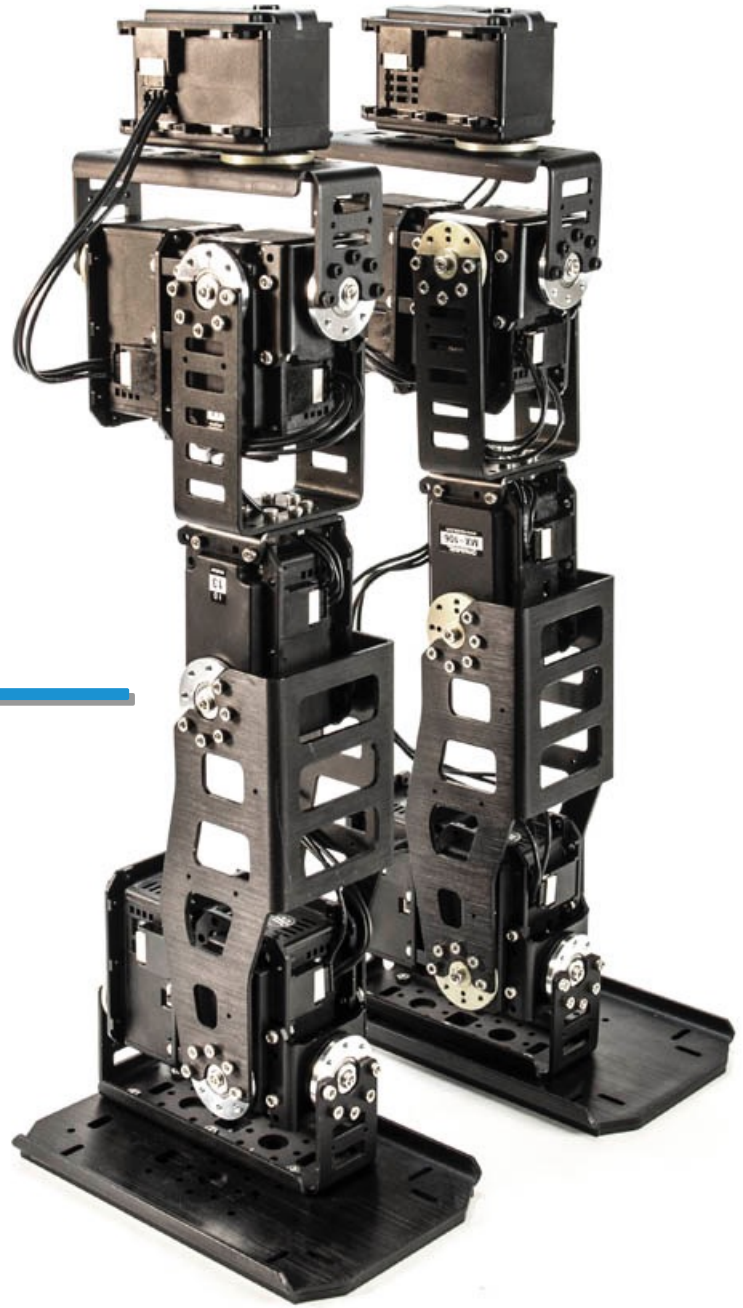
UNIVERSITY OF PLYMOUTH

ROCO318 Mobile and Humanoid Robots



Humanoid Kinematics

Dr Mario Gianni
School of Engineering, Computing and Mathematics
Faculty of Science and Engineering
University of Plymouth

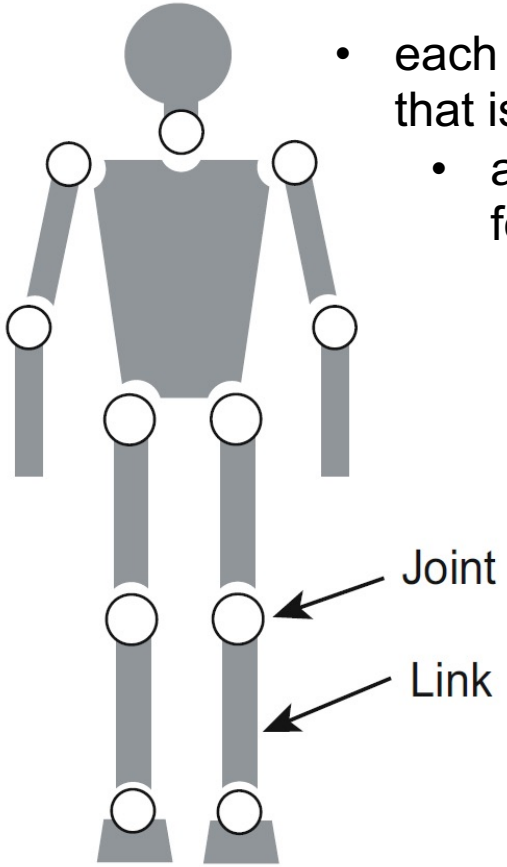
Lecture Content

- Representation of links and joints
- Tree structures and connection rules
- Data Structures in C++
- Forward Kinematics
  - Implementation
- Inverse Kinematics
  - Implementation
- Singular postures
  - Identification methods from linear algebra
- Singularity Robustness
  - Levenberg-Marquardt Method

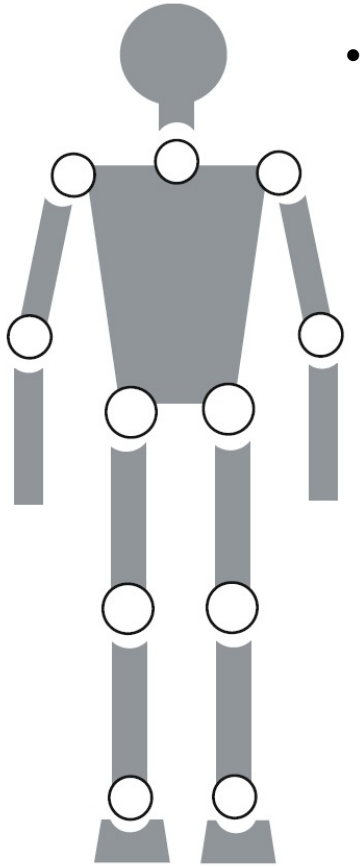
UNIVERSITY OF PLYMOUTH

Representation

- Two approaches to represent the mechanical structure of a humanoid robot



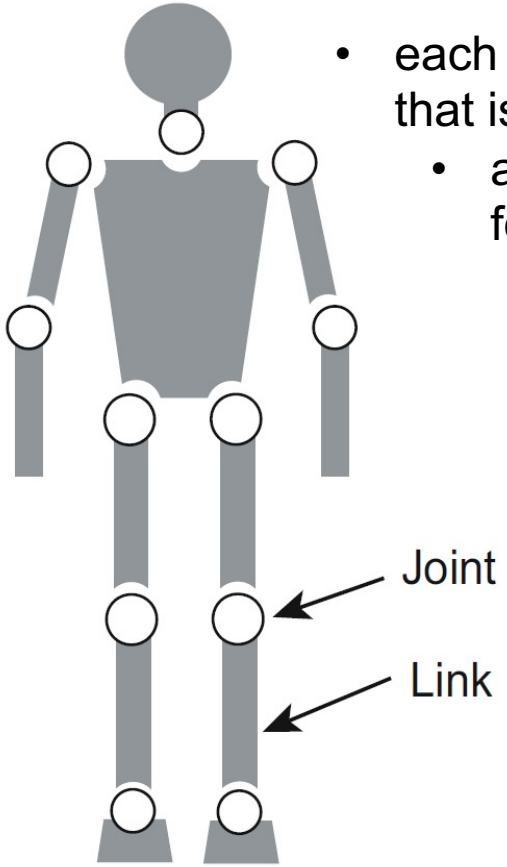
- each joint gets included in the link that is further away from the trunk
  - all links have one joint except for the trunk.



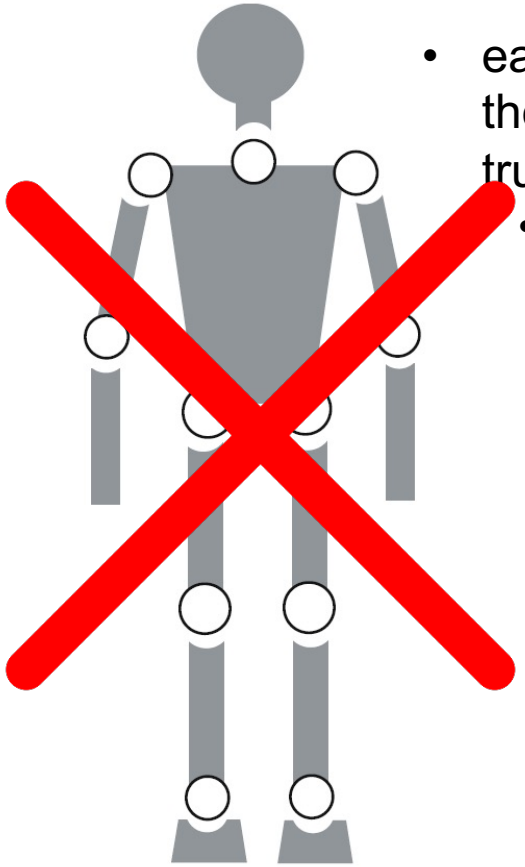
- each joint is defined as a part of the trunk or the link closer to the trunk.
  - the number of joints in each link differs with each joint

Representation

- Two approaches to represent the mechanical structure of a humanoid robot



- each joint gets included in the link that is further away from the trunk
  - all links have one joint except for the trunk.



