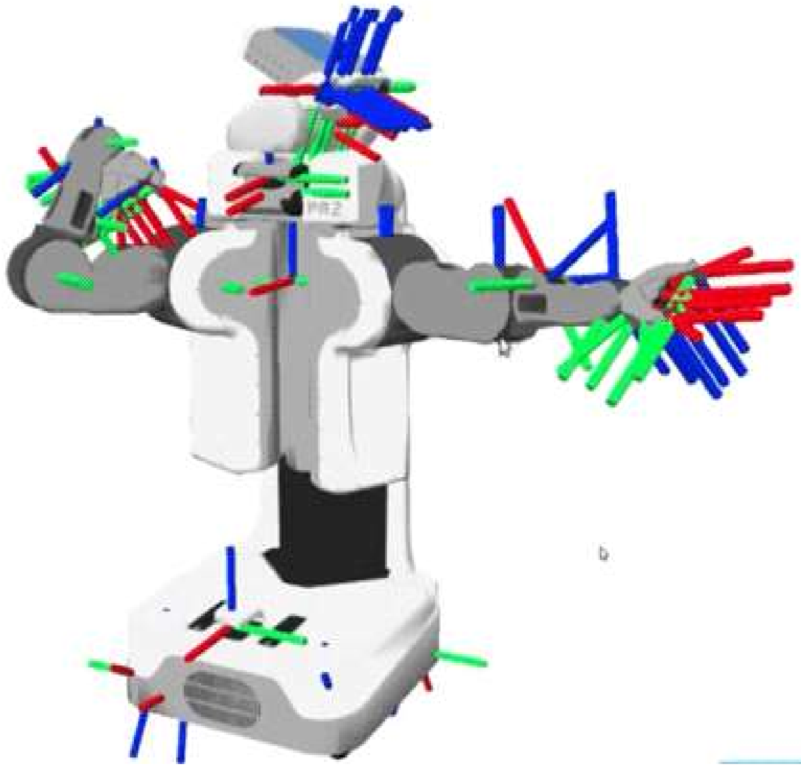
Module 8: ROS Advanced



TF

tf

In ROS the tf (transform) package is used to manage coordinate frame transformations in a robotic system. It provides a flexible and efficient way to represent and broadcast transformations between different coordinate frames. Transformations are essential in robotics to maintain spatial relationships between various components of a robot.

tf

Coordinate Frames:

A coordinate frame represents a specific spatial reference point in a robotic system. For example, a robot might have frames for its base, sensors, end effector, etc.

Transforms:

A transform is a mathematical representation of the translation and rotation between two coordinate frames. In ROS, these transforms are represented using the **tf::Transform** class.

Transform Broadcaster:

The **tf::TransformBroadcaster** class is used to publish transform information to the ROS system. It allows you to broadcast the transformation between different coordinate frames.

Transform Listener:

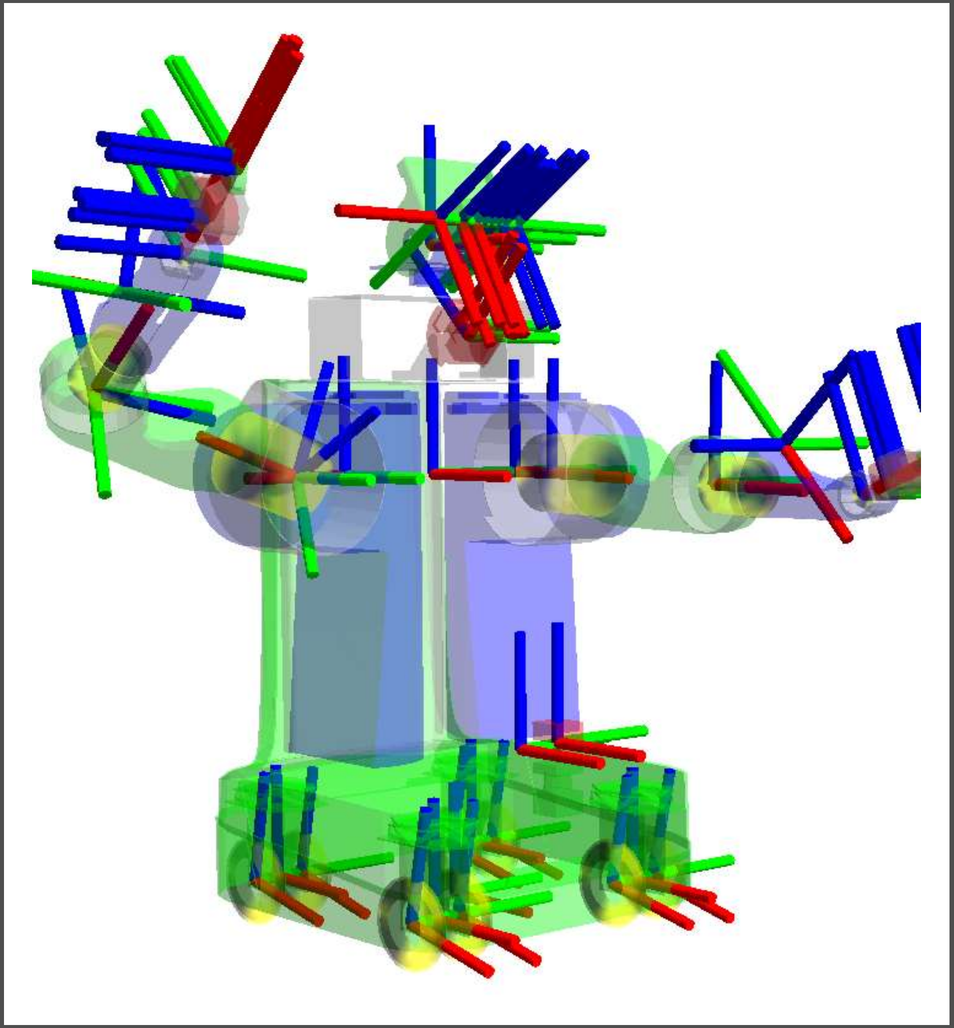
The **tf::TransformListener** class is used to listen for and query transform information. It allows you to retrieve the current transformation between two frames.

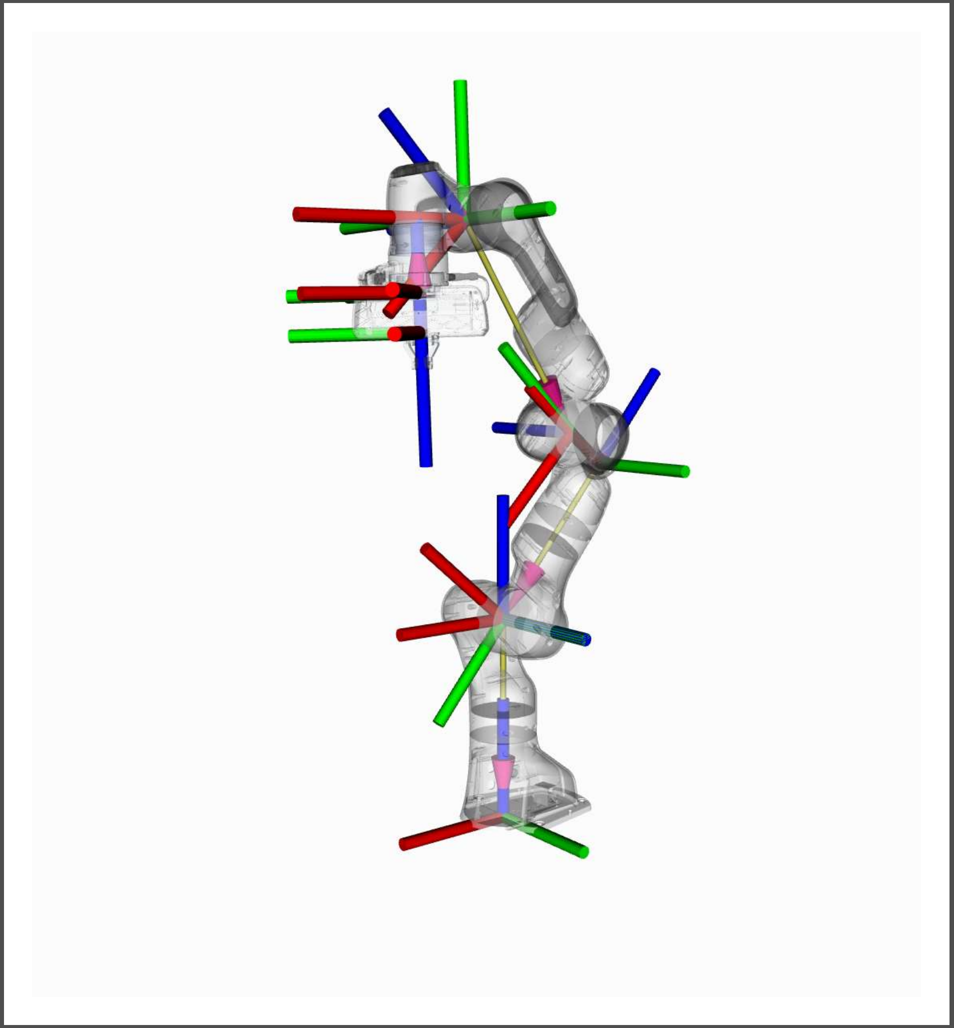
tf Tree:

The **tf** package organizes transforms into a tree structure, known as the tf tree. The tree represents the hierarchical relationship between coordinate frames in a robotic system.

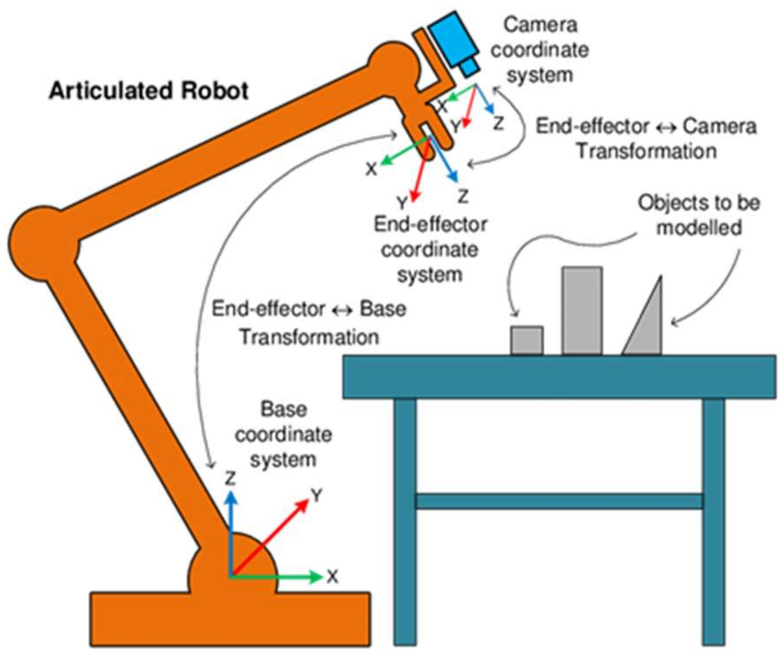
Static and Dynamic Transforms:

Static transforms are constant throughout the robot's operation, while dynamic transforms may change over time (e.g., due to robot motion).





