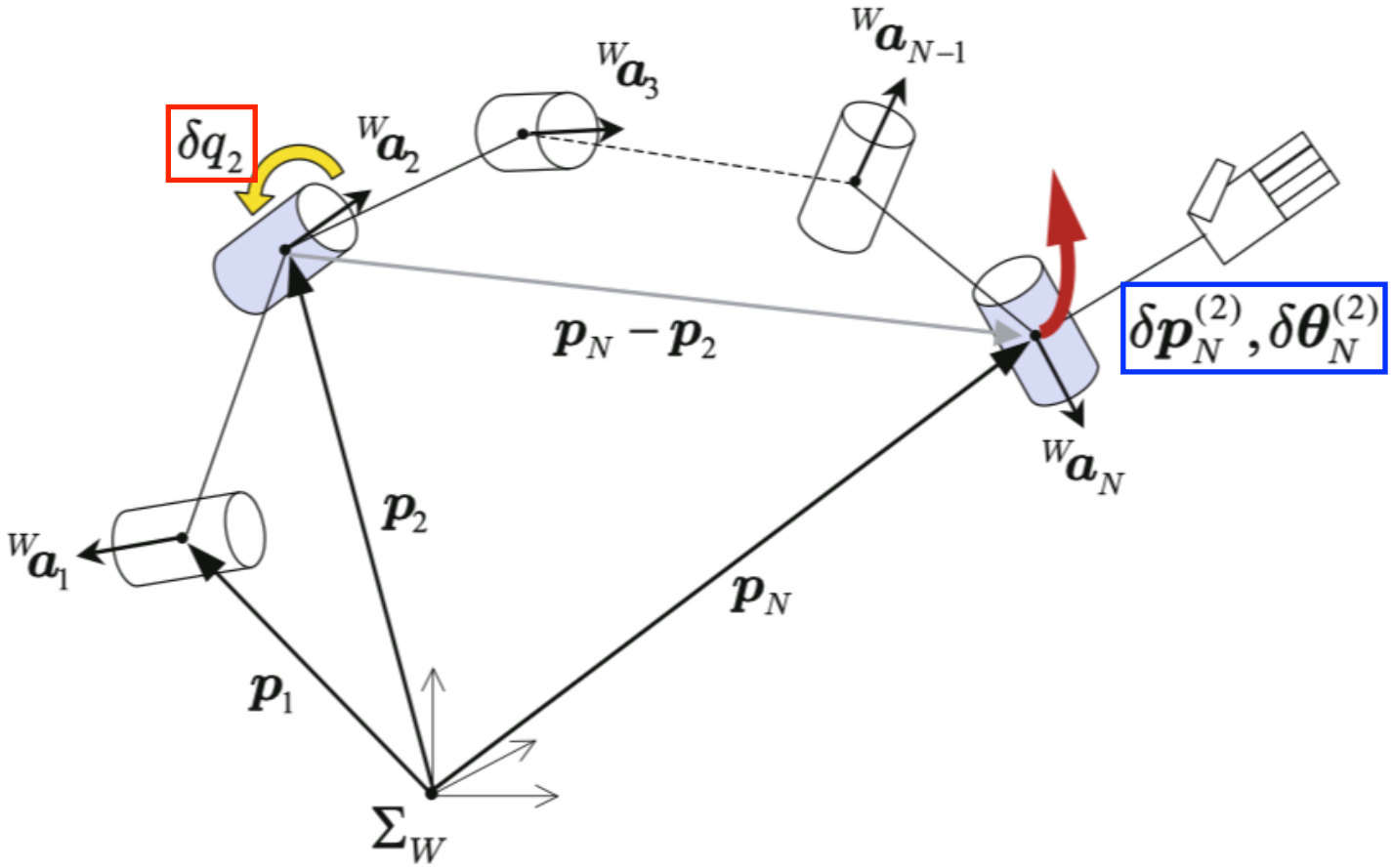
Here $J_{ij}, (i, j = 1 \dots 6)$ are constants which are defined by the current position and attitude of the robots links. There are 6 because of the number of links in the leg. It is too much to write all the components each time so we will simplify it by

$$\begin{bmatrix} \delta \boldsymbol{p} \\ \delta \boldsymbol{\theta} \end{bmatrix} = \boldsymbol{J}\ \delta \boldsymbol{q}. \tag{2.70}$$