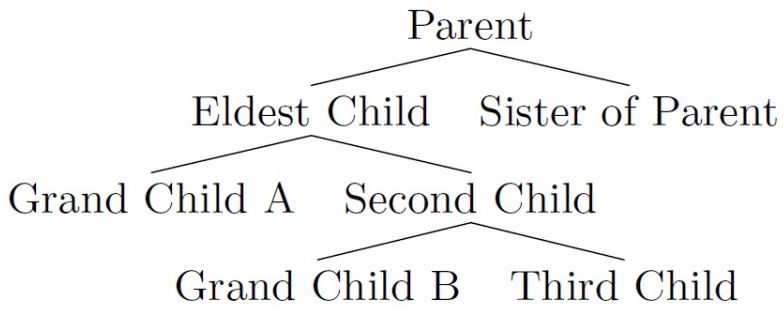
- each joint is defined as a part of the trunk or the link closer to the trunk.
  - the number of joints in each link differs with each joint

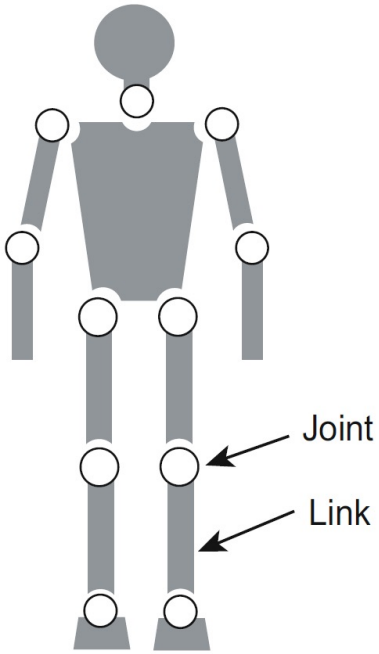
- More convenient from a programming perspective
  - same algorithm to deal with all the joints as all links include just one joint
  - Easy to extend with new joints

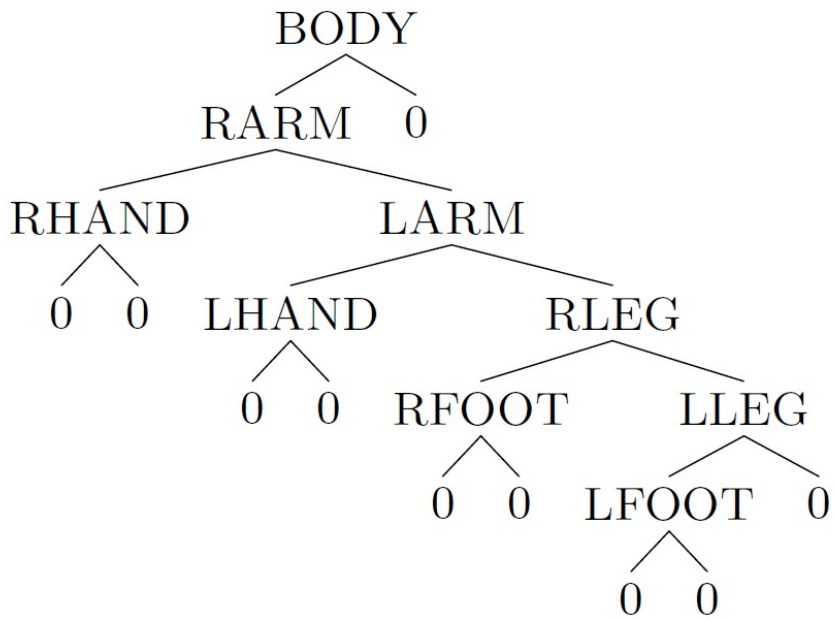
Tree Structure

- Connecting links together forms a tree structure
  - How to connect the links?

(Connecting rule)







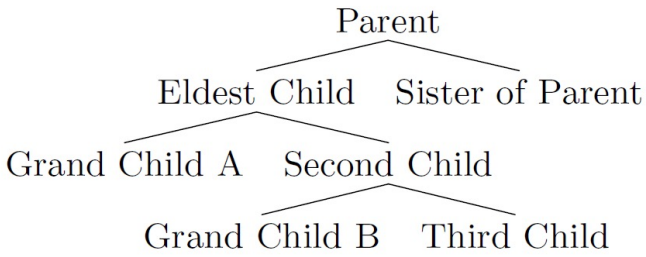
- each link has two connections so that the left lower line connects to its child link and the right lower line connects to its sister link, usually called “sibling”
- For example, if you want to know the sister links of RARM, you go along the lower right connections and gain the links, LARM, RLEG and LLEG.
- The parent of these links will be the upper right BODY link and the child will be the lower left RHAND link.
- 0 is used for no child or sibling links (-1 in practise)

UNIVERSITY OF PLYMOUTH

C++ Data Structure Representing Links

```
#ifndef LINK_DATA_H_
#define LINK_DATA_H_

#include <eigen3/Eigen/Eigen>
#include "robotis_math/robotis_math.h"

namespace robotis_legs
{
     class LinkData
     {
      public:
       LinkData();
       ~LinkData();

       std::string name_; int parent_;  int sibling_;  int child_;

       double mass_;

       Eigen::MatrixXd relative_position_;
       Eigen::MatrixXd joint_axis_;
       Eigen::MatrixXd center_of_mass_;
       Eigen::MatrixXd inertia_;

       double joint_limit_max_; double joint_limit_min_;

       double joint_angle_;  double joint_velocity_;  double joint_acceleration_;

       Eigen::MatrixXd position_; Eigen::MatrixXd orientation_;  Eigen::MatrixXd transformation_;
     };
```

(Connecting rule)



UNIVERSITY OF PLYMOUTH

C++ Data Structure Representing Links

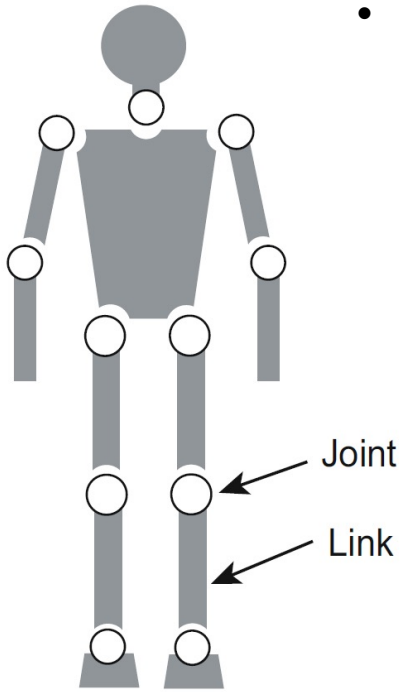
```
for (int id = 0; id <= ALL_JOINT_ID; id++)
    legs_link_data_[id] = new LinkData();
```

```
// left leg knee pitch
legs_link_data_[14]->name_ = "l_knee";
legs_link_data_[14]->parent_ = 12; // Index of the array related to l_hip_pitch link
legs_link_data_[14]->sibling_ = -1;
legs_link_data_[14]->child_ = 16;
…
```

```
// left leg ankle pitch
legs_link_data_[16]->name_ = "l_ank_pitch";
legs_link_data_[16]->parent_ = 14;
legs_link_data_[16]->sibling_ = -1;
legs_link_data_[16]->child_ = 18;
…
```

```
// left leg ankle roll
legs_link_data_[18]->name_ = "l_ank_roll";
legs_link_data_[18]->parent_ = 16;
legs_link_data_[18]->sibling_ = -1;
legs_link_data_[18]->child_ = 30; // Index of the array related to l_leg_end link
…
```

Recursion



- More convenient from a programming perspective, Why?
  - For example, write a function in C++ which sums up all the masses of the link

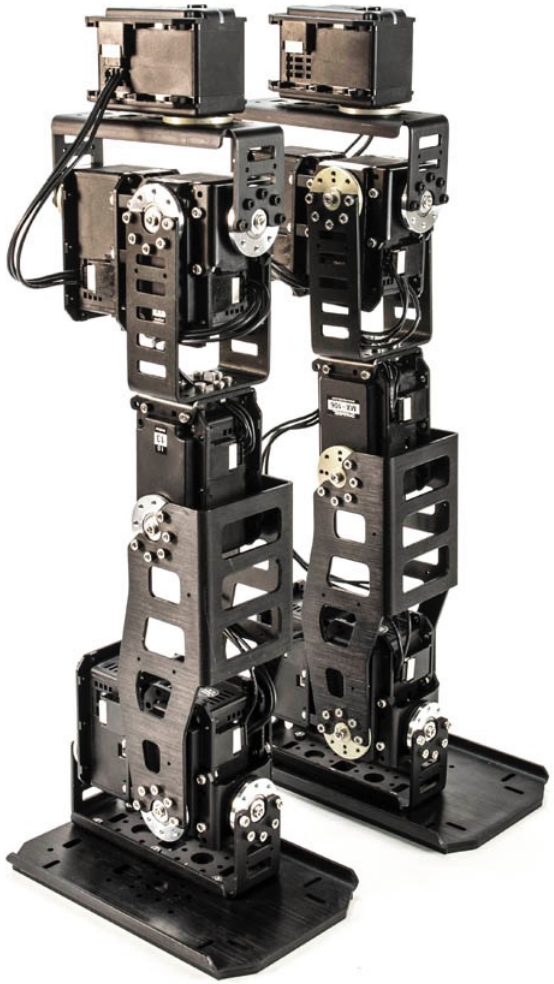
```
double LegsKinematicsDynamics::calcTotalMass(int joint_id)
{
  double mass;

  if (joint_id == -1)
    mass = 0.0;
  else
    mass = legs_link_data_[joint_id]->mass_
           + calcTotalMass(legs_link_data_[joint_id]->sibling_)
           + calcTotalMass(legs_link_data_[joint_id]->child_);

  return mass;
}
```

If the ID number is not -1, then it will return the sum of the mass of the corresponding link, the total mass of the descendant links from the sister, and the total mass of the descendant links from the child.

- Calling a function from inside it is named a **recursive call**.
  - Each time this function is called you move downward in the tree until you reach a node which has neither sister nor child.
  - This gives you a simple way to visit each node in the tree and to apply a desired computation.