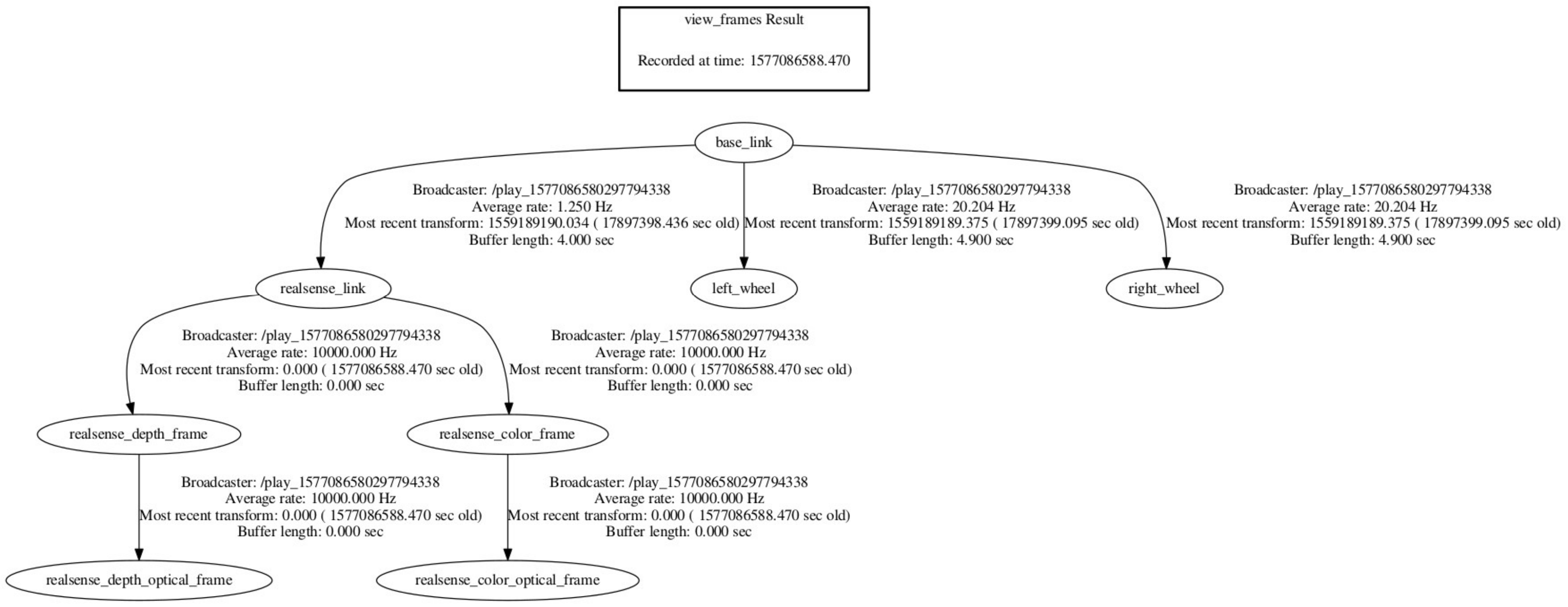
Coordinate Transformation


Articulated Robot
Camera coordinate system
End-effector ↔ Camera Transformation
End-effector coordinate system
Objects to be modelled
End-effector ↔ Base Transformation
Base coordinate system
X
Y
Z

Q: The 3D camera records the point cloud of the object, and the data is initially in the camera coordinate system. Our objective is to obtain the object's pose in the robot base coordinate system, allowing it to perform the pick-up action based on the received information.

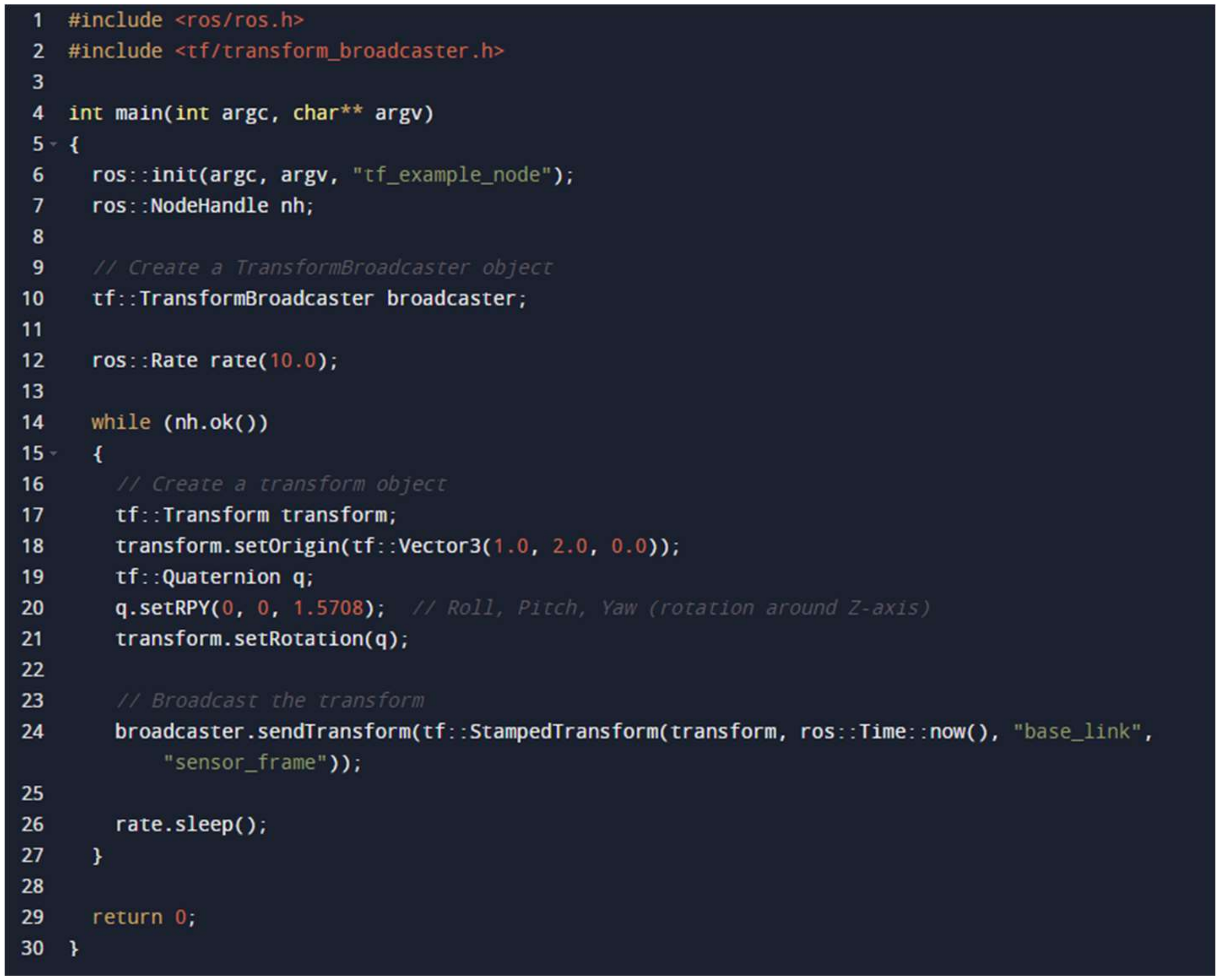
$Pose_{object\_to\_camera} * tf_{camera\_to\_\mathrm{base}}$ $Pose_{object\_to\_camera} * tf_{camera\_to\_end\_effector} * tf_{\mathrm{end}\_effector\_to\_base}$

tf tree



view_frames Result
Recorded at time: 1577086588.470
base_link
Broadcaster: /play_1577086580297794338
Average rate: 1.250 Hz
Most recent transform: 1559189190.034 (17897398.436 sec old)
Buffer length: 4.000 sec
Broadcaster: /play_1577086580297794338
Average rate: 20.204 Hz
Most recent transform: 1559189189.375 (17897399.095 sec old)
Buffer length: 4.900 sec
Broadcaster: /play_1577086580297794338
Average rate: 20.204 Hz
Most recent transform: 1559189189.375 (17897399.095 sec old)
Buffer length: 4.900 sec
realsense_link
left_wheel
right_wheel
Broadcaster: /play_1577086580297794338
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1577086588.470 sec old)
Buffer length: 0.000 sec
Broadcaster: /play_1577086580297794338
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1577086588.470 sec old)
Buffer length: 0.000 sec
realsense_depth_frame
realsense_color_frame
Broadcaster: /play_1577086580297794338
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1577086588.470 sec old)
Buffer length: 0.000 sec
Broadcaster: /play_1577086580297794338
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1577086588.470 sec old)
Buffer length: 0.000 sec
realsense_depth_optical_frame
realsense_color_optical_frame

Example



```cpp
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>

int main(int argc, char** argv)
{
  ros::init(argc, argv, "tf_example_node");
  ros::NodeHandle nh;

  // Create a TransformBroadcaster object
  tf::TransformBroadcaster broadcaster;

  ros::Rate rate(10.0);

  while (nh.ok())
  {
    // Create a transform object
    tf::Transform transform;
    transform.setOrigin(tf::Vector3(1.0, 2.0, 0.0));
    tf::Quaternion q;
    q.setRPY(0, 0, 1.5708);  // Roll, Pitch, Yaw (rotation around Z-axis)
    transform.setRotation(q);

    // Broadcast the transform
    broadcaster.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "base_link",
        "sensor_frame"));

    rate.sleep();
  }

  return 0;
}
```

tf transformation

Various representations of transformation

- Euler Angles
- Quaternion
- Transformation matrix
- Axis Angle

Euler Angles

1. **Representation:**
   - Euler angles represent rotations using three angles: roll (ϕ), pitch (ϑ), and yaw (ψ).
   - Roll is rotation around the x-axis, pitch around the y-axis, and yaw around the z-axis.
   - The sequence of rotations (XYZ, ZYX, etc.) affects the final orientation.
2. **Mathematical Form:**
   - A rotation matrix based on Euler angles (XYZ sequence):

$$R_{XYZ} = R_x(\emptyset).\, R_y(\theta).\, R_z(\psi)$$

   - where R_x , R_y , R_z are rotation matrices for the respective axes.

