{Learn, Create, Innovate};

ROS

Robot
Modelling/Visualisation Tools

Manchester Robotics

Robot Descriptions

URDF Files

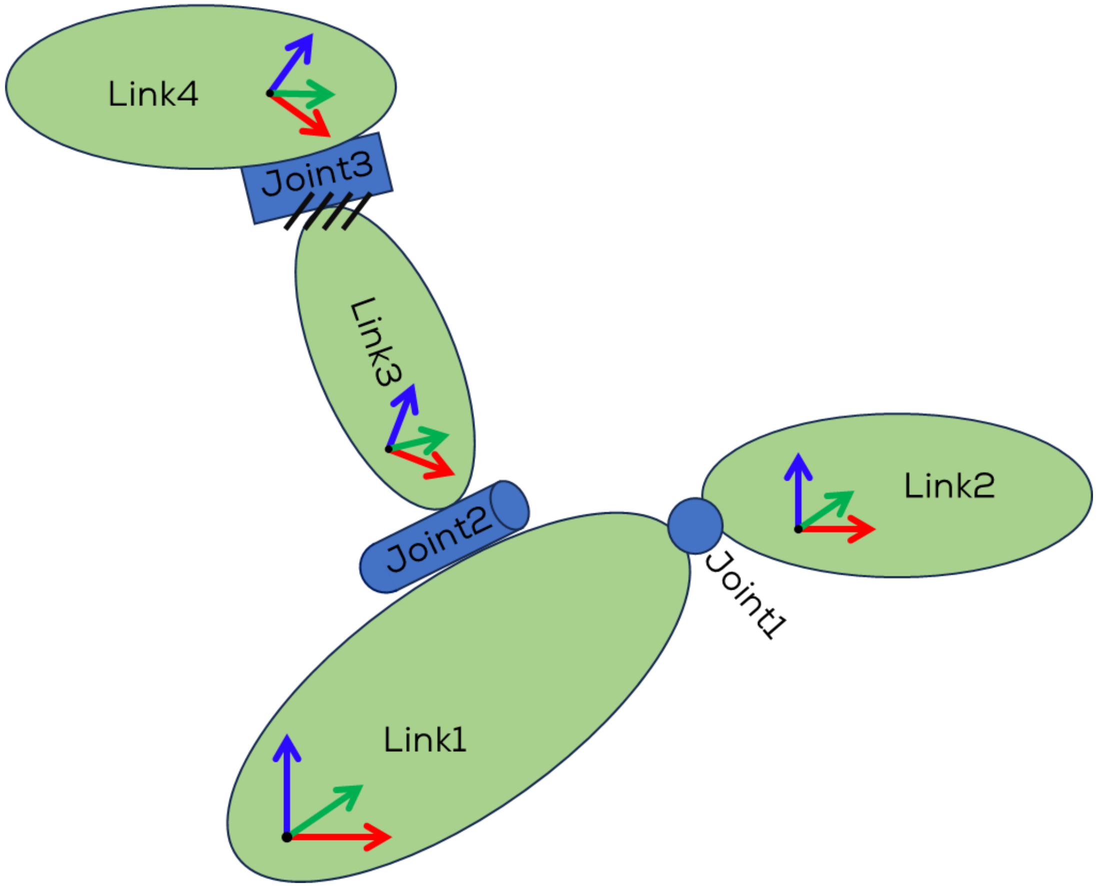
Manchester Robotics

{Learn, Create, Innovate};

URDF Files

- URDF (Unified Robot Description Format) files are XML-based files used to describe a robot's structure, kinematics, dynamics, and visual appearance in the context of robotics and simulation.