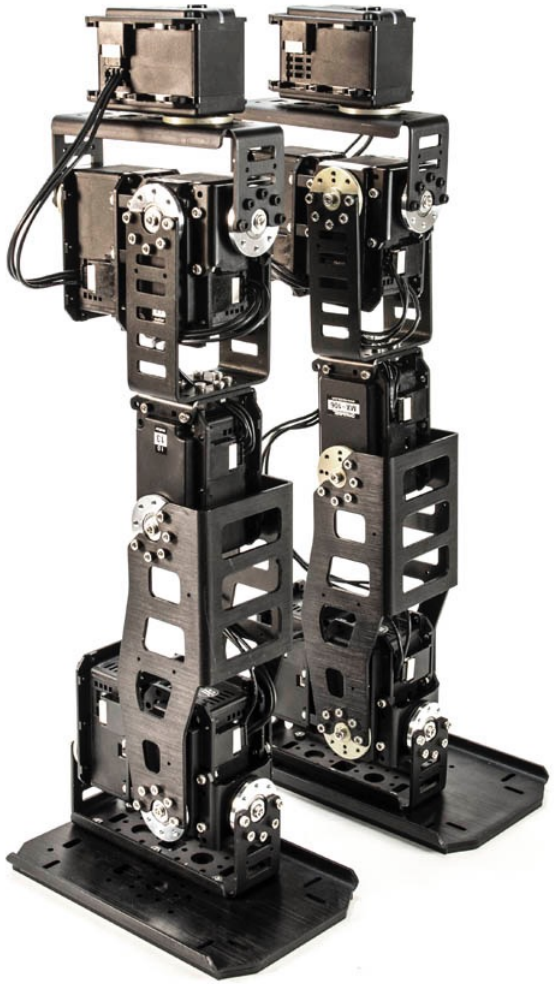
The Humanoid Kinematic Model

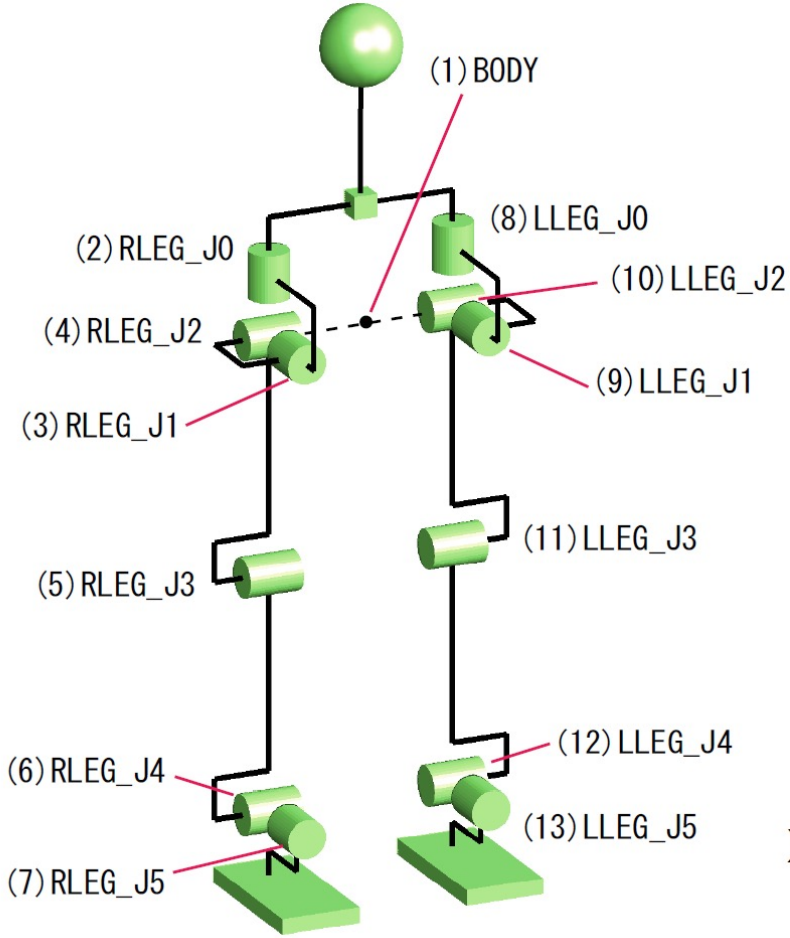
- 12 DOF model
  - It consists of two legs

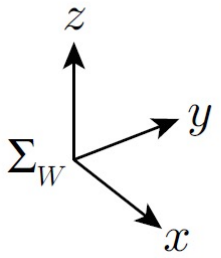


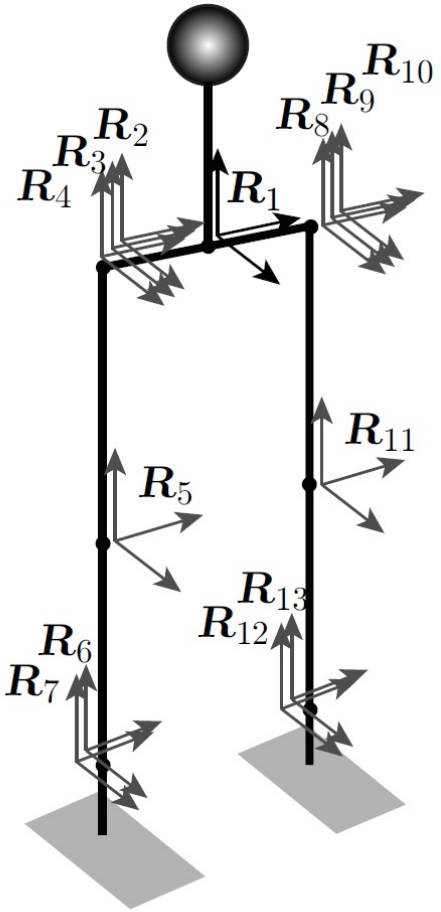
The Humanoid Kinematic Model

- 12 DOF model
  - It consists of two legs







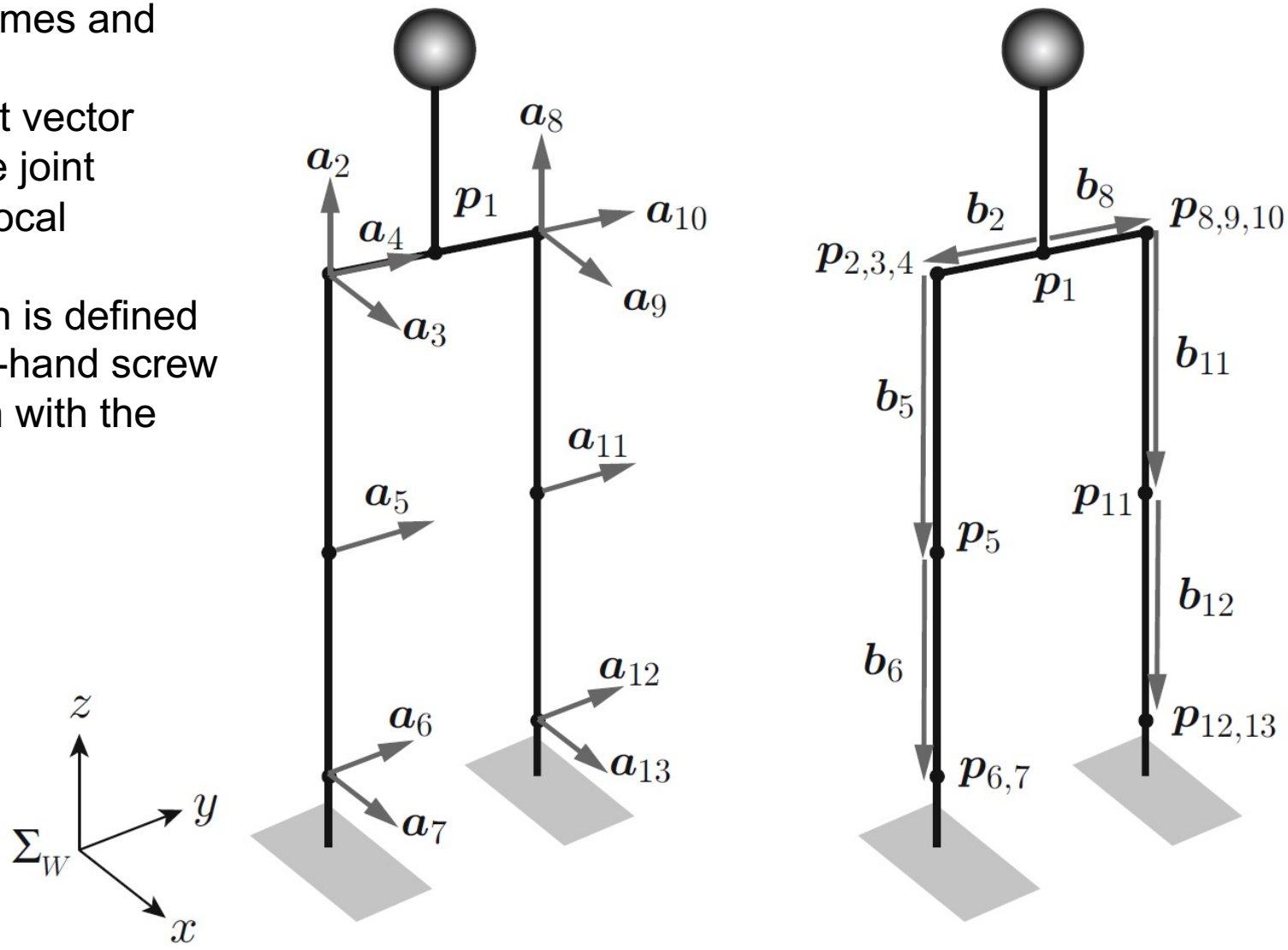


- First step:
  - Define the local coordinates for each link

UNIVERSITY OF PLYMOUTH

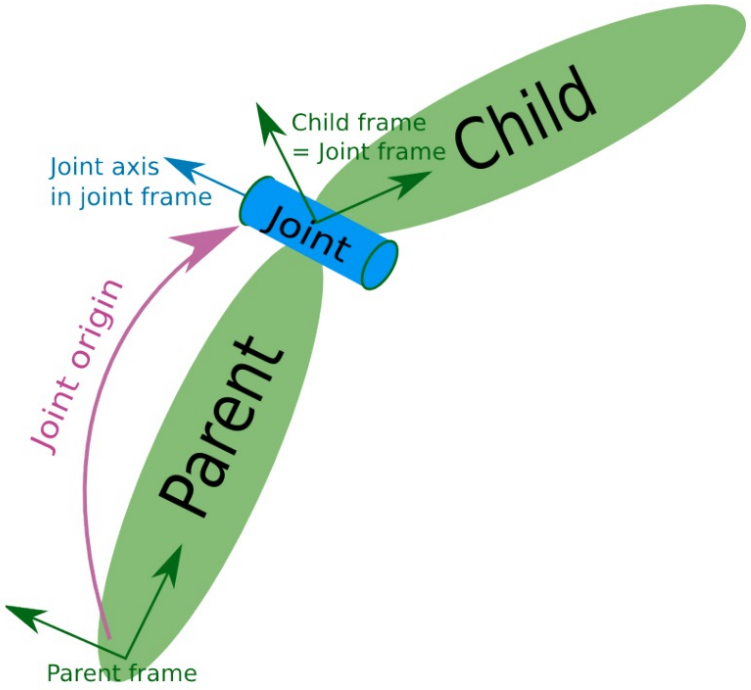
The Humanoid Kinematic Model

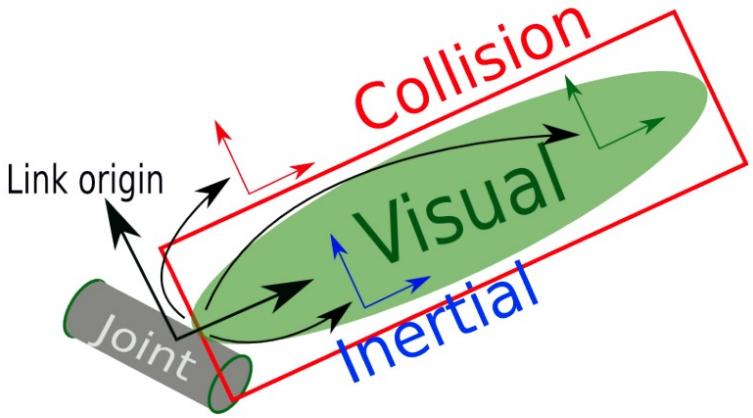
- Second step:
  - Define relative position of the frames and the joint axis vectors
    - The joint axis vector is a unit vector which defines the axis of the joint rotation in the parent link's local coordinates.
    - The positive (+) joint rotation is defined as the way to tighten a right-hand screw placed in the same direction with the joint axis vector.