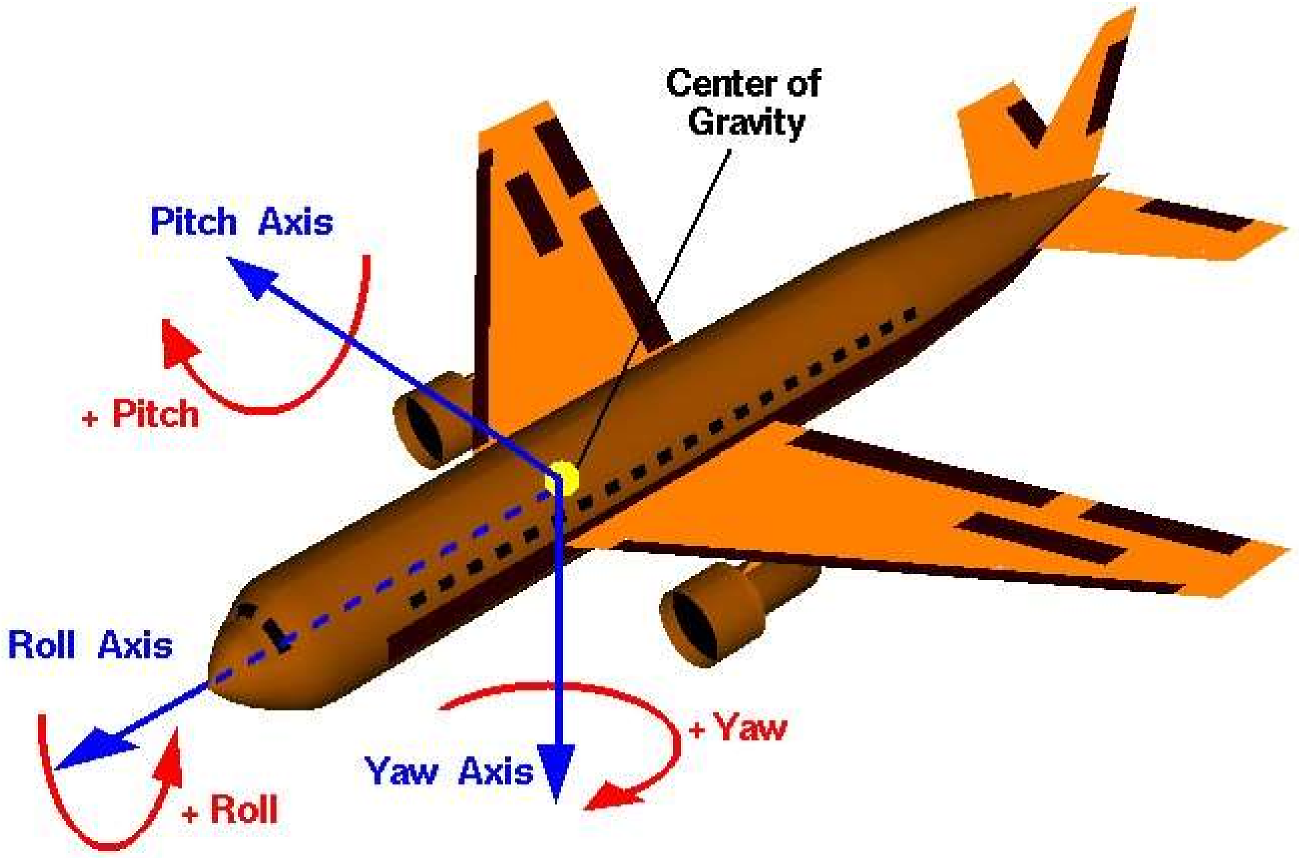
Euler Angles

Advantages:

- Intuitive interpretation of rotations in terms of familiar pitch, yaw, and roll movements.
- Simple to understand and visualize.

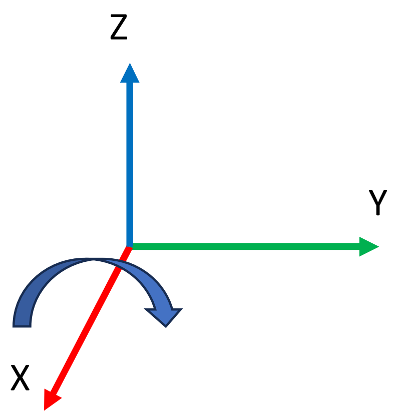
Limitations:

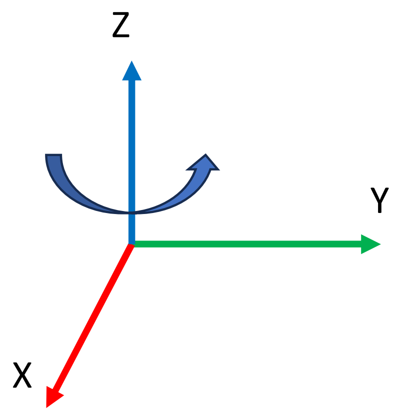
- Gimbal lock: Euler angles can experience gimbal lock, where the rotational freedom is lost, leading to difficulties in representing certain orientations.


Center of Gravity
Pitch Axis
+ Pitch
Roll Axis
+ Roll
Yaw Axis
+ Yaw

Positive/Negative Sign

- Euler angle is positive if the rotation is anti-clockwise looking from the direction of the axis
- Eule angle is negative if the rotation is clockwise looking from the direction of the axis

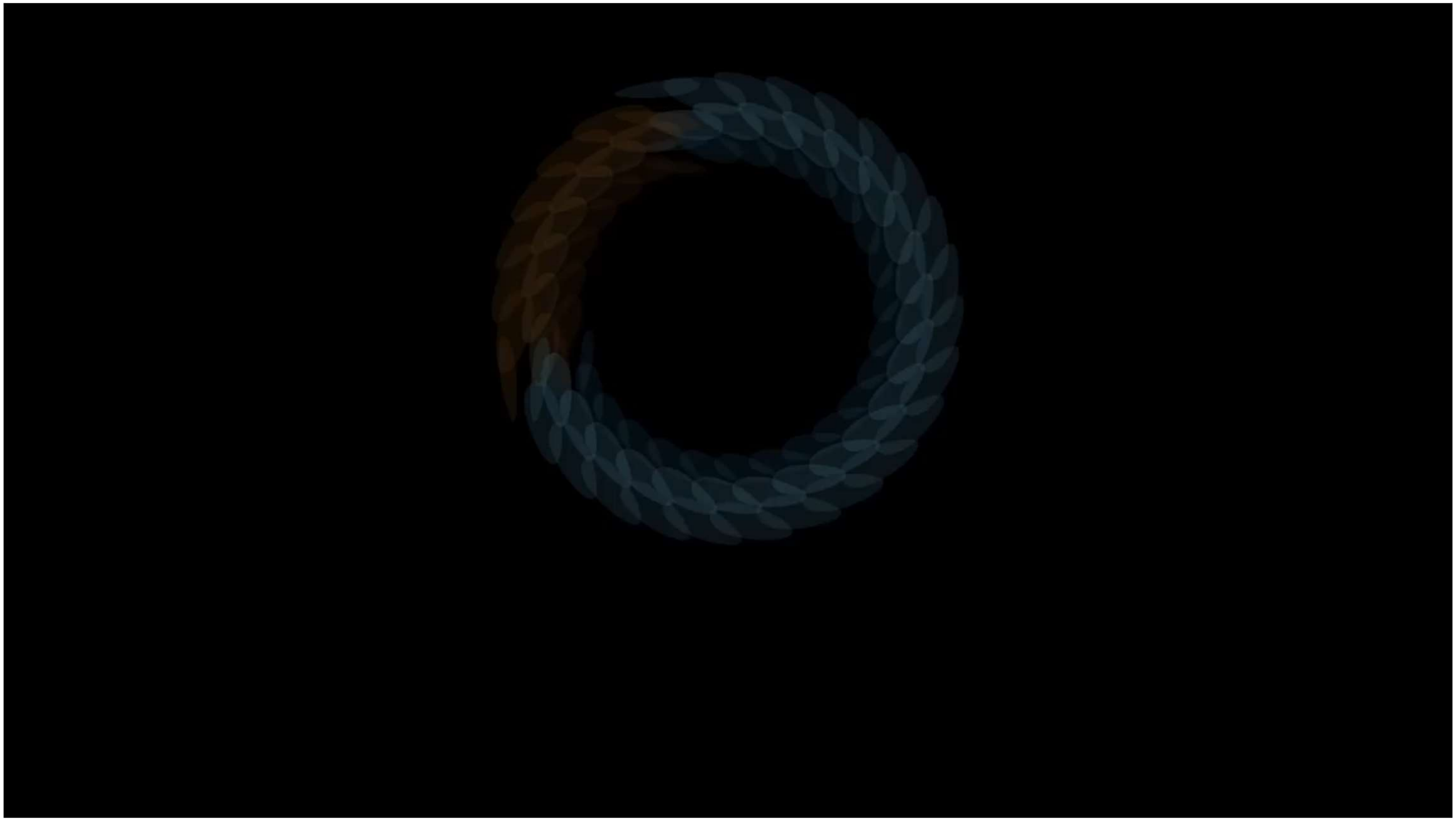




Gimbal Lock

Gimbal lock is a phenomenon that occurs in systems that use Euler angles for representing orientation, such as in 3D graphics, robotics, and aerospace applications. It happens when the rotation axes of a gimbal system become aligned, causing a loss of one degree of freedom and leading to unexpected behavior in orientation representation.

A gimbal is a set of rotating rings or pivots that allow an object to rotate in multiple axes independently. In a three-gimbal system (such as pitch, yaw, and roll), gimbal lock occurs when two of the three gimbals align, effectively reducing the system's degrees of freedom from three to two. This alignment removes one axis of rotation, causing the system to lose its ability to represent certain orientations.