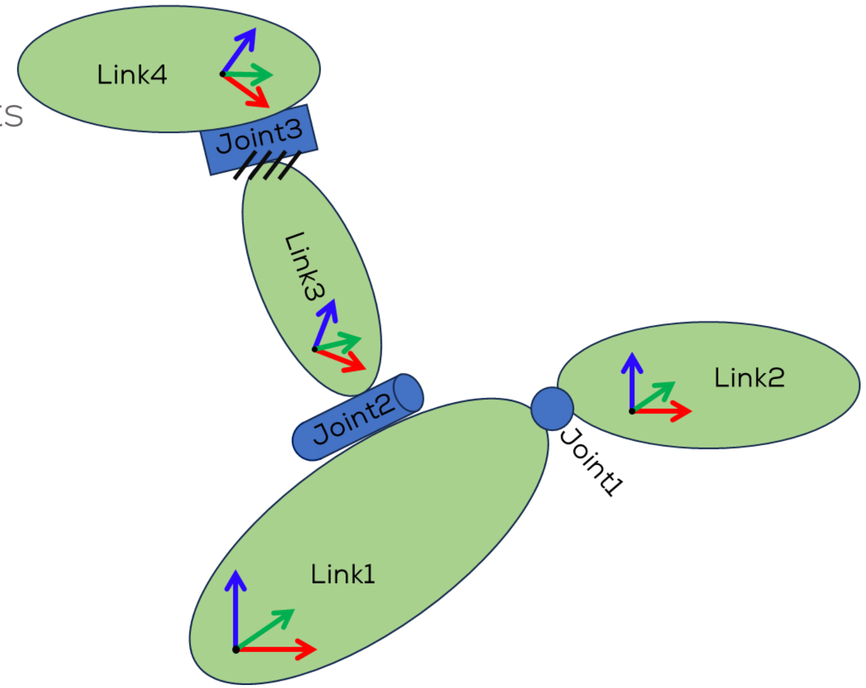
- URDF is commonly used in robotics frameworks like ROS (Robot Operating System) to represent robots and their components.

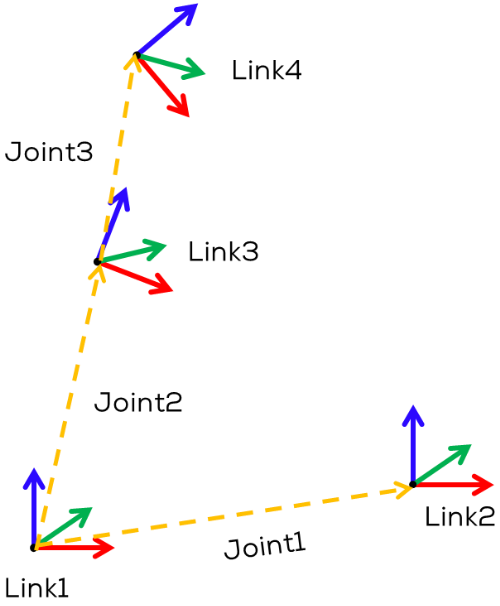


URDF Files

- URDF files provide a standardised format to describe robot models, allowing simulation and visualisation tools to load and manipulate robots accurately.

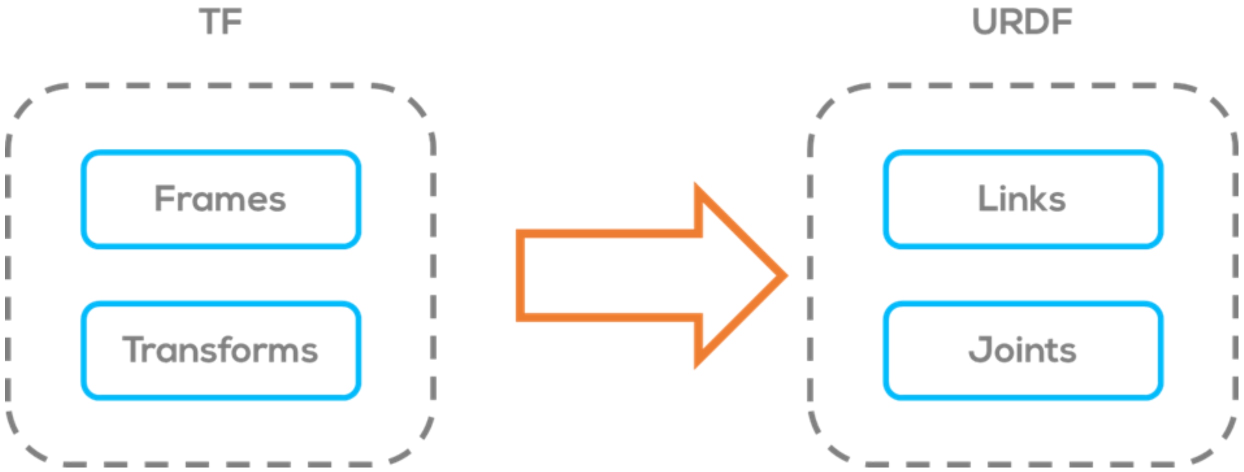
- They are widely used in the robotics community and play a crucial role in developing and integrating robotic systems.