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

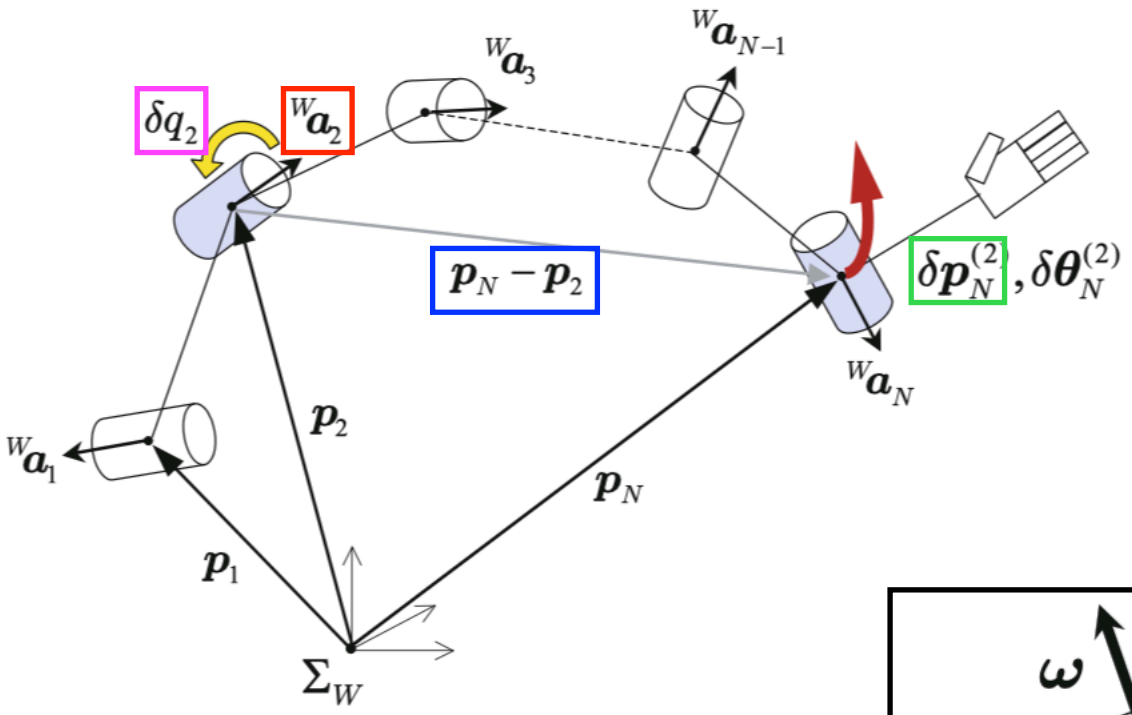
Jacobian Matrix

ROBOT INTELLIGENCE LAB



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

Jacobian Matrix

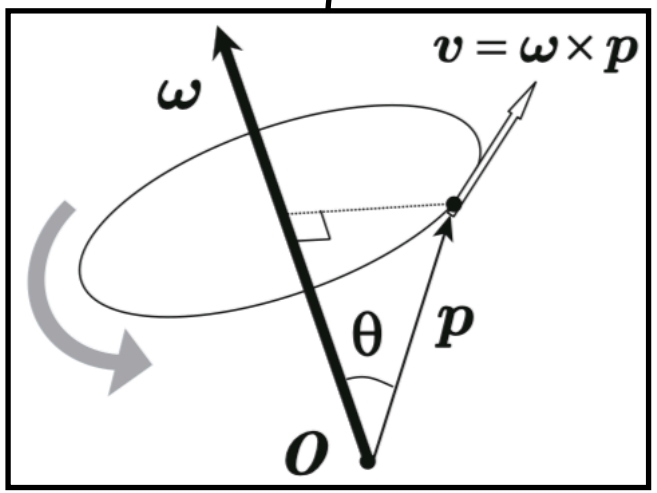


In this chain of links, let us assume we kept all the joints fixed except for the 2nd joint, which we turned by a small angle, δq_2. The amount of the end effector (link N)'s position changed($\delta\boldsymbol{p}_N^{(2)}$), and the amount the attitude of the end effector changed ($\delta\boldsymbol{\theta}_N^{(2)}$)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