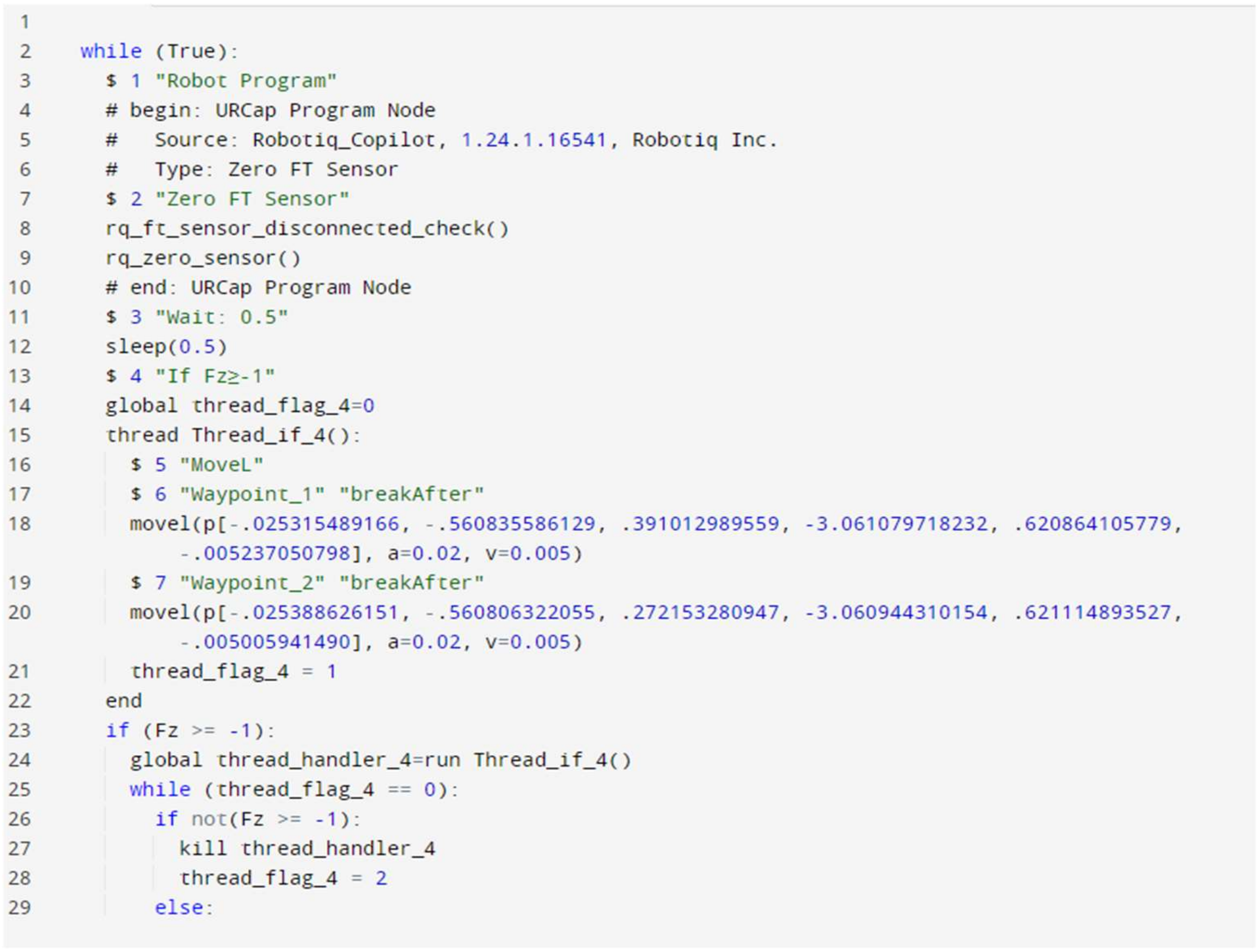


Gimbal Lock

Gimbal lock can lead to problems in systems that rely on accurate orientation representation, such as aircraft navigation systems, robotic arms, or 3D graphics applications. It can cause inaccuracies in calculations, unexpected behavior in control systems, or visual artifacts in rendered images.

To avoid gimbal lock, alternative methods for representing orientation, such as quaternions, are often used. Quaternions offer advantages over Euler angles because they do not suffer from gimbal lock and provide a more robust representation of orientation, especially in systems with continuous rotation or complex motion.

UR Robot Script



```

while (True):
  $ 1 "Robot Program"
  # begin: URCap Program Node
  #   Source: Robotiq_Copilot, 1.24.1.16541, Robotiq Inc.
  #   Type: Zero FT Sensor
  $ 2 "Zero FT Sensor"
  rq_ft_sensor_disconnected_check()
  rq_zero_sensor()
  # end: URCap Program Node
  $ 3 "Wait: 0.5"
  sleep(0.5)
  $ 4 "If Fz≥-1"
  global thread_flag_4=0
  thread Thread_if_4():
    $ 5 "MoveL"
    $ 6 "Waypoint_1" "breakAfter"
    movel(p[-.025315489166, -.560835586129, .391012989559, -3.061079718232, .620864105779,
        -.005237050798], a=0.02, v=0.005)
    $ 7 "Waypoint_2" "breakAfter"
    movel(p[-.025388626151, -.560806322055, .272153280947, -3.060944310154, .621114893527,
        -.005005941490], a=0.02, v=0.005)
    thread_flag_4 = 1
  end
  if (Fz >= -1):
    global thread_handler_4=run Thread_if_4()
    while (thread_flag_4 == 0):
      if not(Fz >= -1):
        kill thread_handler_4
        thread_flag_4 = 2
      else:
```

Quaternion

1. Representation:

- Quaternions are four-dimensional mathematical entities represented as
$$q = w + xi + yj + zk$$
- w is the scalar part, and $xi + yj + zk$ is the vector part.

2. Mathematical Form:

- A quaternion representing a rotation:
$$q = \cos(\frac{\theta}{2}) + \sin\left(\frac{\theta}{2}\right).(ai + bj + ck)$$

where *a*, *b*, and *c* are the components of the rotation axis vector.

Quaternion

Advantages:

- No gimbal lock: Quaternions avoid gimbal lock issues associated with Euler angles.
- Efficient for interpolating and combining multiple rotations.

Limitations:

- Less intuitive for understanding rotations compared to Euler angles.
- Requires conversion to other representations for human interpretation.

Selection rules

- **Euler Angles:**
  - Suitable for simple applications and human communication of orientations.
  - Prone to gimbal lock in certain situations.
- **Quaternions:**
  - Efficient for numerical calculations, interpolations, and combining rotations.
  - Preferred in robotics and computer graphics for stability and performance.

Euler Angles and Quaternion Conversion

Euler angles to Quaternion

3-2-1 sequence (yaw (Body-Z) (ψ), + pitches (Body-Y) θ + rolls (Body-X) φ) to quaternion

$$\begin{aligned}\mathbf{q}_{\mathrm{IB}} &= \begin{bmatrix}\cos(\psi/2)\\0\\0\\\sin(\psi/2)\end{bmatrix}\begin{bmatrix}\cos(\theta/2)\\0\\\sin(\theta/2)\\0\end{bmatrix}\begin{bmatrix}\cos(\phi/2)\\\sin(\phi/2)\\0\\0\end{bmatrix}\\ &= \begin{bmatrix}\cos(\phi/2)\cos(\theta/2)\cos(\psi/2)+\sin(\phi/2)\sin(\theta/2)\sin(\psi/2)\\\sin(\phi/2)\cos(\theta/2)\cos(\psi/2)-\cos(\phi/2)\sin(\theta/2)\sin(\psi/2)\\\cos(\phi/2)\sin(\theta/2)\cos(\psi/2)+\sin(\phi/2)\cos(\theta/2)\sin(\psi/2)\\\cos(\phi/2)\cos(\theta/2)\sin(\psi/2)-\sin(\phi/2)\sin(\theta/2)\cos(\psi/2)\end{bmatrix}\end{aligned}$$

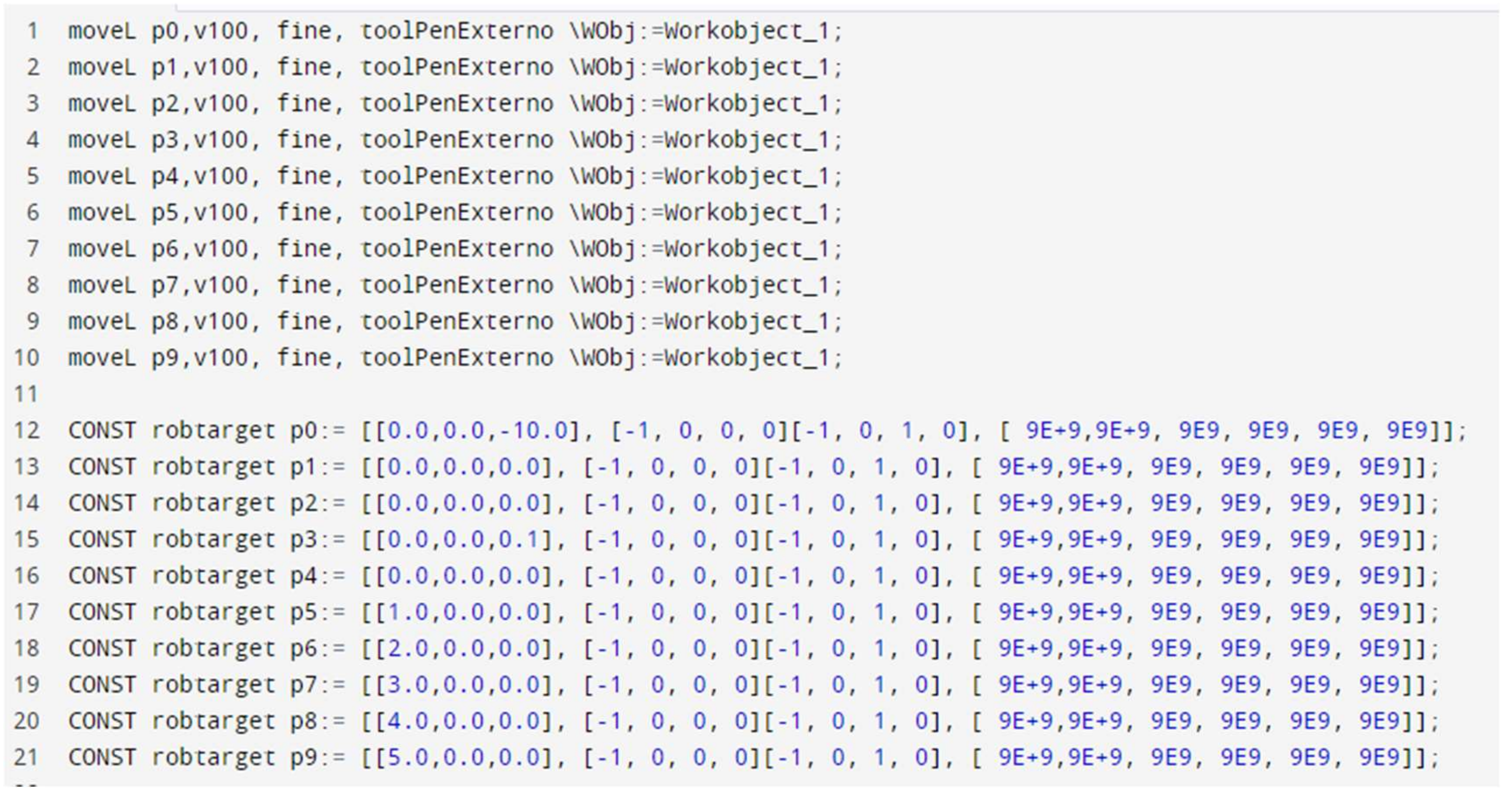
Euler Angles and Quaternion Conversion

- **Quaternion to Euler angles**

Quaternion to 3-2-1 sequence (yaw (Body-Z) (ψ), + pitches (Body-Y) θ + rolls (Body-X) φ)