URDF Files

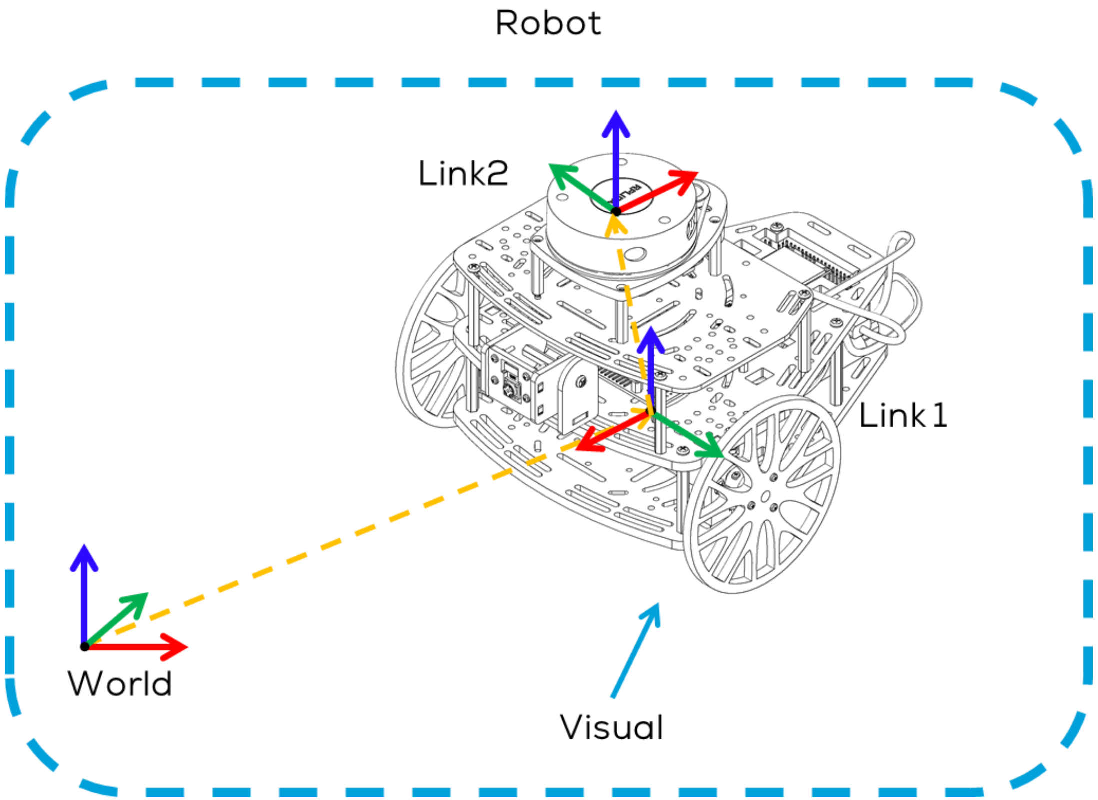
- URDF describes a robot as a tree of links, that are connected by joints.
  - The links represent the physical components of the robot, and the joints represent how one link moves relative to another link, defining the location of the links in space.
- This concept can be related to the TF (Transforms) concepts of frames and links as follows
  - Frames -> Links
  - Transforms -> Joints