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

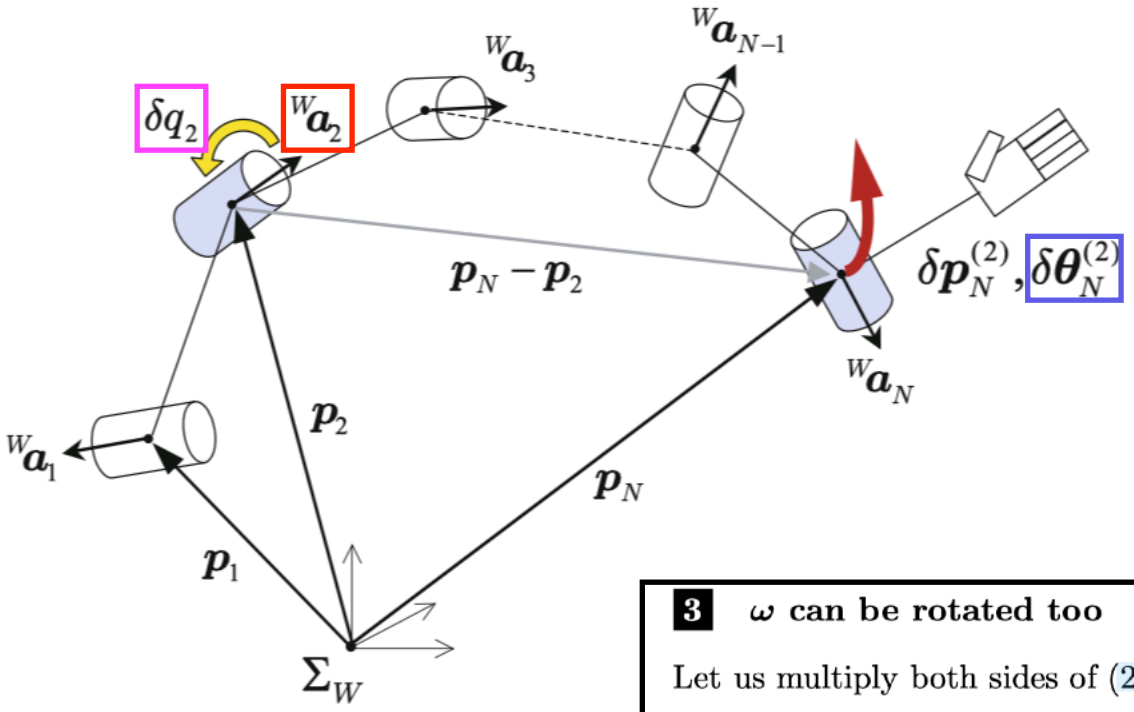
where ${}^W\boldsymbol{a}_2$ is the unit vector for the second joint axis with respect to the world frame

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



Jacobian Matrix

ROBOT INTELLIGENCE LAB



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($\delta \boldsymbol{p}_N^{(2)}$), and the amount the attitude of the end effector changed ($\delta \boldsymbol{\theta}_N^{(2)}$) can be calculated by

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

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

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

3 **ω can be rotated too**

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

$$\boldsymbol{R}\boldsymbol{\omega} = \boldsymbol{R}\boldsymbol{a}\dot{q}. \tag{2.20}$$