URDF and XACRO

- URDF
  - Unified Robot Description Format
  - Kinematic and basic physics description of a robot
  - XML format
    - Tags: link, joint, transmission
    - Kinematic tree structure





UNIVERSITY OF PLYMOUTH

URDF and XACRO

- URDF
  - Link element

```
<!-- left hip yaw link -->
  <link name="l_hip_yaw_link">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://op3_description/meshes/ll1.stl" scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="Grey">
        <color rgba="0.60 0.60 0.60 1.0"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://op3_description/meshes/ll1.stl" scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="Grey">
        <color rgba="0.60 0.60 0.60 1.0"/>
      </material>
    </collision>
    <inertial>
      <origin xyz="-0.00157 0.00000 -0.00774"/>
      <mass value="0.01181"/>
      <inertia ixx="0.00000151" ixy="0.0" ixz="0.00000001" iyy="0.00000430" iyz="0.0" izz="0.00000412"/>
    </inertial>
  </link>
```

name

URDF and XACRO

- URDF
  - Link element

Child element visual:

- visual description of the link
- geometry primitives (box, cylinder, sphere)
- geometry meshes (resources stl/dae)
- origin: placement relatively to link reference frame
- material

```
<!-- left hip yaw link -->
  <link name="l_hip_yaw_link">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://op3_description/meshes/ll1.stl" scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="Grey">
        <color rgba="0.60 0.60 0.60 1.0"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://op3_description/meshes/ll1.stl" scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="Grey">
        <color rgba="0.60 0.60 0.60 1.0"/>
      </material>
    </collision>
    <inertial>
      <origin xyz="-0.00157 0.00000 -0.00774"/>
      <mass value="0.01181"/>
      <inertia ixx="0.00000151" ixy="0.0" ixz="0.00000001" iyy="0.00000430" iyz="0.0" izz="0.00000412"/>
    </inertial>
  </link>
```

URDF and XACRO

- URDF
  - Link element

Physics and collision description:

- child element collision
  - similar to visual description of the link
  - mesh resolution should be low

```
<!-- left hip roll link -->
<link name="l_hip_roll_link">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://op3_description/meshes/ll2.stl" scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="Grey">
        <color rgba="0.60 0.60 0.60 1.0"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://op3_description/meshes/ll2.stl" scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="Grey">
        <color rgba="0.60 0.60 0.60 1.0"/>
      </material>
    </collision>
    <inertial>
      <origin xyz="0.00388 0.00028 -0.01214"/>
      <mass value="0.17886"/>
      <inertia ixx="0.00004661" ixy="-0.00000101" ixz="-0.00000131" iyy="0.00012523" iyz="-0.00000006" izz="0.00010857"/>
    </inertial>
  </link>
```

URDF and XACRO

- URDF
  - Link element

Physics and collision description:
- child element inertial
  - centre of mass
  - mass
  - inertia matrix

```
<!-- left hip roll link -->
<link name="l_hip_roll_link">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://op3_description/meshes/ll2.stl" scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="Grey">
        <color rgba="0.60 0.60 0.60 1.0"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://op3_description/meshes/ll2.stl" scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="Grey">
        <color rgba="0.60 0.60 0.60 1.0"/>
      </material>
    </collision>
    <inertial>
      <origin xyz="0.00388 0.00028 -0.01214"/>
      <mass value="0.17886"/>
      <inertia ixx="0.00004661" ixy="-0.00000101" ixz="-0.00000131" iyy="0.00012523" iyz="-0.00000006" izz="0.00010857"/>
    </inertial>
  </link>
```

URDF and XACRO

- URDF
  - Joint element
    - Robot joint between two links

```
<!-- left hip roll joint -->
  <joint name="l_hip_roll" type="revolute">
    <parent link="l_hip_yaw_link"/>
    <child link="l_hip_roll_link"/>
    <origin rpy="0 0 0" xyz="-0.024 0.0 -0.0285"/>
    <axis xyz="-1 0 0"/>
    <limit effort="1000" lower="-2.82743338823" upper="2.82743338823" velocity="100"/>
    <dynamics damping="0.1" friction="0.0"/>
  </joint>
```

- Syntax:
  - Name and type: continuous , fixed, revolute, prismatic, planar, floating

URDF and XACRO

- URDF
  - Joint element
    - Robot joint between two links

```
<!-- left hip roll joint -->
  <joint name="l_hip_roll" type="revolute">
    <parent link="l_hip_yaw_link"/>
    <child link="l_hip_roll_link"/>
    <origin rpy="0 0 0" xyz="-0.024 0.0 -0.0285"/>
    <axis xyz="-1 0 0"/>
    <limit effort="1000" lower="-2.82743338823" upper="2.82743338823" velocity="100"/>
    <dynamics damping="0.1" friction="0.0"/>
  </joint>
```

- Syntax:
  - Name and type: continuous , fixed, revolute, prismatic, planar, floating
- child element parent

URDF and XACRO

- URDF
  - Joint element
    - Robot joint between two links

```
<!-- left hip roll joint -->
  <joint name="l_hip_roll" type="revolute">
    <parent link="l_hip_yaw_link"/>
    <child link="l_hip_roll_link"/>
    <origin rpy="0 0 0" xyz="-0.024 0.0 -0.0285"/>
    <axis xyz="-1 0 0"/>
    <limit effort="1000" lower="-2.82743338823" upper="2.82743338823" velocity="100"/>
    <dynamics damping="0.1" friction="0.0"/>
  </joint>
```

- Syntax:
  - Name and type: continuous , fixed, revolute, prismatic, planar, floating
- child element parent
- child element child

URDF and XACRO

- URDF
  - Joint element
    - Robot joint between two links

```
<!-- left hip roll joint -->
  <joint name="l_hip_roll" type="revolute">
    <parent link="l_hip_yaw_link"/>
    <child link="l_hip_roll_link"/>
    <origin rpy="0 0 0" xyz="-0.024 0.0 -0.0285"/>
    <axis xyz="-1 0 0"/>
    <limit effort="1000" lower="-2.82743338823" upper="2.82743338823" velocity="100"/>
    <dynamics damping="0.1" friction="0.0"/>
  </joint>
```

- Syntax:
  - Name and type: continuous , fixed, revolute, prismatic, planar, floating
- child element parent
- child element child
- child element origin
  - always in parent reference frame

URDF and XACRO

- URDF
  - Joint element
    - Robot joint between two links

```
<!-- left hip roll joint -->
  <joint name="l_hip_roll" type="revolute">
    <parent link="l_hip_yaw_link"/>
    <child link="l_hip_roll_link"/>
    <origin rpy="0 0 0" xyz="-0.024 0.0 -0.0285"/>
    <axis xyz="-1 0 0"/>
    <limit effort="1000" lower="-2.82743338823" upper="2.82743338823" velocity="100"/>
    <dynamics damping="0.1" friction="0.0"/>
  </joint>
```

- Syntax:
  - Name and type: continuous , fixed, revolute, prismatic, planar, floating
- child element parent
- child element child
- child element origin
  - always in parent reference frame
- child element axis
  - for prismatic and revolute
  - in local joint reference frame

URDF and XACRO

- URDF
  - Joint element
    - Robot joint between two links

```
<!-- left hip roll joint -->
  <joint name="l_hip_roll" type="revolute">
    <parent link="l_hip_yaw_link"/>
    <child link="l_hip_roll_link"/>
    <origin rpy="0 0 0" xyz="-0.024 0.0 -0.0285"/>
    <axis xyz="-1 0 0"/>
    <limit effort="1000" lower="-2.82743338823" upper="2.82743338823" velocity="100"/>
    <dynamics damping="0.1" friction="0.0"/>
  </joint>
```

- Physical limits, and dynamic properties
  - child element limit
    - lower and upper
    - rotation/translation limits
    - maximum velocity
    - maximum effort
  - child element dynamics
    - friction
    - damping

URDF and XACRO

- URDF
  - Transmission between joint and actuator

```
<transmission name="l_hip_roll_tran">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="l_hip_roll">
      <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
    </joint>
    <actuator name="l_hip_roll_motor">
      <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
      <mechanicalReduction>1</mechanicalReduction>
    </actuator>
  </transmission>
```

- element transmission
  - type
  - joint
  - actuator

URDF and XACRO

- URDF
  - Gazebo setting

```
<!-- imu sensor -->
  <gazebo reference="body_link">
    <gravity>true</gravity>
    <sensor name="imu_sensor" type="imu">
      <always_on>true</always_on>
      <update_rate>100</update_rate>
      <visualize>true</visualize>
      <topic>__default_topic__</topic>
      <plugin filename="libgazebo_ros_imu_sensor.so" name="imu_plugin">
        <topicName>robotis_op3/imu</topicName>
        <bodyName>body_link</bodyName>
        <updateRateHZ>10.0</updateRateHZ>
        <gaussianNoise>0.0</gaussianNoise>
        <xyzOffset>0 0 0</xyzOffset>
        <rpyOffset>0 0 0</rpyOffset>
        <frameName>body_link</frameName>
      </plugin>
      <pose>0 0 0 0 0 0</pose>
    </sensor>
  </gazebo>
```

- element gazebo
  - reference

URDF and XACRO

- URDF
  - Gazebo setting

```
<!-- imu sensor -->
  <gazebo reference="body_link">
    <gravity>true</gravity>
    <sensor name="imu_sensor" type="imu">
      <always_on>true</always_on>
      <update_rate>100</update_rate>
      <visualize>true</visualize>
      <topic>__default_topic__</topic>
      <plugin filename="libgazebo_ros_imu_sensor.so" name="imu_plugin">
        <topicName>robotis_op3/imu</topicName>
        <bodyName>body_link</bodyName>
        <updateRateHZ>10.0</updateRateHZ>
        <gaussianNoise>0.0</gaussianNoise>
        <xyzOffset>0 0 0</xyzOffset>
        <rpyOffset>0 0 0</rpyOffset>
        <frameName>body_link</frameName>
      </plugin>
      <pose>0 0 0 0 0 0</pose>
    </sensor>
  </gazebo>
```

- element gazebo
  - reference
  - sensors

URDF and XACRO

- URDF
  - Gazebo setting

```
<!-- imu sensor -->
  <gazebo reference="body_link">
    <gravity>true</gravity>
    <sensor name="imu_sensor" type="imu">
      <always_on>true</always_on>
      <update_rate>100</update_rate>
      <visualize>true</visualize>
      <topic>__default_topic__</topic>
      <plugin filename="libgazebo_ros_imu_sensor.so" name="imu_plugin">
        <topicName>robotis_op3/imu</topicName>
        <bodyName>body_link</bodyName>
        <updateRateHZ>10.0</updateRateHZ>
        <gaussianNoise>0.0</gaussianNoise>
        <xyzOffset>0 0 0</xyzOffset>
        <rpyOffset>0 0 0</rpyOffset>
        <frameName>body_link</frameName>
      </plugin>
      <pose>0 0 0 0 0 0</pose>
    </sensor>
  </gazebo>
```

- element gazebo
  - reference
  - sensors
  - plugins

URDF and XACRO

- URDF
  - Gazebo setting

```
<!-- imu sensor -->
  <gazebo reference="body_link">
    <gravity>true</gravity>
    <sensor name="imu_sensor" type="imu">
      <always_on>true</always_on>
      <update_rate>100</update_rate>
      <visualize>true</visualize>
      <topic>__default_topic__</topic>
      <plugin filename="libgazebo_ros_imu_sensor.so" name="imu_plugin">
        <topicName>robotis_op3/imu</topicName>
        <bodyName>body_link</bodyName>
        <updateRateHZ>10.0</updateRateHZ>
        <gaussianNoise>0.0</gaussianNoise>
        <xyzOffset>0 0 0</xyzOffset>
        <rpyOffset>0 0 0</rpyOffset>
        <frameName>body_link</frameName>
      </plugin>
      <pose>0 0 0 0 0 0</pose>
    </sensor>
  </gazebo>
```

- element gazebo
  - reference
  - sensors
  - plugins
  - additional properties (self collide, gravity enable, ...)