URDF Files

Contents

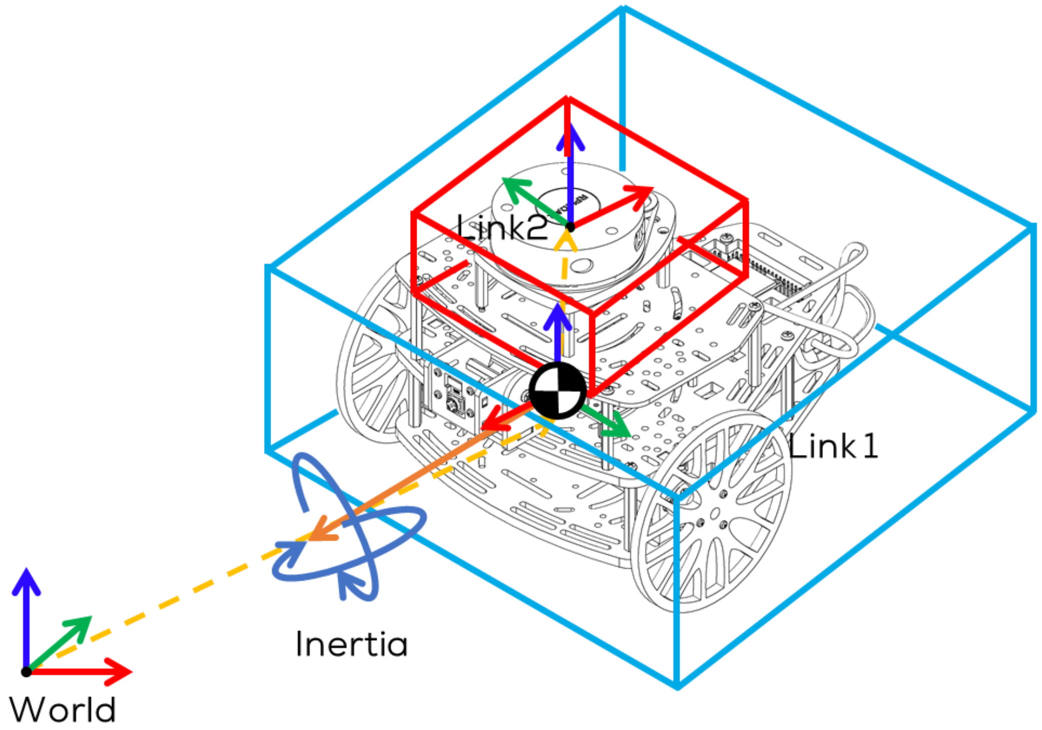
- **Robot:** The root element of the URDF file, representing the entire robot. It encapsulates all the other elements.
- **Links:** A link represents a rigid body or a component of the robot. It describes the visual, inertial, and collision properties of the link. Each link may have one or more visual and collision elements associated with it.
- **Visual:** Defines the visual appearance of a link, including its geometry (shape), material properties (e.g., color, texture), and transformation (position and orientation) relative to the link.



URDF Files

Contents

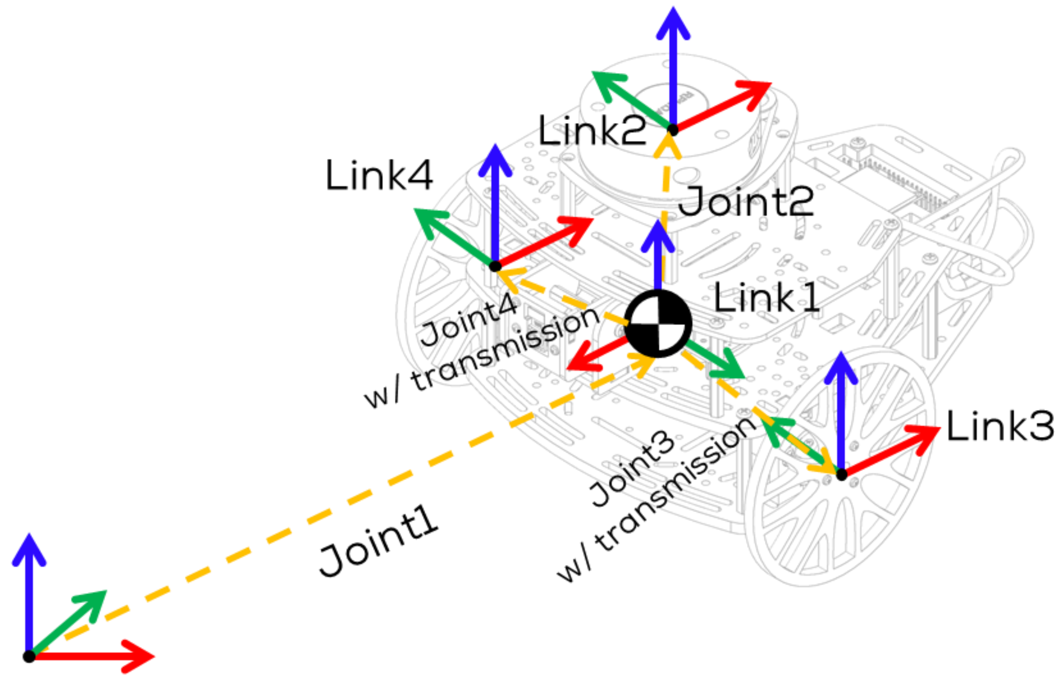
- **Collision:** Specifies the collision properties of a link, such as the collision geometry and its transformation relative to the link.
- **Inertial:** Describes the inertial properties of a link, such as mass, center of mass, and moments of inertia. These properties are used for dynamics calculations.