If we introduce following new vectors,

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

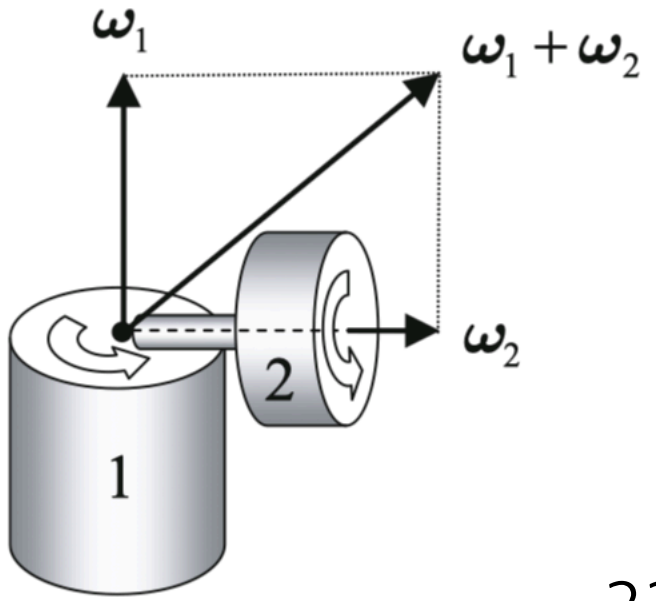
then we can rewrite (2.20) as

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

Additivity of v and ω

ROBOT INTELLIGENCE LAB

Both directional and angular velocities can be **added**



Jacobian Matrix

ROBOT INTELLIGENCE LAB

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

$$\begin{cases} \delta \boldsymbol{p}_N = \sum_{j=1}^{N} \delta \boldsymbol{p}_N^{(j)} \\ \delta \boldsymbol{\theta}_N = \sum_{j=1}^{N} \delta \boldsymbol{\theta}_N^{(j)} \end{cases}. \tag{2.72}$$

We can rewrite the above as a matrix to obtain

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

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

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

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

Both directional and angular velocities are linear functions w.r.t. joint velocity

Jacobian Matrix

ROBOT INTELLIGENCE LAB

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

additivity of directional velocity
additivity of angular velocity

$$\begin{cases} \delta \boldsymbol{p}_N = \sum_{j=1}^{N} \delta \boldsymbol{p}_N^{(j)} \\ \delta \boldsymbol{\theta}_N = \sum_{j=1}^{N} \delta \boldsymbol{\theta}_N^{(j)} \end{cases} . \tag{2.72}$$

We can rewrite the above as a matrix to obtain

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

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

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

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

Jacobian Matrix

ROBOT INTELLIGENCE LAB

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

$$\begin{cases} \delta \boldsymbol{p}_N = \sum_{j=1}^{N} \delta \boldsymbol{p}_N^{(j)} \\ \delta \boldsymbol{\theta}_N = \sum_{j=1}^{N} \delta \boldsymbol{\theta}_N^{(j)} \end{cases}. \tag{2.72}$$

We can rewrite the above as a matrix to obtain

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

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

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

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

Jacobian matrix maps joint velocities to directional and angualr velocities of a certain joint

(Numerical) Inverse Kinematics

ROBOT INTELLIGENCE LAB

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

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

$$\boldsymbol{e} := \dot{\boldsymbol{x}} - \boldsymbol{J}\dot{\boldsymbol{q}}. \tag{2.78}$$

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

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

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

$$\frac{\partial E(\dot{\boldsymbol{q}})}{\partial \dot{\boldsymbol{q}}} = 0. \tag{2.80}$$

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

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

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

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

Unfortunately, we cannot use this at singularity because,

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

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

IK with Singularity Robustness

ROBOT INTELLIGENCE LAB

Now, we slightly modify the cost function.