URDF and XACRO

- URDF
  - Main limitations
    - Immutability
      - Robot description cannot be changed
      - Robot description read once from parameter server
    - No cycles
      - Joint only have single parent and child
      - Only acyclic, directed graphs (or trees) can be modelled
        - Real-world impact
          - Dual-arm manipulation, parallel grippers
    - Limited sensor elements
    - Low reusability
      - Only a single <robot> tag in URDF
        - Multiple robots can only be modelled as a single chain robot

The solution to some of these problems can be XACRO

URDF and XACRO

- XACRO
  - XML Macro language used for URDF simplification
  - Increase modularity
  - Reduce redundancy
  - Permit Parametrization
  - Generate URDF on-the-fly

- How
  - Inclusion
  - Macros
  - Properties
  - Templates
  - Parameters
  - Expansion of all xacro statements
  - Command line and output to `stdout`

URDF and XACRO

- XACRO

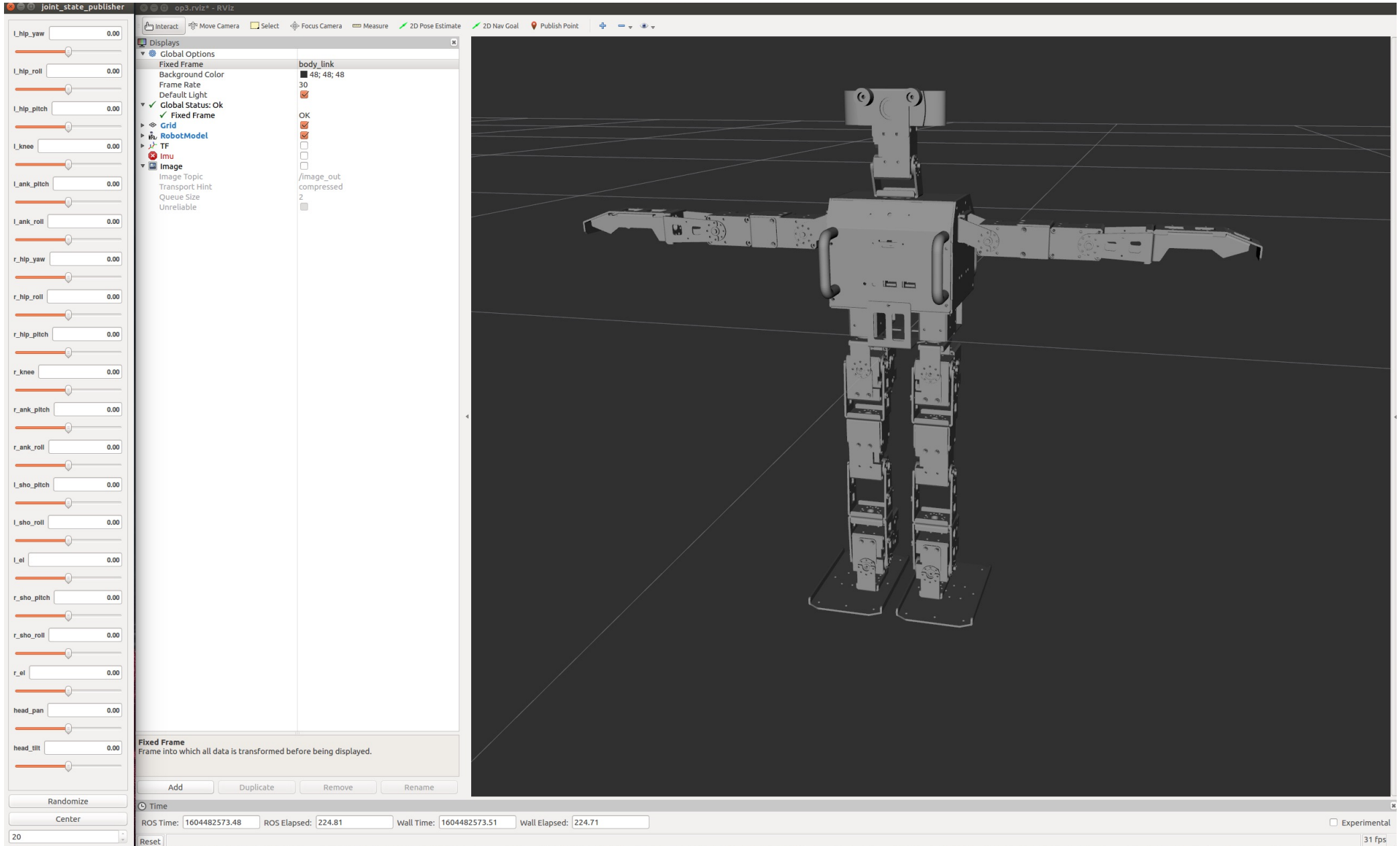
```
...
<xacro:macro name="arm" params="parent arm_name">
   <link name="${arm_name}_link_1"  />
   <joint name="${arm_name}_joint_1" type="..">
      <parent link="${parent}"  />
      <child link='${arm_name}_link_1"  />
   </joint>
</xacro:macro>
...
```

- Every xml elements starts with xacro
  - Name of the macro
  - Parameters (`parent` and `arm_name`)

- To now create a robot with two arms

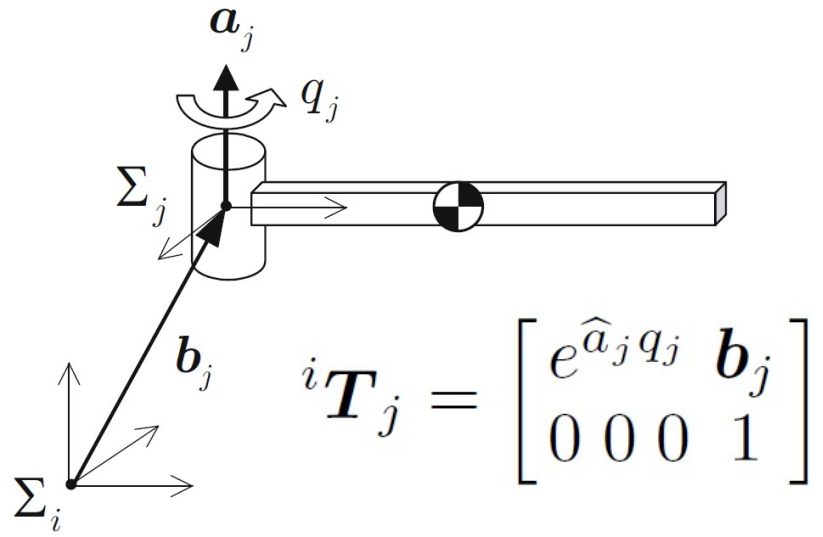
```
...
<link name="torso"  />
...
<xacro:arm parent="torso" arm_name="left"  />
<xacro:arm parent="torso" arm_name="right"  />
...
```

The Humanoid Kinematic Model

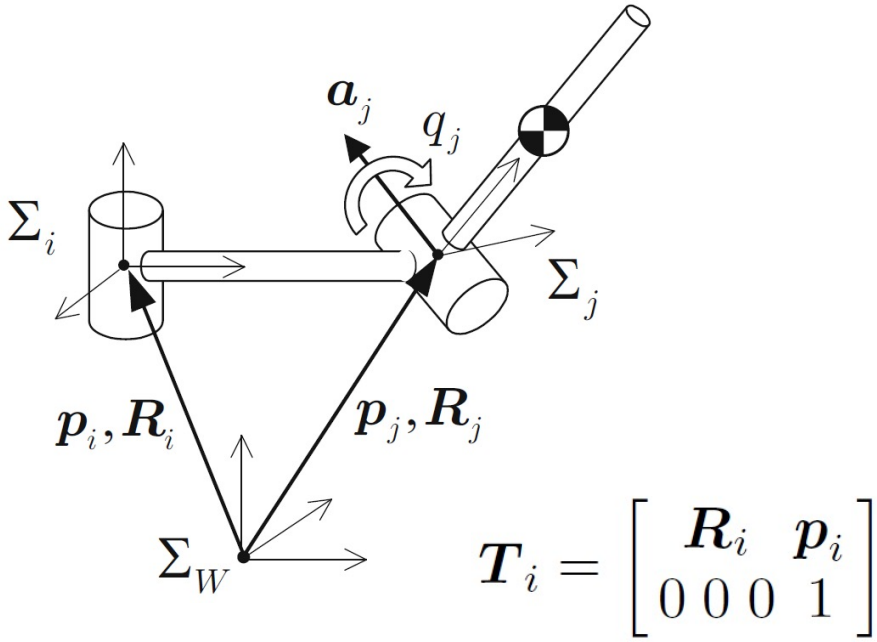


The Humanoid Kinematic Model

- Third step:
  - Forward Kinematics
    - Chain rule of homogeneous transformations



1. Calculate the homogeneous transform of a single link
   - Local coordinate system including origin and joint axis



2. Calculate the homogeneous transforms of two links
   - Absolute position and attitude of the parent link is known
   - Relative position of frame j wrt i calculated in 1)

3. Chain rule

$$T_j = T_i {}^iT_j \longrightarrow \begin{aligned} p_j &= p_i + R_i b_j \\ R_j &= R_i e^{\widehat{a}_j q_j} \end{aligned}$$

Using this relationship and the recursive algorithm, the Forward Kinematics can be performed by an extremely simple script