URDF Files

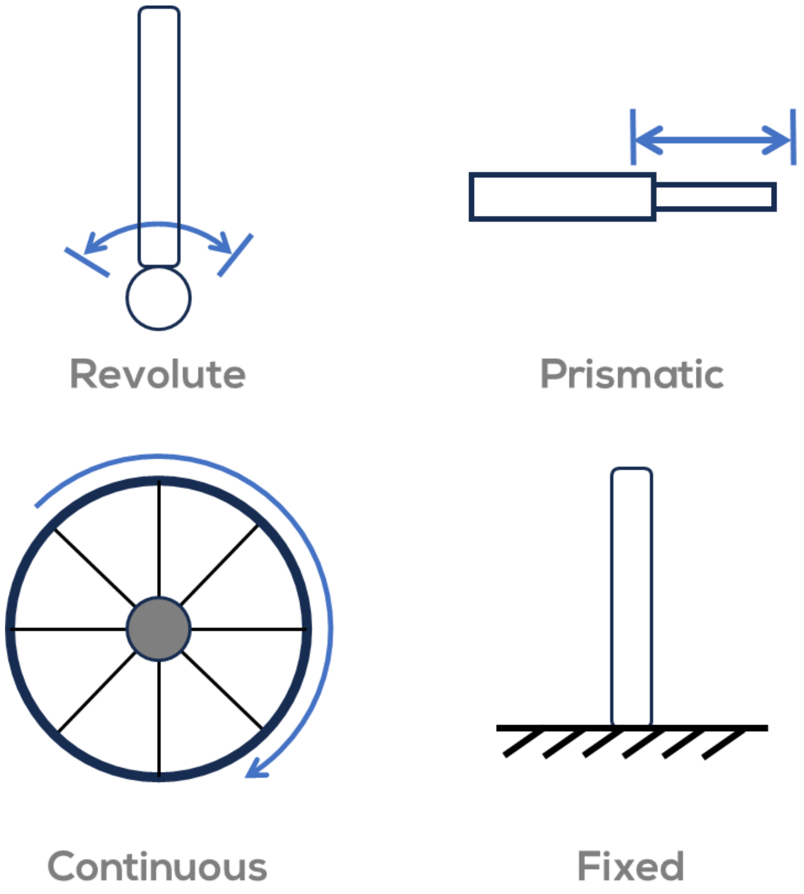
Contents

- **Joints:** Joints define the kinematic connections between links. They specify the type of joint (e.g., revolute, prismatic) and its properties (e.g., limits, axis, origin).
- **Transmission:** A transmission element connects a joint to an actuator, specifying how the joint motion is driven.
- **Plugins:** URDF supports plugins, allowing users to extend the capabilities of the robot description. Plugins can provide additional features like custom collision checking or dynamic properties.



URDF Files: Joints

- Revolute – A rotational motion, with minimum/maximum angle limits.
- Continuous – A rotational motion with no limit (e.g. a wheel).
- Prismatic – A linear sliding motion, with minimum/maximum position limits.
- Fixed – The child link is rigidly connected to the parent link. This is what we use for those “convenience” links.
  - *Some other joints might be available (some deprecated). Some only work on Gazebo.



Syntax

- **XML Declaration:** The file begins with an XML declaration that specifies the version of XML being used. For URDF, it is typically <?xml version="1.0"?>.
- **Root Element:** The root element of the URDF file is <robot>. It encapsulates all the other elements in the file and typically includes attributes like name to specify the robot's name.
- **Links:** Inside the <robot> element, you define the robot's links using the <link> element. Each <link> represents a component or a rigid body of the robot. It may have attributes like name.

```xml
<?xml version="1.0"?>

<robot name="single_link_arm">

  <link name="link1">
    <visual>
      <geometry>
        <cylinder length="0.4" radius="0.02" />
      </geometry>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.4" radius="0.02" />
      </geometry>
    </collision>
    <inertial>
      <mass value="1" />
      <origin xyz="0 0 0" rpy="0 0 0" />
      <inertia ixx="0.01" iyy="0.01" izz="0.01" ixy="0" ixz="0" iyz="0" />
    </inertial>
  </link>

  <joint name="joint1" type="revolute">
    <origin xyz="0 0 0" rpy="0 0 0" />
    <parent link="world" />
    <child link="link1" />
    <axis xyz="0 0 1" />
    <limit lower="-1.57" upper="1.57" effort="5" velocity="2" />
  </joint>

</robot>
```