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

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

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

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

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

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

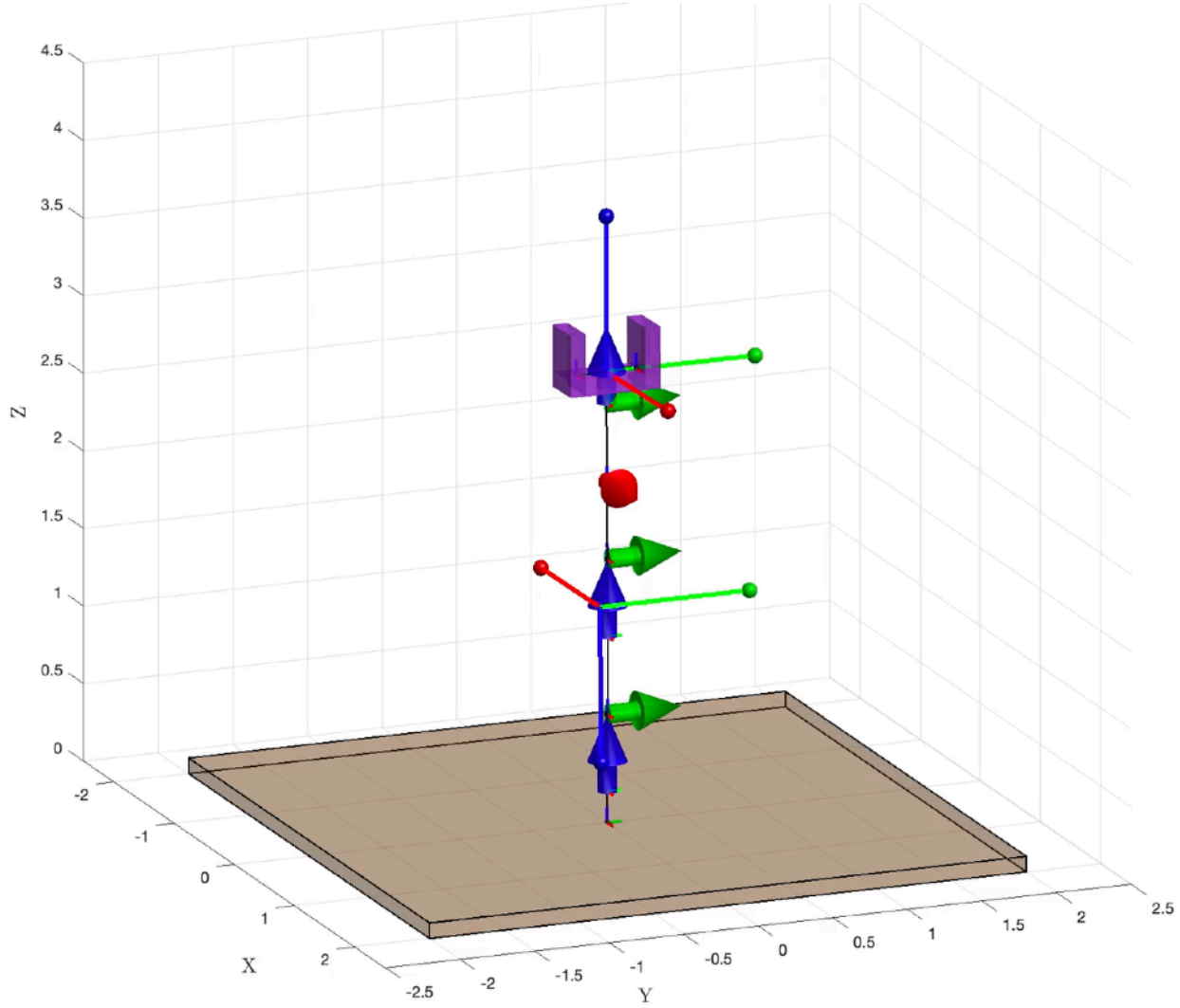
$$\begin{aligned} \dot{\boldsymbol{q}} &= \boldsymbol{J}^{\#\lambda}\dot{\boldsymbol{x}} \\ \boldsymbol{J}^{\#\lambda} &:= (\boldsymbol{J}^T\boldsymbol{J} + \lambda\boldsymbol{E})^{-1}\boldsymbol{J}^T. \end{aligned} \tag{2.86}$$

The matrix $\boldsymbol{J}^{\#\lambda}$ is called an SR inverse[21].

*SR: singularity-Robust

IK in Action

ROBOT INTELLIGENCE LAB



To compute Jacobian Matrix

ROBOT INTELLIGENCE LAB

Target joint (e.g., end-effector) position

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

To compute Jacobian Matrix

ROBOT INTELLIGENCE LAB

Revolute joint axis in World coordinate

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

To compute Jacobian Matrix

ROBOT INTELLIGENCE LAB

Location of each revolute joint in World coordinate

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

IK in Action

ROBOT INTELLIGENCE LAB

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

```
% 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_control' 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

ROBOT INTELLIGENCE LAB

Now, we slightly modify the cost function.

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

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

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

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

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

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

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

The matrix $\boldsymbol{J}^{\#\lambda}$ is called an SR inverse[21].