Forward Kinematics

```
void LegsKinematicsDynamics::calcForwardKinematics(int joint_id)
{
  if (joint_id == -1)
    return;


  if (joint_id == 0)
  {
    legs_link_data_[0]->position_ = Eigen::MatrixXd::Zero(3, 1);
    legs_link_data_[0]->orientation_ = robotis_framework::calcRodrigues(
        robotis_framework::calcHatto(legs_link_data_[0]->joint_axis_), op3_link_data_[0]->joint_angle_);
  }


  if (joint_id != 0)
  {
    int parent = legs_link_data_[joint_id]->parent_;


    legs_link_data_[joint_id]->position_ = legs_link_data_[parent]->orientation_
        * legs_link_data_[joint_id]->relative_position_ + legs_link_data_[parent]->position_;


    legs_link_data_[joint_id]->orientation_ = legs_link_data_[parent]->orientation_
        * robotis_framework::calcRodrigues(robotis_framework::calcHatto(legs_link_data_[joint_id]->joint_axis_),
                                           legs_link_data_[joint_id]->joint_angle_);


    legs_link_data_[joint_id]->transformation_.block<3, 1>(0, 3) = legs_link_data_[joint_id]->position_;
    legs_link_data_[joint_id]->transformation_.block<3, 3>(0, 0) = legs_link_data_[joint_id]->orientation_;
  }


  calcForwardKinematics(legs_link_data_[joint_id]->sibling_);
  calcForwardKinematics(legs_link_data_[joint_id]->child_);
}
```

set the absolute position and attitude
of the base link and all joint angles

Forward Kinematics

```
void LegsKinematicsDynamics::calcForwardKinematics(int joint_id)
{
  if (joint_id == -1)
    return;

  if (joint_id == 0)
  {
    legs_link_data_[0]->position_ = Eigen::MatrixXd::Zero(3, 1);
    legs_link_data_[0]->orientation_ = robotis_framework::calcRodrigues(
        robotis_framework::calcHatto(legs_link_data_[0]->joint_axis_), legs_link_data_[0]->joint_angle_);
  }

  if (joint_id != 0)
  {
    int parent = legs_link_data_[joint_id]->parent_;

    legs_link_data_[joint_id]->position_ = legs_link_data_[parent]->orientation_
        * legs_link_data_[joint_id]->relative_position_ + legs_link_data_[parent]->position_;

    legs_link_data_[joint_id]->orientation_ = legs_link_data_[parent]->orientation_
        * robotis_framework::calcRodrigues(robotis_framework::calcHatto(legs_link_data_[joint_id]->joint_axis_),
                                           legs_link_data_[joint_id]->joint_angle_);

    legs_link_data_[joint_id]->transformation_.block<3, 1>(0, 3) = legs_link_data_[joint_id]->position_;
    legs_link_data_[joint_id]->transformation_.block<3, 3>(0, 0) = legs_link_data_[joint_id]->orientation_;
  }

  calcForwardKinematics(legs_link_data_[joint_id]->sibling_);
  calcForwardKinematics(legs_link_data_[joint_id]->child_);
}
```

$$p_j = p_i + R_i b_j$$

$$R_j = R_i e^{\widehat{a}_j q_j}$$

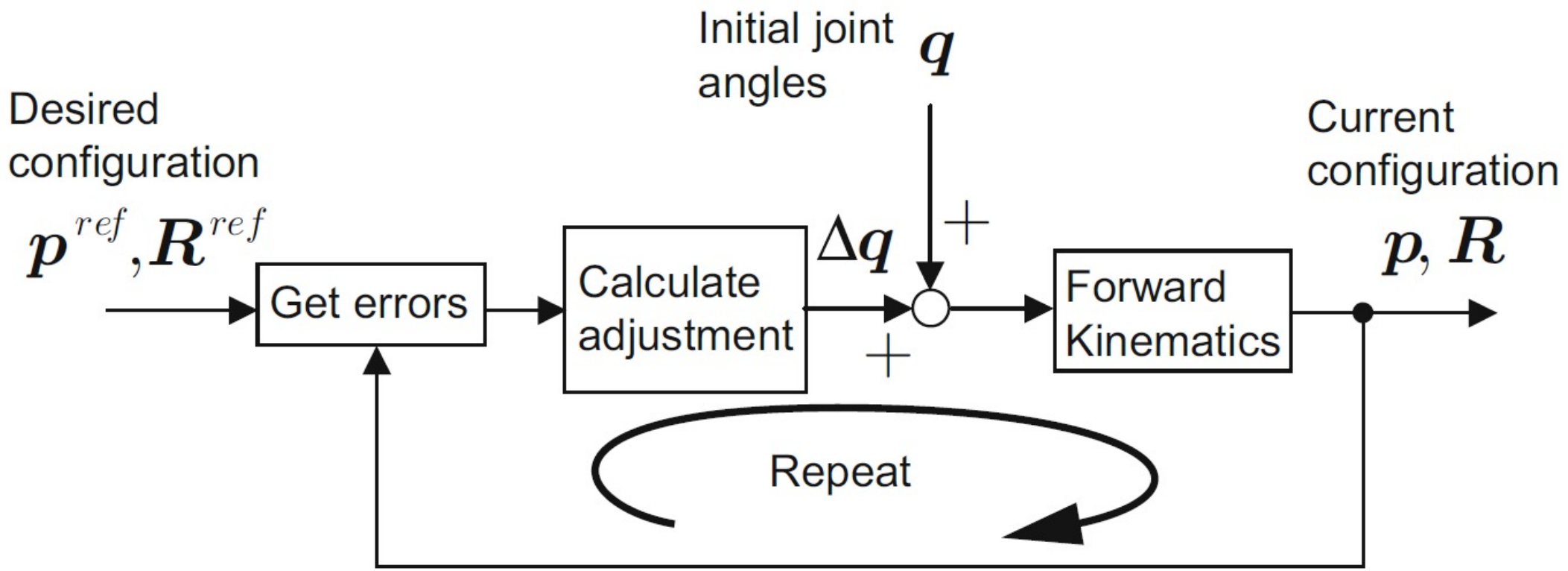
- Rodrigues' formula:
  - It gives you the rotation matrix directly from a constant angular velocity vector

recursive calls

UNIVERSITY OF PLYMOUTH

Inverse Kinematics

- Numerical solution



Inverse Kinematics

- Numerical solution

while $\|\mathbf{x}^d - \mathbf{x}(t)\|^2 < \epsilon$ do

1. Compute the Jacobian $J(\mathbf{q}(t))$ given the configuration of the joints $\mathbf{q}(t)$ at time t

2. Compute the inverse $J^{-1}(\mathbf{q}(t))$ of the Jacobian

3. Compute

$$\Delta\mathbf{q}(t) = J^{-1}(\mathbf{q}(t))\Delta\mathbf{x}$$

with $\Delta\mathbf{x} = \|\mathbf{x}^d - \mathbf{x}(t)\|^2$

4. Compute

$$\mathbf{q}(t+1) = \mathbf{q}(t) + \eta\Delta\mathbf{q}$$

Newton's Method

```
bool LegsKinematicsDynamics::calcInverseKinematics(int to, Eigen::MatrixXd tar_position, Eigen::MatrixXd tar_orientation,
                                                   int max_iter, double ik_err)
{
  bool ik_success = false;  bool limit_success = false;

  std::vector<int> idx = findRoute(to);

  for (int iter = 0; iter < max_iter; iter++)
  {
    Eigen::MatrixXd jacobian = calcJacobian(idx);

    Eigen::MatrixXd curr_position = legs_link_data_[to]->position_;
    Eigen::MatrixXd curr_orientation = legs_link_data_[to]->orientation_;

    Eigen::MatrixXd err = calcVWerr(tar_position, curr_position, tar_orientation, curr_orientation);

    if (err.norm() < ik_err) {
      ik_success = true;
      break;
    }
    else
      ik_success = false;

    Eigen::MatrixXd jacobian_trans = jacobian * jacobian.transpose();
    Eigen::MatrixXd jacobian_inverse = jacobian.transpose() * jacobian_trans.inverse();

    Eigen::MatrixXd delta_angle = jacobian_inverse * err;

    for (int id = 0; id < idx.size(); id++)
    {
      int joint_num = idx[id];
      legs_link_data_[joint_num]->joint_angle_ += delta_angle.coeff(id);
    }

    calcForwardKinematics(0);
  }
```

returns the links that you need to go through to get to the target link from the base link

Newton's Method

```
bool LegsKinematicsDynamics::calcInverseKinematics(int to, Eigen::MatrixXd tar_position, Eigen::MatrixXd tar_orientation,
                                                   int max_iter, double ik_err)
{
  bool ik_success = false;  bool limit_success = false;

  std::vector<int> idx = findRoute(to);

  for (int iter = 0; iter < max_iter; iter++)
  {
    Eigen::MatrixXd jacobian = calcJacobian(idx);

    Eigen::MatrixXd curr_position = legs_link_data_[to]->position_;
    Eigen::MatrixXd curr_orientation = legs_link_data_[to]->orientation_;

    Eigen::MatrixXd err = calcVWerr(tar_position, curr_position, tar_orientation, curr_orientation);

    if (err.norm() < ik_err) {
      ik_success = true;
      break;
    }
    else
      ik_success = false;

    Eigen::MatrixXd jacobian_trans = jacobian * jacobian.transpose();
    Eigen::MatrixXd jacobian_inverse = jacobian.transpose() * jacobian_trans.inverse();

    Eigen::MatrixXd delta_angle = jacobian_inverse * err;

    for (int id = 0; id < idx.size(); id++)
    {
      int joint_num = idx[id];
      legs_link_data_[joint_num]->joint_angle_ += delta_angle.coeff(id);
    }

    calcForwardKinematics(0);
  }
```

$$\Delta \mathbf{x} = \|\mathbf{x}^d - \mathbf{x}(t)\|^2$$

```
Eigen::MatrixXd pos_err = tar_position - curr_position;
Eigen::MatrixXd ori_err = curr_orientation.transpose() * tar_orientation;
Eigen::MatrixXd ori_err_dash = curr_orientation * robotis_framework::convertRotToOmega(ori_err);
```

Transform the rotation matrix into the corresponding angular velocity vector

$$\boldsymbol{\omega} = (\ln \boldsymbol{R})^\vee$$

$$(\ln \boldsymbol{R})^\vee = \begin{cases} [0\ 0\ 0]^T & (\text{if } \boldsymbol{R} = \boldsymbol{E}) \\ \frac{\pi}{2} \begin{bmatrix} r_{11}+1 \\ r_{22}+1 \\ r_{33}+1 \end{bmatrix} & (\text{else if } \boldsymbol{R} \text{ is diagonal}) \\ \theta \frac{\boldsymbol{l}}{\|\boldsymbol{l}\|} & (\text{otherwise}) \end{cases}$$

Newton's Method

```
bool LegsKinematicsDynamics::calcInverseKinematics(int to, Eigen::MatrixXd tar_position, Eigen::MatrixXd tar_orientation,
                                                   int max_iter, double ik_err)
{
  bool ik_success = false;  bool limit_success = false;

  std::vector<int> idx = findRoute(to);

  for (int iter = 0; iter < max_iter; iter++)
  {
    Eigen::MatrixXd jacobian = calcJacobian(idx);

    Eigen::MatrixXd curr_position = legs_link_data_[to]->position_;
    Eigen::MatrixXd curr_orientation = legs_link_data_[to]->orientation_;

    Eigen::MatrixXd err = calcVWerr(tar_position, curr_position, tar_orientation, curr_orientation);

    if (err.norm() < ik_err) {
      ik_success = true;
      break;
    }
    else
      ik_success = false;

    Eigen::MatrixXd jacobian_trans = jacobian * jacobian.transpose();
    Eigen::MatrixXd jacobian_inverse = jacobian.transpose() * jacobian_trans.inverse();

    Eigen::MatrixXd delta_angle = jacobian_inverse * err;

    for (int id = 0; id < idx.size(); id++)
    {
      int joint_num = idx[id];
      legs_link_data_[joint_num]->joint_angle_ += delta_angle.coeff(id);
    }

    calcForwardKinematics(0);
  }
```

$$\Delta\mathbf{q}(t) = J^{-1}(\mathbf{q}(t))\Delta\mathbf{x}$$

$$\mathbf{q}(t+1) = \mathbf{q}(t) + \eta\Delta\mathbf{q}$$

Inverse Kinematics

```
for (int id = 0; id < idx.size(); id++)
  {
    int joint_num = idx[id];


    if (legs_link_data_[joint_num]->joint_angle_ >= legs_link_data_[joint_num]->joint_limit_max_)
    {
      limit_success = false;
      break;
    }
    else if (legs_link_data_[joint_num]->joint_angle_ <= legs_link_data_[joint_num]->joint_limit_min_)
    {
      limit_success = false;
      break;
    }
    else
      limit_success = true;
  }


  if (ik_success == true && limit_success == true)
    return true;
  else
    return false;
}
```

CAUTION! When using this program on a real robot, you need to continuously check whether the joint angles exceed their limits. In the worst case, you could destroy your robot or otherwise cause major injury or death.