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \operatorname{atan2}\left(2(q_w q_x + q_y q_z), 1 - 2(q_x^2 + q_y^2)\right) \\ -\pi/2 + 2\operatorname{atan2}\left(\sqrt{1 + 2(q_w q_y - q_x q_z)}, \sqrt{1 - 2(q_w q_y - q_x q_z)}\right) \\ \operatorname{atan2}\left(2(q_w q_z + q_x q_y), 1 - 2(q_y^2 + q_z^2)\right) \end{bmatrix}$$

ABB Robot Script



```
moveL p0,v100, fine, toolPenExterno \WObj:=Workobject_1;
moveL p1,v100, fine, toolPenExterno \WObj:=Workobject_1;
moveL p2,v100, fine, toolPenExterno \WObj:=Workobject_1;
moveL p3,v100, fine, toolPenExterno \WObj:=Workobject_1;
moveL p4,v100, fine, toolPenExterno \WObj:=Workobject_1;
moveL p5,v100, fine, toolPenExterno \WObj:=Workobject_1;
moveL p6,v100, fine, toolPenExterno \WObj:=Workobject_1;
moveL p7,v100, fine, toolPenExterno \WObj:=Workobject_1;
moveL p8,v100, fine, toolPenExterno \WObj:=Workobject_1;
moveL p9,v100, fine, toolPenExterno \WObj:=Workobject_1;

CONST robtarget p0:= [[0.0,0.0,-10.0], [-1, 0, 0, 0][-1, 0, 1, 0], [ 9E+9,9E+9, 9E9, 9E9, 9E9, 9E9]];
CONST robtarget p1:= [[0.0,0.0,0.0], [-1, 0, 0, 0][-1, 0, 1, 0], [ 9E+9,9E+9, 9E9, 9E9, 9E9, 9E9]];
CONST robtarget p2:= [[0.0,0.0,0.0], [-1, 0, 0, 0][-1, 0, 1, 0], [ 9E+9,9E+9, 9E9, 9E9, 9E9, 9E9]];
CONST robtarget p3:= [[0.0,0.0,0.1], [-1, 0, 0, 0][-1, 0, 1, 0], [ 9E+9,9E+9, 9E9, 9E9, 9E9, 9E9]];
CONST robtarget p4:= [[0.0,0.0,0.0], [-1, 0, 0, 0][-1, 0, 1, 0], [ 9E+9,9E+9, 9E9, 9E9, 9E9, 9E9]];
CONST robtarget p5:= [[1.0,0.0,0.0], [-1, 0, 0, 0][-1, 0, 1, 0], [ 9E+9,9E+9, 9E9, 9E9, 9E9, 9E9]];
CONST robtarget p6:= [[2.0,0.0,0.0], [-1, 0, 0, 0][-1, 0, 1, 0], [ 9E+9,9E+9, 9E9, 9E9, 9E9, 9E9]];
CONST robtarget p7:= [[3.0,0.0,0.0], [-1, 0, 0, 0][-1, 0, 1, 0], [ 9E+9,9E+9, 9E9, 9E9, 9E9, 9E9]];
CONST robtarget p8:= [[4.0,0.0,0.0], [-1, 0, 0, 0][-1, 0, 1, 0], [ 9E+9,9E+9, 9E9, 9E9, 9E9, 9E9]];
CONST robtarget p9:= [[5.0,0.0,0.0], [-1, 0, 0, 0][-1, 0, 1, 0], [ 9E+9,9E+9, 9E9, 9E9, 9E9, 9E9]];
```

Transformation matrix

- A transformation matrix is a mathematical representation used to perform various geometric transformations in 3D space, such as translation, rotation, scaling, and shearing. These matrices are crucial in computer graphics, computer vision, robotics, and other fields dealing with spatial transformations.
- The most common type of transformation matrix is the homogeneous transformation matrix, which can represent both rotations and translations. In a 4x4 homogeneous transformation matrix, the top-left 3x3 matrix typically represents the rotation and scaling components, and the rightmost column represents the translation.

Transformation matrix

- General Structure of a 4x4 Homogeneous Transformation Matrix

$$T = \begin{matrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{matrix}$$

Where

R_{ij} are the elements of the 3x3 rotation matrix.

T_x , T_y , T_z are the translation components.

The last row [0,0,0,1] is a homogenizing row to make the matrix compatible with homogeneous coordinates.

Translation

- Translation matrix

$$T = \begin{matrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{matrix}$$

Rotation R_X

- about X-axis

$$R_X(\emptyset) = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\emptyset) & -\sin(\emptyset) & 0 \\ 0 & \sin(\emptyset) & \cos(\emptyset) & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Rotation R_Y

- about Y-axis

$$R_X(\theta) = \begin{matrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Rotation R_z

- about Z-axis

$$R_X(\psi) = \begin{matrix} \cos(\psi) & -\sin(\psi) & 0 & 0 \\ \sin(\psi) & \cos(\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Combine multiple transformations

- To combine multiple transformations, you can multiply the corresponding transformation matrices in the desired order. For example, to apply a rotation followed by a translation:

$$T_{combined} = T_{Translation} . T_{Rotation}$$

Euler Angles to Transformation Matrix

$$\mathrm{R} = R_X . R_Y . R_Z$$

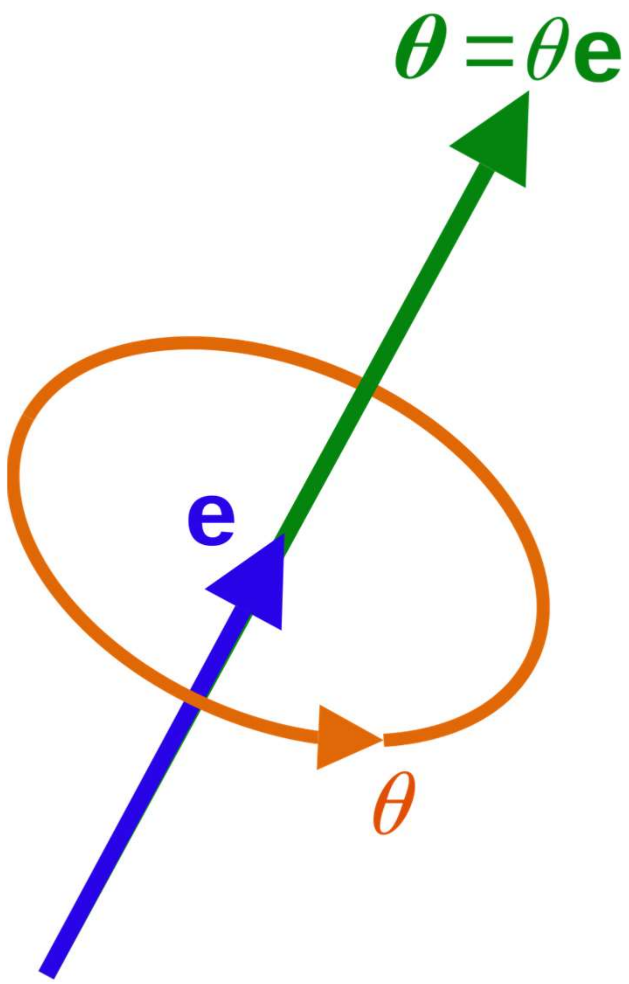
$$R_X(\emptyset) = \begin{matrix} 1 & 0 & 0 \\ 0 & \cos(\emptyset) & -\sin(\emptyset) \\ 0 & \sin(\emptyset) & \cos(\emptyset) \end{matrix}$$

$$R_X(\theta) = \begin{matrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{matrix}$$

$$R_X(\psi) = \begin{matrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{matrix}$$

Axis Angle

- The axis-angle representation is another way to describe 3D rotations. In this representation, a rotation is specified by an axis of rotation and an angle by which the rotation occurs around that axis. This representation is often used in computer graphics, robotics, and other fields to represent orientation.


$\boldsymbol{\theta} = \theta\mathbf{e}$
$\mathbf{e}$
θ

Axis Angle

- The axis-angle representation is typically denoted as (**a**,ϑ), where:
- **a** is a unit vector representing the axis of rotation.
- ϑ is the angle of rotation in radians.
- The rotation matrix *R* representing the rotation can be obtained using Rodrigues' rotation formula:

$$R = I + \sin(\theta)K + (1 - \cos\theta)K^2$$

where:

I is the identity matrix.

K is the skew-symmetric matrix associated with the unit vector **a**:

$$K = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$$

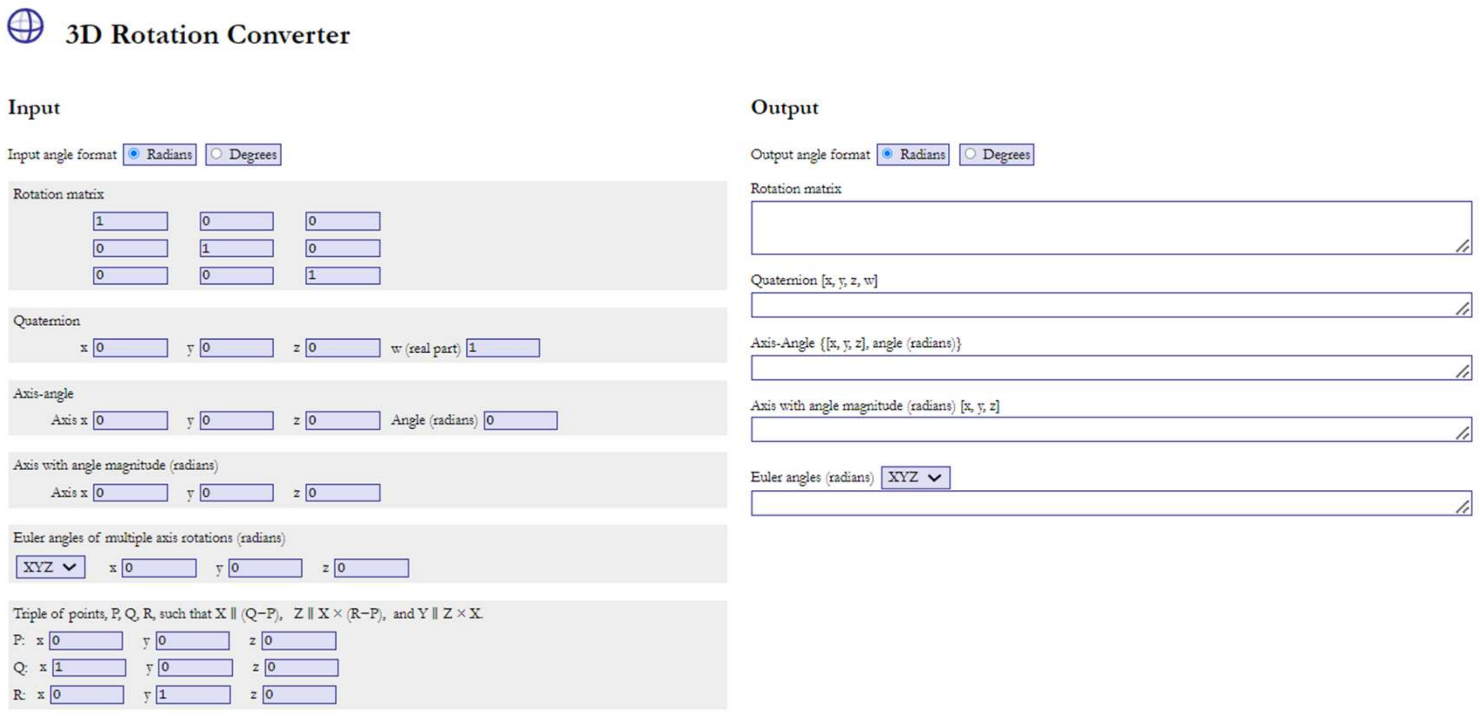
Conversion from Rotation Matrix to Axis-Angle