```
% 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

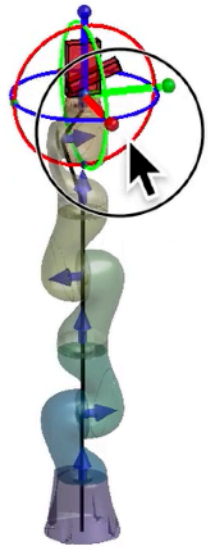
```
% Initialize a kinematic chain
chain = init_chain('name','kinematic_chain');
% Add joint to the chain
chain = add_joint_to_chain(chain,'name','world');
chain = add_joint_to_chain(chain,'name','J1','parent_name','world',...
    'p_offset',cv([0,0,0.5]),'a',cv([0,0,1]));
chain = add_joint_to_chain(chain,'name','J2','parent_name','J1',...
    '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 offs

    % 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 appe
    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

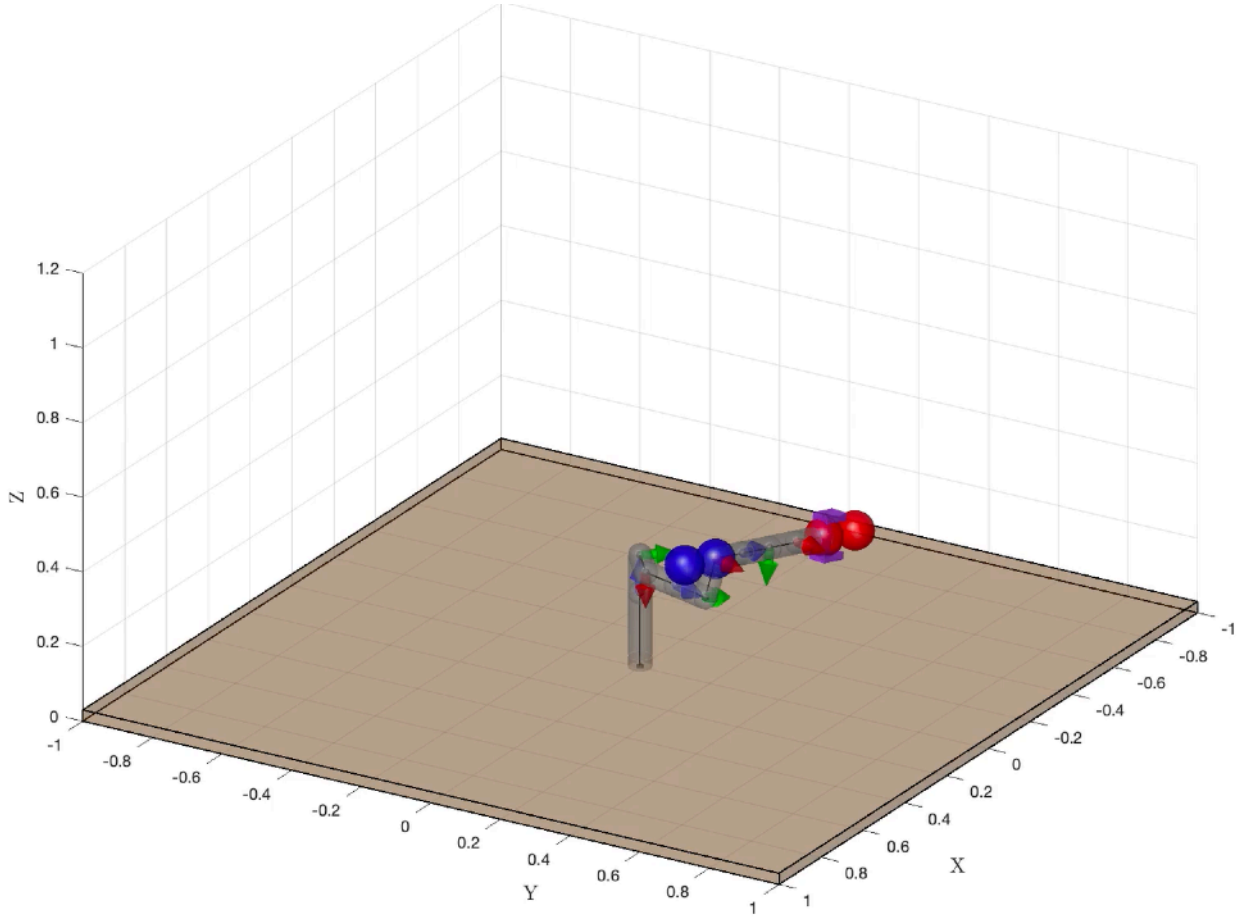
ROBOT INTELLIGENCE LAB

[Consider Joint Limit] Tick:[4495] IK error:[0.011]
(Press [t]:toggle [q]:quit)



Homework

ROBOT INTELLIGENCE LAB



Implement IK with two different target joints.

(Hint: Augmented Jacobian Method)

Thank You