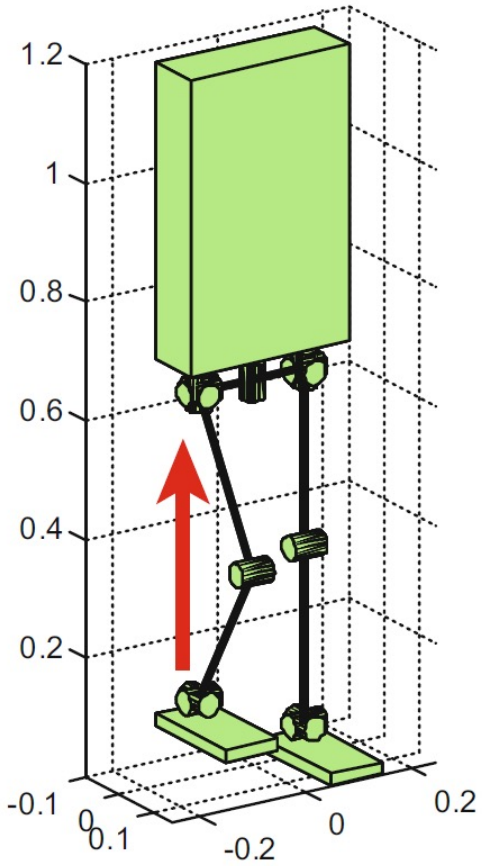
Jacobian and Joint Velocity

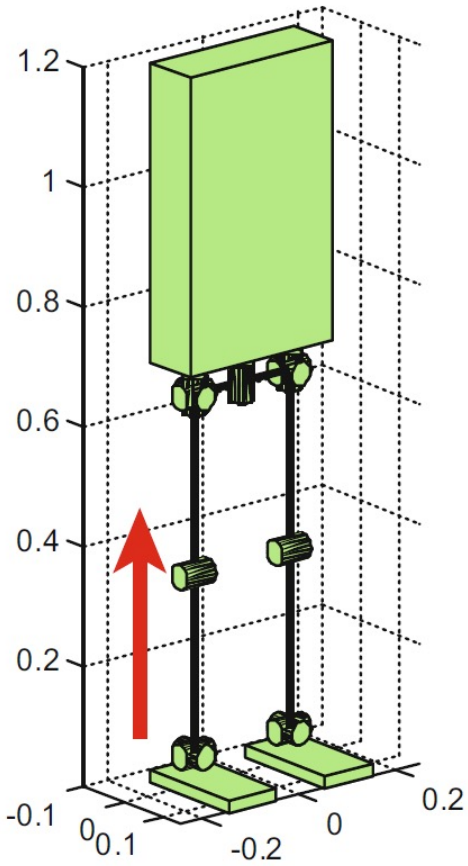
- Relation between the joint speed and the end effector speed

end effector speed → $\dot{\mathbf{x}} = \frac{d}{dt} f(\mathbf{q}) = \frac{d}{dq} f(\mathbf{q}) \frac{d}{dt} \mathbf{q} = J(\mathbf{q}) \dot{\mathbf{q}}$ ← joint speed

- Two initial postures

right hip pitch = -30°
knee pitch = 60°
ankle pitch = -30°



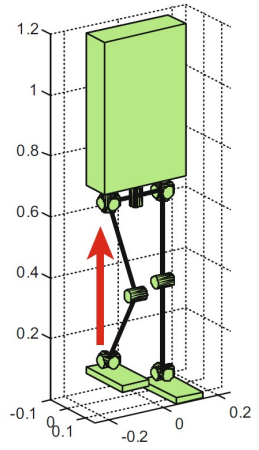


right hip pitch = 0°
knee pitch = 0°
ankle pitch = 0°

UNIVERSITY OF PLYMOUTH

Jacobian and Joint Velocity

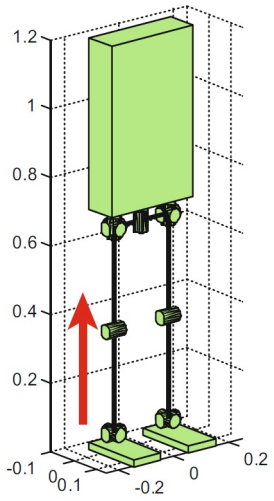
- Suppose we want to let the foot lift vertically with 0.1m/s



right hip pitch = -30°
knee pitch = 60°
ankle pitch = -30°

$$\dot{\boldsymbol{q}} = J^{-1}(\boldsymbol{q})\dot{\boldsymbol{x}}$$

Joint speed:
- hip pitch = -0.33 rad/s
- knee pitch = 0.67 rad/s
- ankle pitch = -0.33 rad/s



right hip pitch = 0°
knee pitch = 0°
ankle pitch = 0°

$$\dot{\boldsymbol{q}} = J^{-1}(\boldsymbol{q})\dot{\boldsymbol{x}}$$

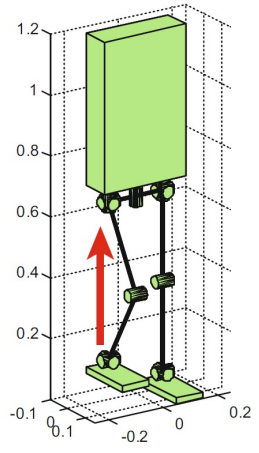
Joint speed:
- hip pitch = NaN
- knee pitch = Nan
- ankle pitch = NaN

Matrix is singular to working precision

UNIVERSITY OF PLYMOUTH

Jacobian and Joint Velocity

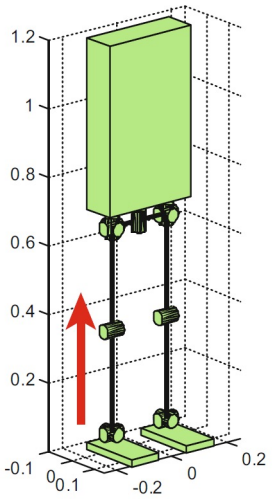
- Suppose we want to let the foot lift vertically with 0.1m/s



right hip pitch = -30°
knee pitch = 60°
ankle pitch = -30°

$$\dot{\boldsymbol{q}} = J^{-1}(\boldsymbol{q})\dot{\boldsymbol{x}}$$

Joint speed:
- hip pitch = -0.33 rad/s
- knee pitch = 0.67 rad/s
- ankle pitch = -0.33 rad/s



right hip pitch = 0°
knee pitch = 0°
ankle pitch = 0°

$$\dot{\boldsymbol{q}} = J^{-1}(\boldsymbol{q})\dot{\boldsymbol{x}}$$

Joint speed:
- hip pitch = NaN
- knee pitch = Nan
- ankle pitch = NaN

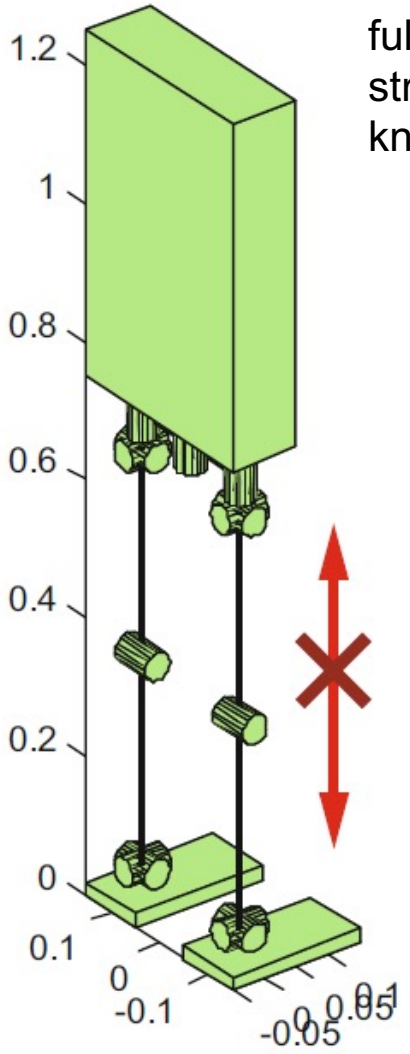
Matrix is singular to working precision

Singular pose:
- The robot can never move its foot vertically by applying any joint speeds at this configuration