- To convert a rotation matrix R to axis-angle representation, you can use the following steps:

$$\theta = \cos^{-1}\left(\frac{R_{11} + R_{22} + R_{33} - 1}{2}\right)$$

$$\mathrm{a} = \frac{1}{2\sin(\theta)}\begin{bmatrix} R_{32} & R_{23} \\ R_{13} & R_{31} \\ R_{21} & R_{12} \end{bmatrix}$$

Online converter



3D Rotation Converter

Input

Input angle format [✓ Radians] Degrees

Rotation matrix

1	0	0
0	1	0
0	0	1

Quaternion

x 0 y 0 z 0 w (real part) 1

Axis-angle

Axis x 0 y 0 z 0 Angle (radians) 0

Axis with angle magnitude (radians)

Axis x 0 y 0 z 0

Euler angles of multiple axis rotations (radians)

XYZ x 0 y 0 z 0

Triple of points, P, Q, R, such that X ∥ (Q−P), Z ∥ X × (R−P), and Y ∥ Z × X.

P:	x 0	y 0	z 0
Q:	x 1	y 0	z 0
R:	x 0	y 1	z 0

Output

Output angle format [✓ Radians] Degrees

Rotation matrix

Quaternion [x, y, z, w]

Axis-Angle {[x, y, z], angle (radians)}

Axis with angle magnitude (radians) [x, y, z]

Euler angles (radians) XYZ

https://www.andre-gaschler.com/rotationconverter/

URDF

URDF

URDF stands for Unified Robot Description Format. It is an XML file format used in the field of robotics to describe the physical and kinematic properties of a robot. URDF files are commonly used in the context of ROS, a framework for building robot software.

The URDF file contains information about a robot's structure, including links, joints, sensors, visual and collision properties, and more. This file is essential for simulation, visualization, and control of robotic systems.

Similar to DH-parameters

URDF key element

Robot Description: The **<robot>** tag encloses the entire description of the robot.

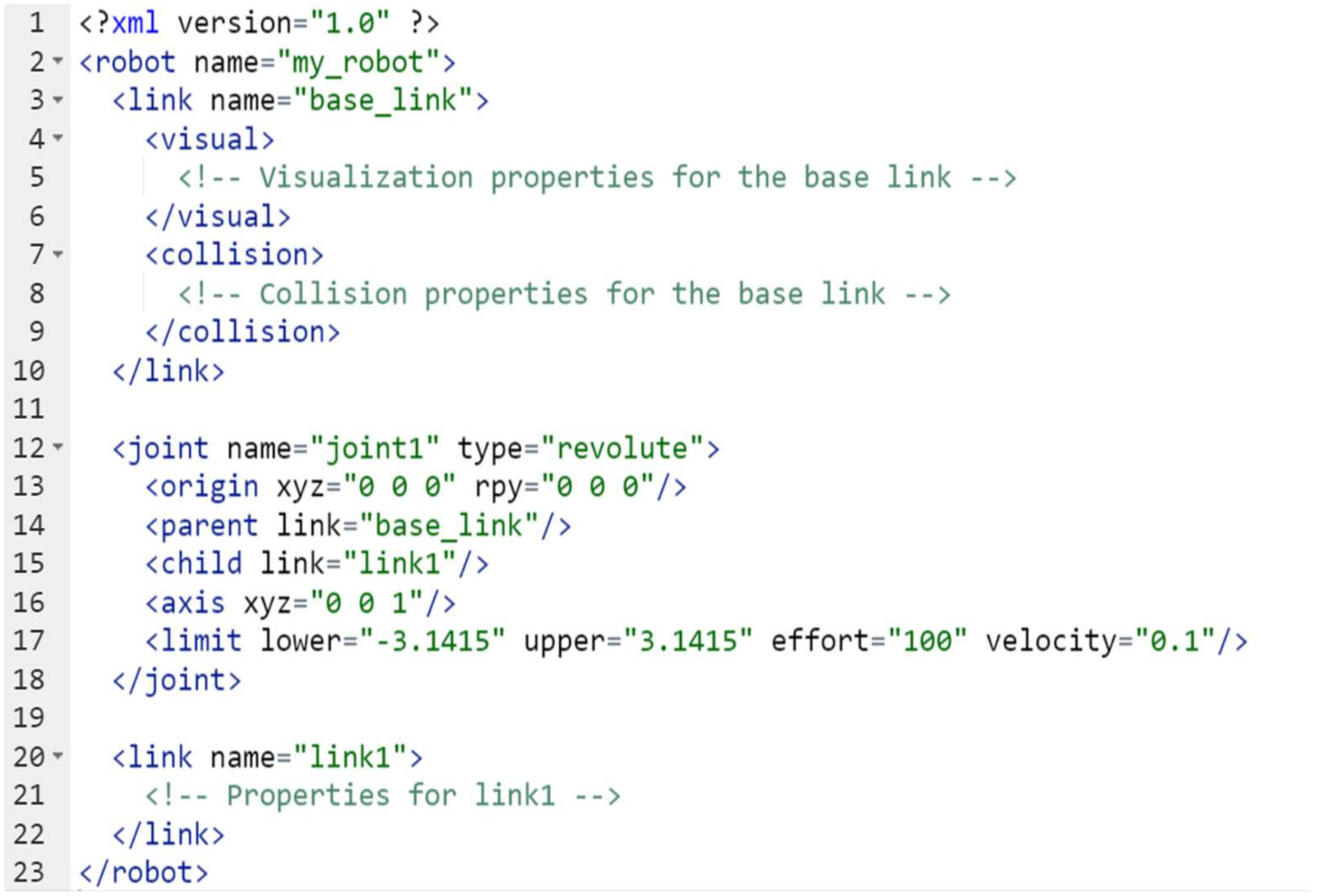
Links: The **<link>** tag defines the physical properties of a rigid body or a link in the robot. This includes visual and collision properties.

Joints: The **<joint>** tag describes the kinematic properties of joints connecting two links. It includes information about the joint type (revolute, prismatic, etc.), axis, limits, and origin.

Visual and Collision Properties: Inside the **<link>** tag, there are **<visual>** and **<collision>** tags that describe how the link looks visually and how it behaves in collision checks, respectively.

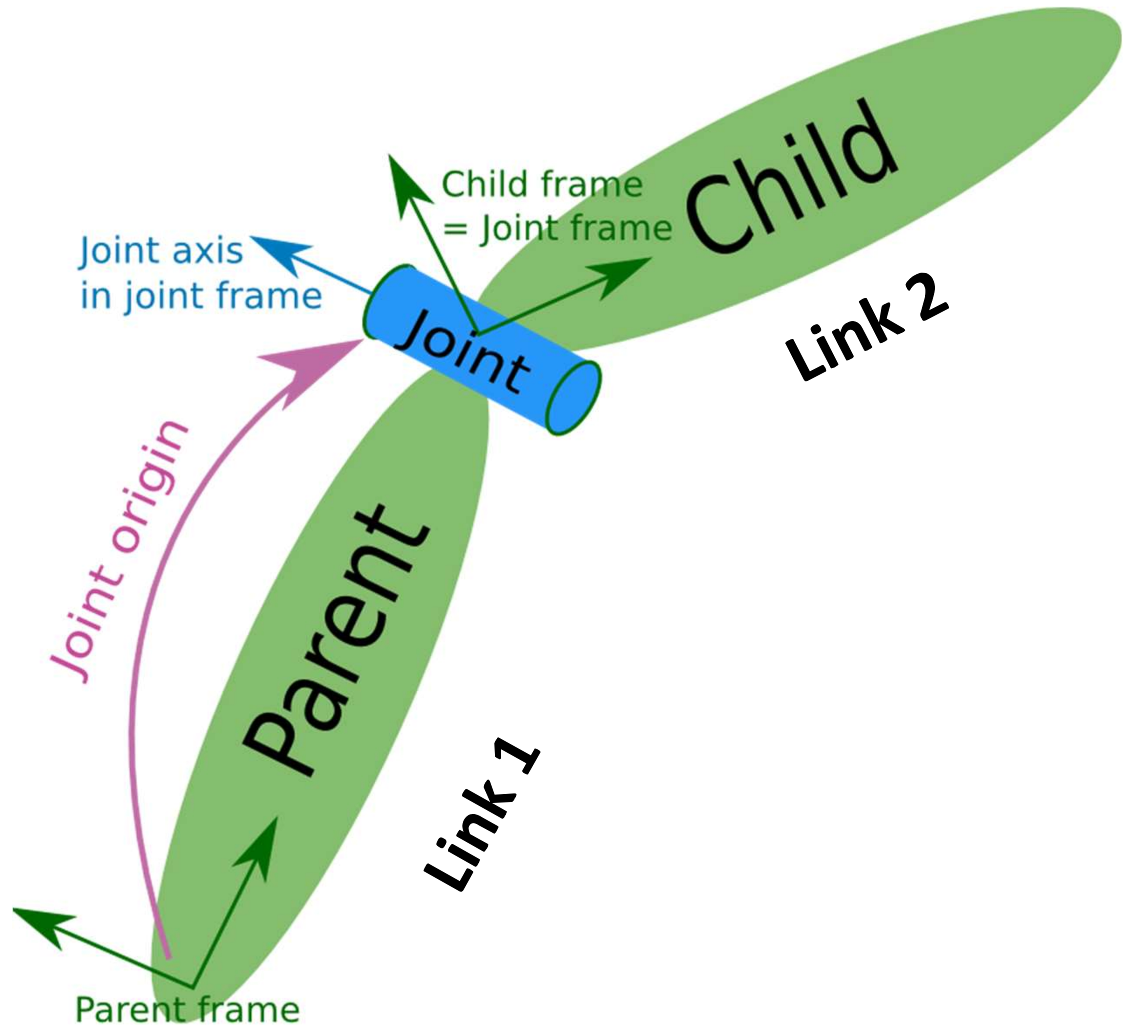
Transmission: The **<transmission>** tag specifies how joint motion is transmitted to the robot. This is important for control and simulation.