Syntax

- Visuals: Within each <link>, you can define the visual properties using the <visual> element. It includes attributes like names and contains child elements to describe the visual geometry, material properties, and transformations.
- Collisions: Similar to visuals, collisions are defined using the <collision> element within each <link>. It represents the collision geometry and properties associated with a link.
- Inertial: The <inertial> element is used to specify the inertial properties of a link. It includes child elements such as <mass> to define the mass, <inertia> to specify the moments of inertia, and <origin> to describe the position and orientation of the inertial frame.

```xml
<?xml version="1.0"?>

<robot name="single_link_arm">

  <link name="link1">
    <visual>
      <geometry>
        <cylinder length="0.4" radius="0.02" />
      </geometry>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.4" radius="0.02" />
      </geometry>
    </collision>
    <inertial>
      <mass value="1" />
      <origin xyz="0 0 0" rpy="0 0 0" />
      <inertia ixx="0.01" iyy="0.01" izz="0.01" ixy="0" ixz="0" iyz="0" />
    </inertial>
  </link>

  <joint name="joint1" type="revolute">
    <origin xyz="0 0 0" rpy="0 0 0" />
    <parent link="world" />
    <child link="link1" />
    <axis xyz="0 0 1" />
    <limit lower="-1.57" upper="1.57" effort="5" velocity="2" />
  </joint>

</robot>
```

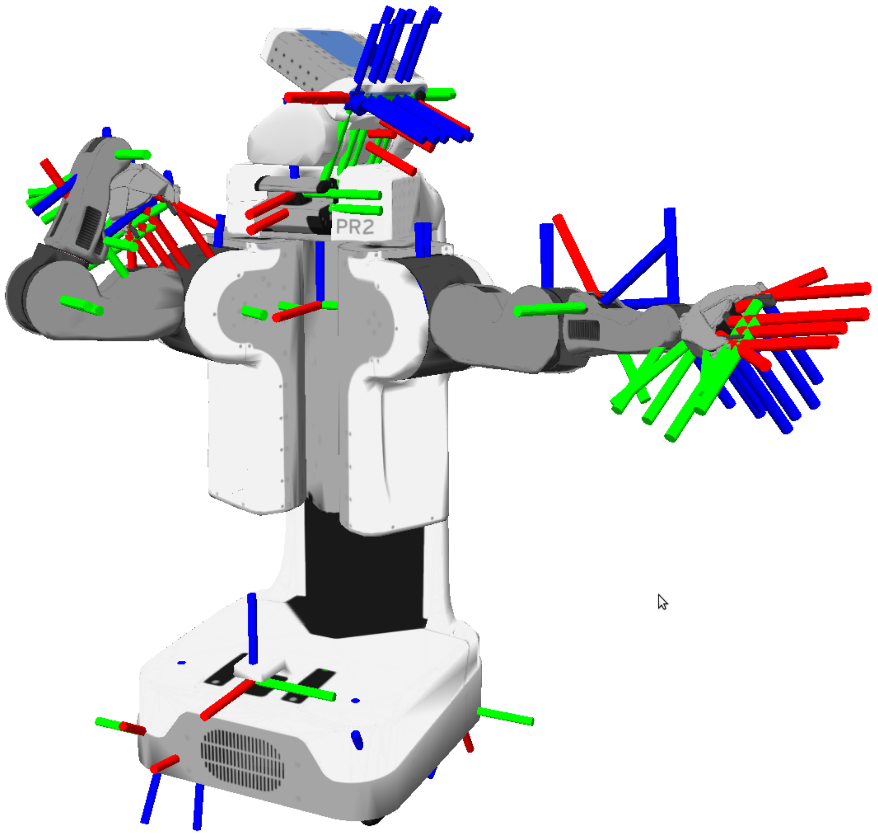
Syntax

- **Joints:** Joints are defined within <robot> using the <joint> element. Each joint element represents a kinematic connection between two links. Joints have attributes like name, type, and contain child elements to define properties like limits, axis, and origin.
- **Transmission:** The <transmission> element connects a joint to an actuator, specifying how the joint motion is driven. It includes child elements like <type>, <joint> (to specify the joint being controlled), and <actuator> (to define the actuator properties).
- **Plugins:** URDF supports plugins for extending its capabilities. Plugins are added as <plugin> elements within the relevant sections of the URDF file. They can provide additional features or custom functionality.

```xml
<?xml version="1.0"?>

<robot name="single_link_arm">

  <link name="link1">
    <visual>
      <geometry>
        <cylinder length="0.4" radius="0.02" />
      </geometry>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.4" radius="0.02" />
      </geometry>
    </collision>
    <inertial>
      <mass value="1" />
      <origin xyz="0 0 0" rpy="0 0 0" />
      <inertia ixx="0.01" iyy="0.01" izz="0.01" ixy="0" ixz="0" iyz="0" />
    </inertial>
  </link>

  <joint name="joint1" type="revolute">
    <origin xyz="0 0 0" rpy="0 0 0" />
    <parent link="world" />
    <child link="link1" />
    <axis xyz="0 0 1" />
    <limit lower="-1.57" upper="1.57" effort="5" velocity="2" />
  </joint>

</robot>
```