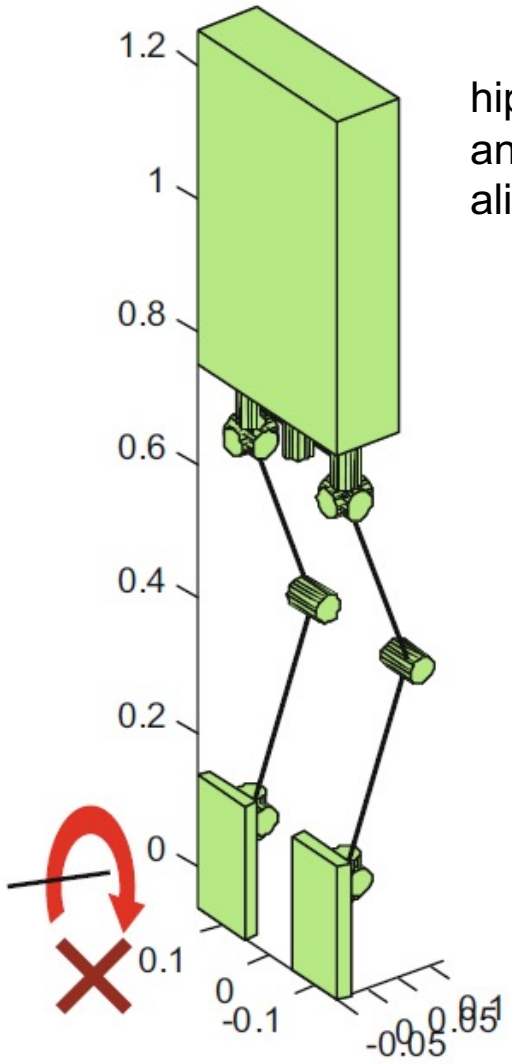
UNIVERSITY OF PLYMOUTH

Singular postures

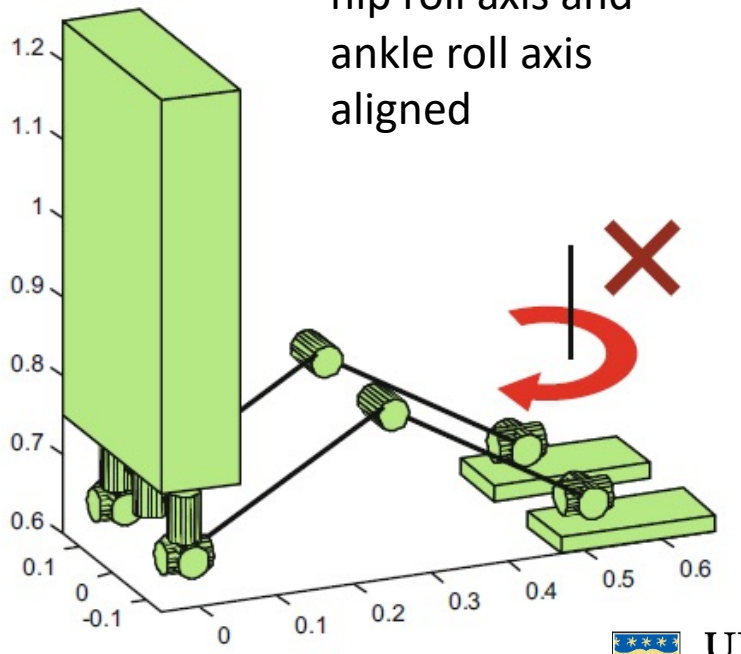
- In these postures there exist directions in which the end point cannot move (shown with arrows).
- In these cases the Jacobian inverse cannot be solved.



fully stretched knees



hip yaw axis and ankle roll axis aligned

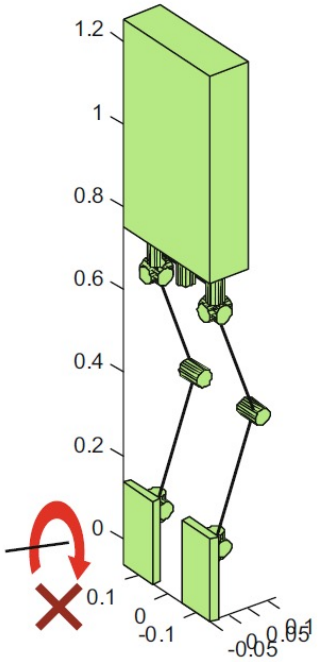


hip roll axis and ankle roll axis aligned

UNIVERSITY OF PLYMOUTH

Singular postures

- hip yaw axis and ankle roll axis aligned



- If you compute the inverse of the Jacobian in this configuration then you end up in numerical errors (numerical instability of the Newton's Method)
  - However if you put this configuration into the Jacobian and observe the resulting matrix, then with few tools from linear algebra you can have an estimate of the occurrence of this singularity
    - How close the configuration is to a singular posture

- From linear algebra
  - Determinant
    - Tend to zero as the posture of the robot is close to a singularity

$$|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} \square & \square & \square \\ \square & e & f \\ \square & h & i \end{vmatrix} - b \begin{vmatrix} \square & \square & \square \\ d & \square & f \\ g & \square & i \end{vmatrix} + c \begin{vmatrix} \square & \square & \square \\ d & e & \square \\ g & h & \square \end{vmatrix}$$

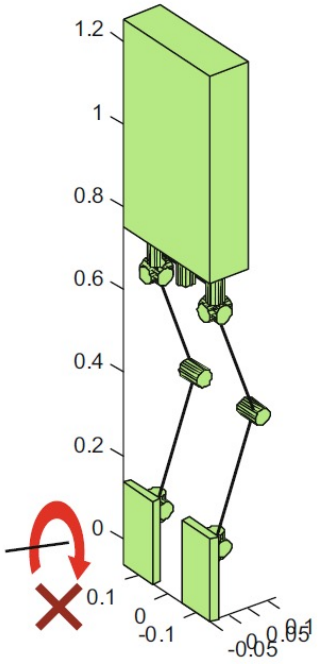
$$= a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

$$= aei + bfg + cdh - ceg - bdi - afh.$$

UNIVERSITY OF PLYMOUTH

Singular postures

- hip yaw axis and ankle roll axis aligned

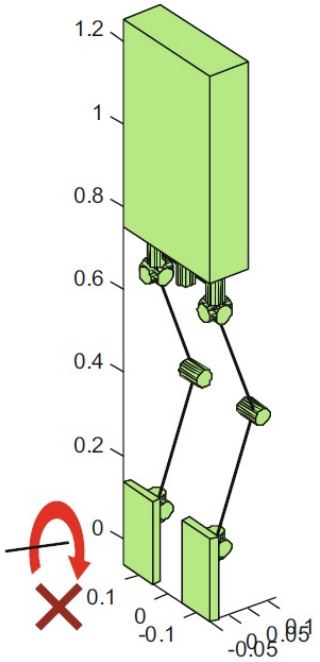


- If you compute the inverse of the Jacobian in this configuration then you end up in numerical errors (numerical instability of the Newton's Method)
  - However if you put this configuration into the Jacobian and observe the resulting matrix, then with few tools from linear algebra you can have an estimate of the occurrence of this singularity
    - How close the configuration is to a singular posture

- From linear algebra
  - Rank
    - Maximal number of linearly independent columns of A
    - Non singular postures
      - Identical to the dimension of the space spanned by the rows of the Jacobian

Singular postures

- hip yaw axis and ankle roll axis aligned



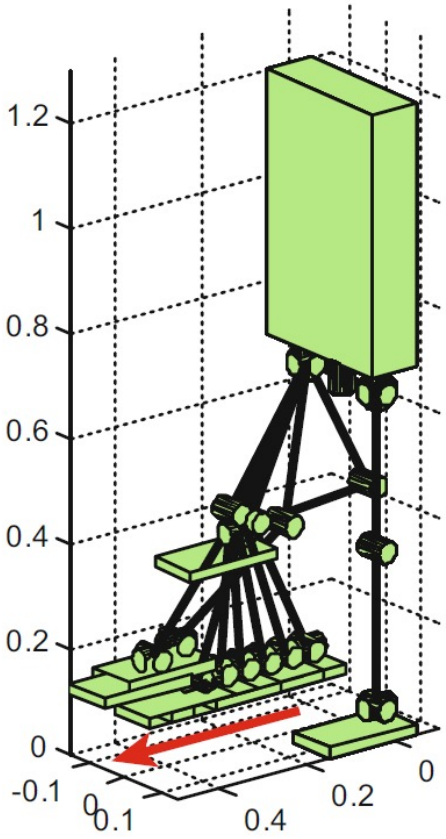
- If you compute the inverse of the Jacobian in this configuration then you end up in numerical errors (numerical instability of the Newton's Method)
  - However if you put this configuration into the Jacobian and observe the resulting matrix, then with few tools from linear algebra you can have an estimate of the occurrence of this singularity
    - How close the configuration is to a singular posture

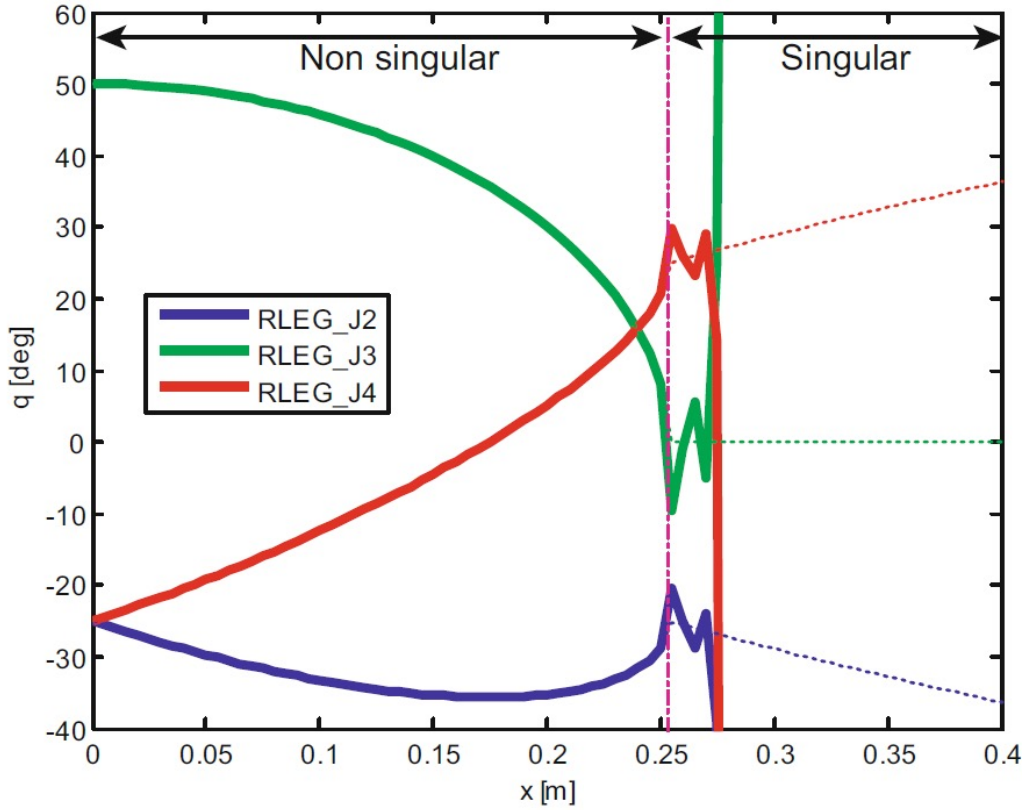
- Other methods (numerical)
  - LU Decomposition
  - Singular Value Decomposition (SVD) (computational demanding)
  - QR Decomposition (less expensive and more numerically robust)

UNIVERSITY OF PLYMOUTH

Singularity Robustness

- The Newton's does not work correctly around singular postures because of numerical instability





- joint angles calculated for letting the right foot move forward from a non-singular pose
- hip pitch, the knee pitch, and the ankle pitch angles at the given target foot position
- Analytical solutions are shown by dotted lines for comparison

- The foot position reached the singularity at the vertical chain line, then the numerical results started to vibrate and went off the chart (for example, the knee angle reaches over 8000 degrees).

UNIVERSITY OF PLYMOUTH

The Levenberg-Marquardt Method

Samuel R. Buss. *Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods*. IEEE Journal of Robotics and Automation. 2004 [Online]