URDF Example


```xml
<?xml version="1.0" ?>
<robot name="my_robot">
  <link name="base_link">
    <visual>
      <!-- Visualization properties for the base link -->
    </visual>
    <collision>
      <!-- Collision properties for the base link -->
    </collision>
  </link>

  <joint name="joint1" type="revolute">
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <parent link="base_link"/>
    <child link="link1"/>
    <axis xyz="0 0 1"/>
    <limit lower="-3.1415" upper="3.1415" effort="100" velocity="0.1"/>
  </joint>

  <link name="link1">
    <!-- Properties for link1 -->
  </link>
</robot>
```


Child frame
= Joint frame
Child
Joint axis
in joint frame
Joint
Link 2
Joint origin
Parent
Link 1
Parent frame

Links

The <link> element in a URDF file describes a rigid body or a link in a robot. It provides information about the physical properties of the link, such as its visual representation, collision properties, inertia, and more. Here's a breakdown of the key components within the <link> element:

Link Name (name attribute)

- The name attribute uniquely identifies the link within the robot description.

```
<link name="base_link">
  <!-- Link properties go here -->
</link>
```

Visual Properties (<visual>)

- The <visual> tag defines the visual representation of the link for visualization purposes. It includes geometry, material properties, and an origin.

```
<visual>
  <geometry>
    <!-- Define the visual geometry, e.g., box, cylinder, mesh -->
  </geometry>
  <material>
    <!-- Define visual material properties like color or texture -->
  </material>
  <origin xyz="0 0 0" rpy="0 0 0"/>
</visual>
```

Collision Properties (<collision>)

- The <collision> tag describes the collision properties of the link, which are used in physics simulations to approximate how the robot interacts with its environment.

```
<collision>
  <geometry>
    <!-- Define the collision geometry, usually simpler than visual geometry -->
  </geometry>
  <!-- Collision-specific properties can be specified here -->
</collision>
```

Visual and Collision Geometry

- Inside the <geometry> tags of <visual> and <collision>, you can define the geometric shape of the link. This can be a box, cylinder, sphere, or even a mesh.

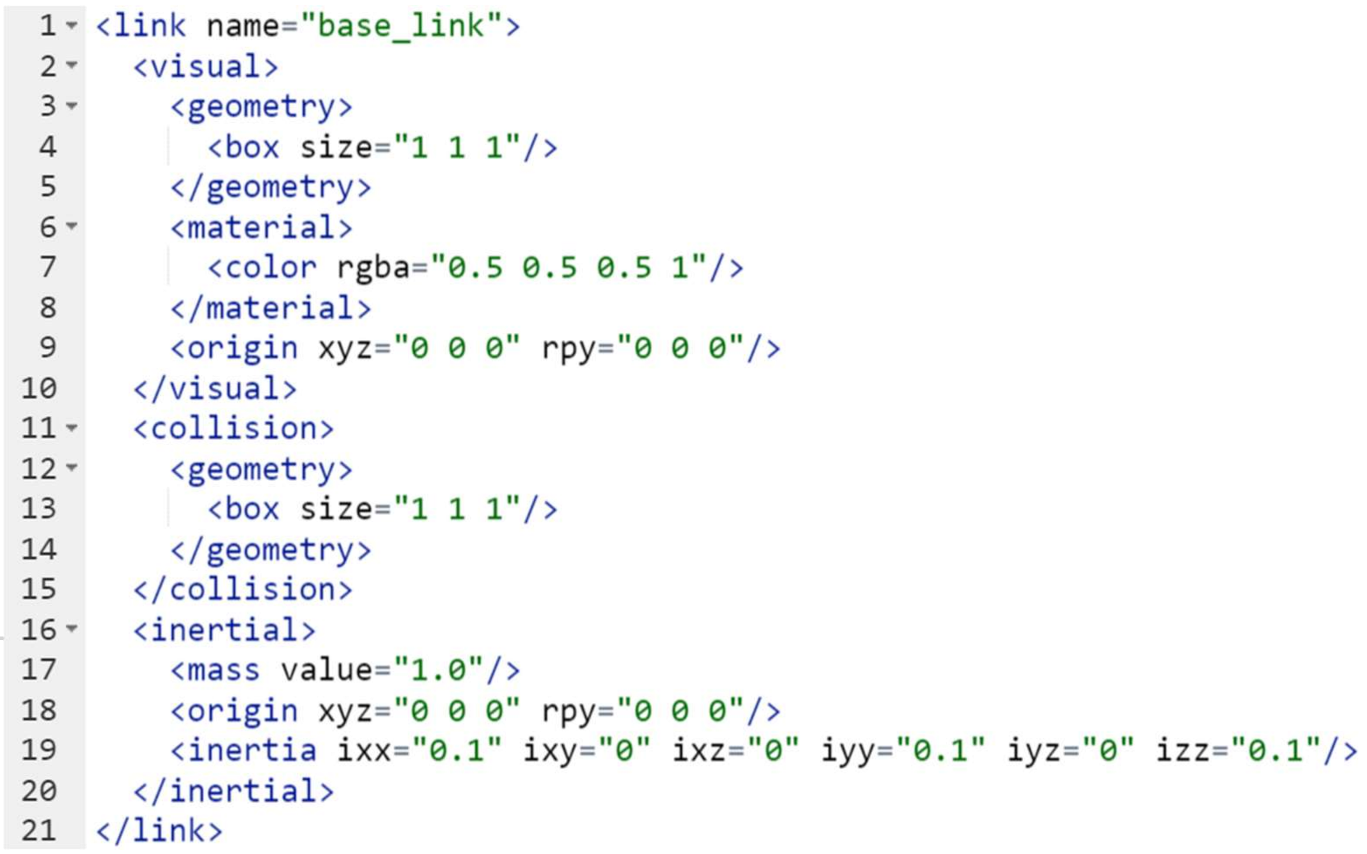
```
<geometry>
  <box size="1 1 1"/>
</geometry>
```

Inertia (<inertial>)

- The <inertial> tag provides information about the link's inertia, which is essential for dynamics calculations.
- It includes the mass of the link and the rotational and translational inertia matrices.

```
<inertial>
  <mass value="1.0"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <inertia ixx="0.1" ixy="0" ixz="0" iyy="0.1"
iyz="0" izz="0.1"/>
</inertial>
```

Links example



```xml
<link name="base_link">
  <visual>
    <geometry>
      <box size="1 1 1"/>
    </geometry>
    <material>
      <color rgba="0.5 0.5 0.5 1"/>
    </material>
    <origin xyz="0 0 0" rpy="0 0 0"/>
  </visual>
  <collision>
    <geometry>
      <box size="1 1 1"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="1.0"/>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <inertia ixx="0.1" ixy="0" ixz="0" iyy="0.1" iyz="0" izz="0.1"/>
  </inertial>
</link>
```

Joints (<joint>)

- The <joint> element in a URDF (Unified Robot Description Format) file describes the kinematic properties of a joint connecting two links in a robot. Joints define the relationship between links and allow for movement. Here's an overview of the key components within the <joint> element:

Joint Name (name attribute)

- The name attribute uniquely identifies the joint within the robot description.

```
<joint name="joint1" type="revolute">
  <!-- Joint properties go here -->
</joint>
```

Joint Type (type attribute)

The type attribute specifies the type of joint. Common types include:

- **revolute**: Rotational joint (e.g., a hinge joint), limited range specified by the upper and lower limits.
- **prismatic**: Translational joint (e.g., a sliding joint).
- **continuous**: Similar to revolute but with continuous rotation, no upper and lower limits

```
<joint name="joint1" type="revolute">
  <!-- Other joint properties -->
</joint>
```

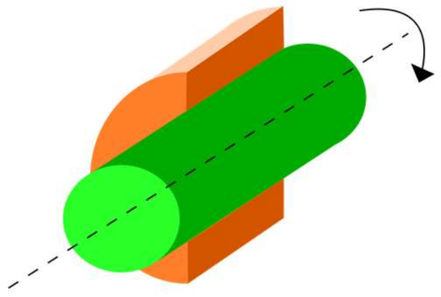
Joint Type (type attribute)

The type attribute specifies the type of joint. Common types include:

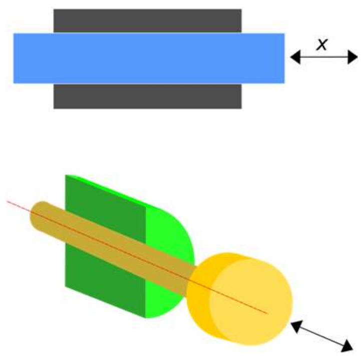
- **fixed** — this is not really a joint because it cannot move. All degrees of freedom are locked. This type of joint does not require the <axis>, <calibration>, <dynamics>, <limits> or <safety_controller>.
- **floating** — this joint allows motion for all 6 degrees of freedom.
- **planar** — this joint allows motion in a plane perpendicular to the axis.

```
<joint name="joint1" type=“planar">
  <!-- Other joint properties -->
</joint>
```

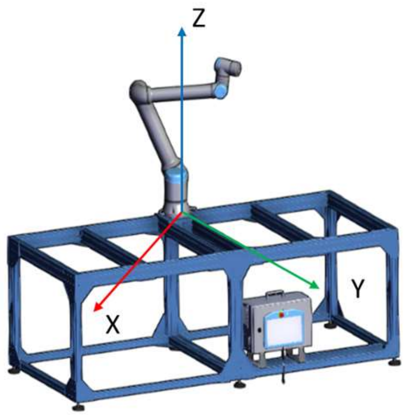
Joint Type (type attribute)



Revolute Joint



Prismatic Joint



Fixed Joint

Joint Origin (<origin> tag):

- The <origin> tag specifies the joint's position and orientation relative to its parent link.

```
<joint name="joint1" type="revolute">
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <!-- Other joint properties -->
</joint>
```

Parent and Child Links (<parent> and <child> tags)

- The <parent> tag specifies the name of the parent link.
- The <child> tag specifies the name of the child link.

```
<joint name="joint1" type="revolute">
  <parent link="parent_link"/>
  <child link="child_link"/>
  <!-- Other joint properties -->
</joint>
```

Joint Axis (<axis> tag):

- The <axis> tag defines the joint axis. For example, in a revolute joint, this axis represents the axis of rotation.