Robot State Publisher

- URDF files require a translator, so that ROS can use them.
- Different translators have been developed to "translate" URDF files into TF functions.
- ROS has different packages to manage URDF files.
- To transform the URDF files to robot states, visualise the robot and transform data between different coordinate frames; ROS uses a package called *"robot_state_publisher"*.
- This package allows you to publish the state of a robot to tf2.



Robot State Publisher

- Once the state gets published, it is available to all components in the system that also use tf2.
- The package takes the joint angles of the robot as input and publishes the 3D poses of the robot links, using a kinematic tree model of the robot.
- The package can both be used as a library and as a ROS node.

Usage as a ROS Node:

- *"robot_state_publisher"* uses the URDF specified by the parameter *"robot_description"* and the joint positions from the topic *"joint_states"* to calculate the forward kinematics of the robot and publish the results via tf2.

```
<?xml version="1.0"?>
<launch>

    <arg name="fixed_joint_test" default="$(find joints_act)/urdf/fixed_ex.urdf"/>

    <param name="robot_description" command="cat $(arg fixed_joint_test)" />
    <node pkg="robot_state_publisher" type="robot_state_publisher" name="link_ex_pub" >
    </node>

</launch>
```

Activity 1

Creating a simple URDF file

{Learn, Create, Innovate};

Activity 1

1. Make a new package called "joints_act" with the following library packages

geometry_msgs, nav_msgs, rospy, sensor_msgs, std_msgs, tf2_ros, tf_conversions visualization_msgs

```
catkin_create_pkg joints_act geometry_msgs, nav_msgs,
rospy, sensor_msgs, std_msgs, tf2_ros, tf_conversions
visualization_msgs
```

2. Create a "urdf" folder inside the previously created package and a URDF file "fixed_ex.urdf" inside

```
mkdir urdf && touch urdf/fixed_ex.urdf
```

3. Paste the following code inside the "fixed_ex.urdf" file.
    - Code can be found on GitHub.
4. Create a launch file

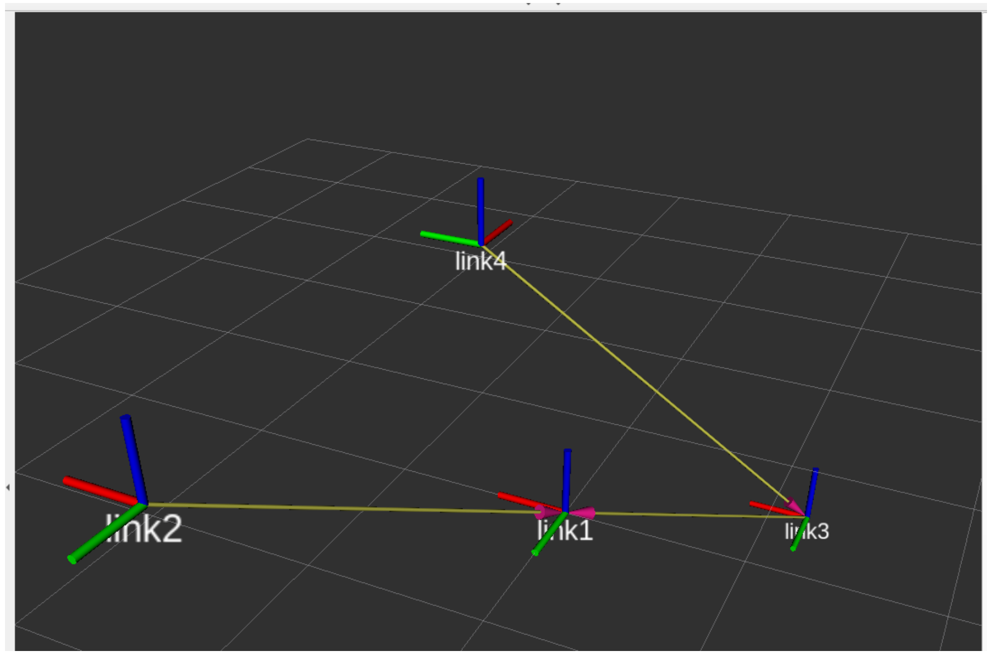
```
mkdir launch && touch launch/fixed.launch
```

```
<?xml version="1.0"?>

<robot name="link_example">

  <link name="link1" />
  <link name="link2" />
  <link name="link3" />
  <link name="link4" />

    <joint name="joint1" type="fixed">
        <parent link="link1"/>
        <child link="link2"/>
        <origin xyz="1 2 1" rpy="0 0 0" />
    </joint>

    <joint name="joint2" type="fixed">
        <parent link="link1"/>
        <child link="link3"/>
        <origin xyz="-1 -2 -1" rpy="0 0 0" />
    </joint>

    <joint name="joint3" type="fixed">
        <parent link="link3"/>
        <child link="link4"/>
        <origin xyz="2 1 2" rpy="0 0 -1.57" />
    </joint>

</robot>
```