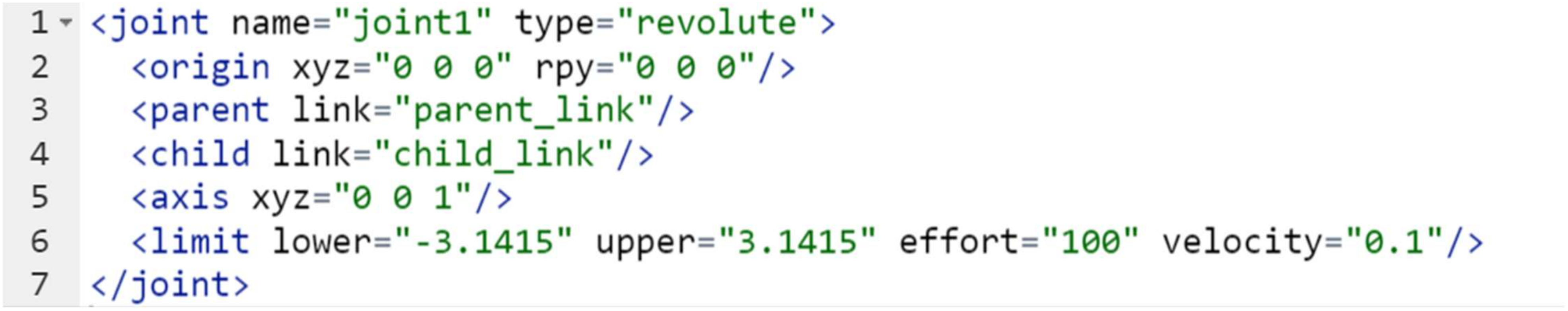
```
<joint name="joint1" type="revolute">
  <axis xyz="0 0 1"/>
  <!-- Other joint properties -->
</joint>
```

Joint Limits (<limit> tag)

- The <limit> tag specifies constraints on joint motion, such as position limits, effort limits, and velocity limits.

```
<joint name="joint1" type="revolute">
  <limit lower="-3.1415" upper="3.1415" effort="100" velocity="0.1"/>
  <!-- Other joint properties -->
</joint>
```

Example



```
<joint name="joint1" type="revolute">
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <parent link="parent_link"/>
  <child link="child_link"/>
  <axis xyz="0 0 1"/>
  <limit lower="-3.1415" upper="3.1415" effort="100" velocity="0.1"/>
</joint>
```

XACRO

Xacro is a XML macro language used to simplify the authoring of XML files in the context of robot description in the Robot Operating System (ROS). ROS is a widely used framework for building robotic systems, and Xacro provides a way to define robot models using XML syntax with macros, making it easier to generate and manage complex robot descriptions.

The term "xacro" stands for "XML Macros", and it allows users to create parameterized XML files by defining reusable macros. This is particularly useful when working with robot models that have repetitive structures or when parameters need to be easily modified without directly editing the XML files.

Reusable Macros

- Macros are predefined blocks of code that encapsulate a specific functionality or behavior and can be used multiple times throughout a program or document. These macros are designed to promote code reuse and modularity by allowing developers to define common operations or structures once and then use them wherever needed.

- In Xacro, reusable macros are defined using XML syntax and can include parameters to make them customizable. These macros can represent complex robot components, such as joints, links, sensors, or entire robot modules. By encapsulating these components in macros, developers can easily instantiate them multiple times with different parameters, reducing redundancy and making the code more concise and maintainable.

XACRO

Xacro files typically have the ".xacro" extension and are preprocessed to generate standard XML files that can be used within ROS. This preprocessing step replaces macro invocations with their expanded definitions, allowing for more flexible and maintainable robot descriptions.

```
<?xml version="1.0"?>
<robot
        xmlns:xacro="http://wiki.ros.org/xacro">
        <xacro:include filename="$(find ur_description)/urdf/inc/ur_transmissions.xacro" />
        <xacro:include filename="$(find ur_description)/urdf/inc/ur_common.xacro" />
        <xacro:macro name="ur_robot" params="
    prefix
    joint_limits_parameters_file
    kinematics_parameters_file
    physical_parameters_file
    visual_parameters_file
    transmission_hw_interface:=hardware_interface/PositionJointInterface
    safety_limits:=false
    safety_pos_margin:=0.15
    safety_k_position:=20"
  >
                <!-- Load configuration data from the provided .yaml files -->
                <xacro:read_model_data
      joint_limits_parameters_file="${joint_limits_parameters_file}"
      kinematics_parameters_file="${kinematics_parameters_file}"
      physical_parameters_file="${physical_parameters_file}"
      visual_parameters_file="${visual_parameters_file}"/>
                <!-- Add URDF transmission elements (for ros_control) -->
                <xacro:ur_arm_transmission prefix="${prefix}" hw_interface="${transmission_hw_interfa
                <!-- links: main serial chain -->
                <link name="${prefix}base_link"/>
                <link name="${prefix}base_link_inertia">
                        <visual>
                                <origin xyz="0 0 0" rpy="0 0 ${pi}"/>
                                <geometry>
                                        <mesh filename="${base_visual_mesh}"/>
                                </geometry>
                                <material name="${base_visual_material_name}">
                                        <color rgba="${base_visual_material_color}"/>
                                </material>
                </visual>
                <collision>
                        <origin xyz="0 0 0" rpy="0 0 ${pi}"/>
                        <geometry>
                                <mesh filename="${base_collision_mesh}"/>
                        </geometry>
                </collision>
                <xacro:cylinder_inertial radius="${base_inertia_radius}" length="${base_inert
                        <origin xyz="0 0 0" rpy="0 0 0" />
                </xacro:cylinder_inertial>
        </link>
        <!-- joints: main serial chain -->
        <joint name="${prefix}base_link-base_link_inertia" type="fixed">
                <parent link="${prefix}base_link" />
                <child link="${prefix}base_link_inertia" />
                <!-- 'base_link' is REP-103 aligned (so X+ forward), while the internal
     frames of the robot/controller have X+ pointing backwards.
     Use the joint between 'base_link' and 'base_link_inertia' (a dummy
     link/frame) to introduce the necessary rotation over Z (of pi rad).
-->
                <origin xyz="0 0 0" rpy="0 0 ${pi}" />
        </joint>
        <joint name="${prefix}shoulder_pan_joint" type="revolute">
                <parent link="${prefix}base_link_inertia" />
                <child link="${prefix}shoulder_link" />
                <origin xyz="${shoulder_x} ${shoulder_y} ${shoulder_z}" rpy="${shoulder_roll}
                <axis xyz="0 0 1" />
                <limit lower="${shoulder_pan_lower_limit}" upper="${shoulder_pan_upper_limit}
   effort="${shoulder_pan_effort_limit}" velocity="${shoulder_pan_velocity_limit}"/>
                <xacro:if value="${safety_limits}">
                        <safety_controller soft_lower_limit="${shoulder_pan_lower_limit + saf
                </xacro:if>
                <dynamics damping="0" friction="0"/>
        </joint>
   </xacro:macro>
</robot>
```

URDF vs XACRO

URDF (Unified Robot Description Format):

URDF is an XML-based file format used to describe the physical and kinematic properties of robots in ROS.

It provides a straightforward way to define robot links, joints, sensors, and other components.

URDF files can be verbose, especially for complex robot models, as they require explicit XML markup for each component.

URDF files are directly parsed by ROS tools for visualization, simulation, and control.

Xacro (XML Macros):

Xacro is an extension of URDF that allows for more flexible and concise robot descriptions through the use of macros.

Xacro files have the ".xacro" extension and are preprocessed to generate URDF files.

Xacro macros enable code reuse and parameterization, making it easier to manage and modify robot descriptions.

Xacro allows developers to define reusable components and parameterize them, reducing redundancy and improving code maintainability.

Summary

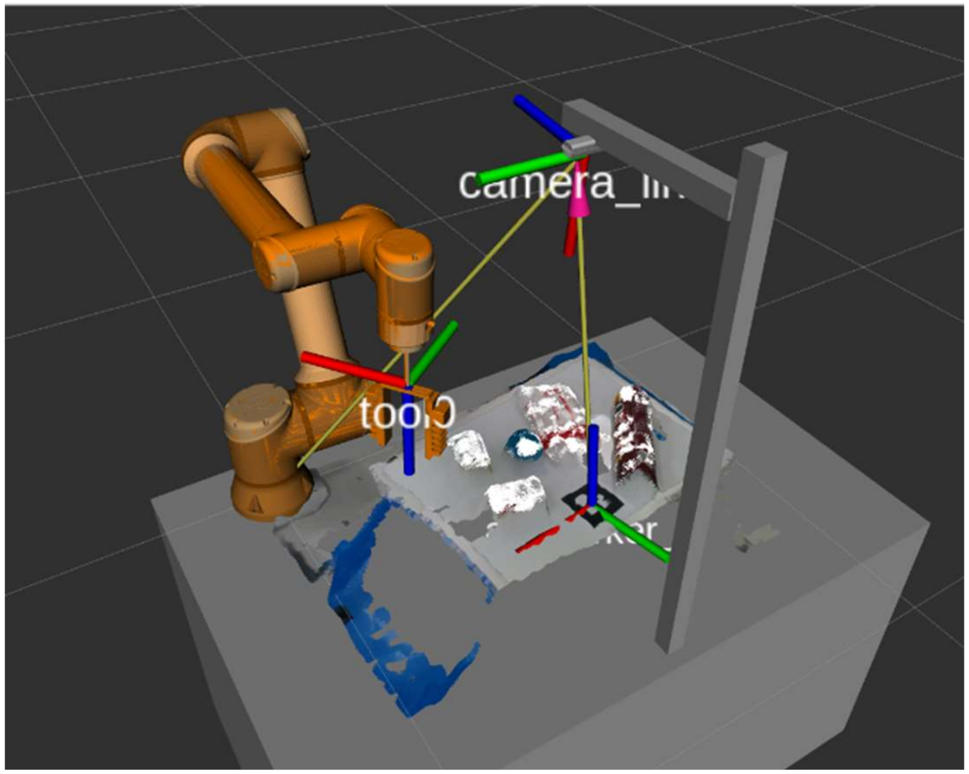
URDF (Unified Robot Description Format) and Xacro are both used in the Robot Operating System (ROS) for describing robot models, but they serve slightly different purposes and have different syntaxes. URDF is the base format for describing robot models in ROS, while Xacro provides a more powerful and flexible way to define these models by introducing macros and parameterization. Xacro files are processed to generate URDF files, which are then used by ROS tools for various tasks such as visualization, simulation, and control.

Practice

- Designing a URDF for a robotic bin sorting application involves defining the structure, joints, and sensors of the robot.

- Robot: UR10e robot
- Gripper: Robotic 2F-145
- Camera: Realsense d435i Camera
- Fixture : Camera stand. Table, Conveyor
- All the CAD data are given.

Layout


camera_lin
tool0