Activity 1

5. Paste the following code inside the launch file

```
<?xml version="1.0"?>
<launch>

    <arg name="fixed_joint_test" default="$(find 
joints_act)/urdf/fixed_ex.urdf"/>
    <param name="robot_description" command="cat $(arg fixed_joint_test)" 
/>
    <node pkg="robot_state_publisher" type="robot_state_publisher" 
name="link_ex_pub" >
    </node>

<node name="rviz" pkg="rviz" type="rviz“ required="true" />
</launch>
```



6. Add the marker
   - Press Add
   - ››By display type››TF

Activity 2

Creating a movable joint using URDF

{Learn, Create, Innovate};

Activity 2

1. Create a new "urdf" named "continuos_ex.urdf" inside the previously created package "joints_act"

```
touch urdf/ continuos_ex.urdf
```

2. Paste the following code inside the "continuos_ex.urdf" file.
   - Code can be found on GitHub.
3. Create a launch file

```
touch launch/continuos.launch
```

5. Add the marker
   - Press Add
   - ››By display type››TF
6. Run the tf_tree…

```
rosrun rqt_tf_tree rqt_tf_tree
```

7. Why nothing appears? … Because ROS does not know the state of the movable joints! (that is why ROS does not show the movable joints)

```
<?xml version="1.0"?>
<launch>

    <arg name="continuos_ex" default="$(find
joints_act)/urdf/continuos_ex.urdf"/>
    <param name="robot_description" command="cat $(arg continuos_ex)" />
    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="continuos_test_pub" >
    </node>

<node name="rviz" pkg="rviz" type="rviz" required="true" />

</launch>
```

Joint State Publisher

- ROS does not know the *"state"* of the joints in a robot i.e., position, velocity and effort.
- The Joint State Publisher is a package in Robot Operating System (ROS), designed to publish joint state information for a robot, so ROS can " know " each joint's state at a point in time.
- It's a critical component in ROS for robotics applications because it allows us to test and publish the states of the robot's joints.
- When using non-fixed joints, ROS will not publish the TF information unless they are "known"; in other words, ROS needs to know the state of the joint. To do this, the state needs to be published in the */joint_state* topic.

```
student@ubuntu:~$ rostopic list
/clicked_point
/initialpose
/joint_states
/move_base_simple/goal
/rosout
/rosout_agg
/tf
/tf_static
```

```
student@ubuntu:~$ rostopic info /joint_states
Type: sensor_msgs/JointState

Publishers: None

Subscribers:
 * /continuos_test_pub (http://ubuntu:39191/)
```

Joint State Publisher

- This package publishes sensor_msgs/JointState messages for a robot.
- The package reads the robot_description parameter from the parameter server, finds all the non-fixed joints and publishes a JointState message with all those joints defined.
- The joint state publisher, publishes default positional values to the joints, to the robot state publisher, using the JointState message, in the /joint_state topic.

- Remember!
  - The joint state publisher is not intended to send commands to a joint (except in this activity, where is used to test our URDF file) it is intended to read and visualise the state of joints from the different robot sensors.
  - When testing our robot, in RVIZ, the joint_state_publisher allows us to make a simple test by publishing into the /joint_state topic and moving the Joint; this, however, is only for testing purposes and should not be used when reading real data!

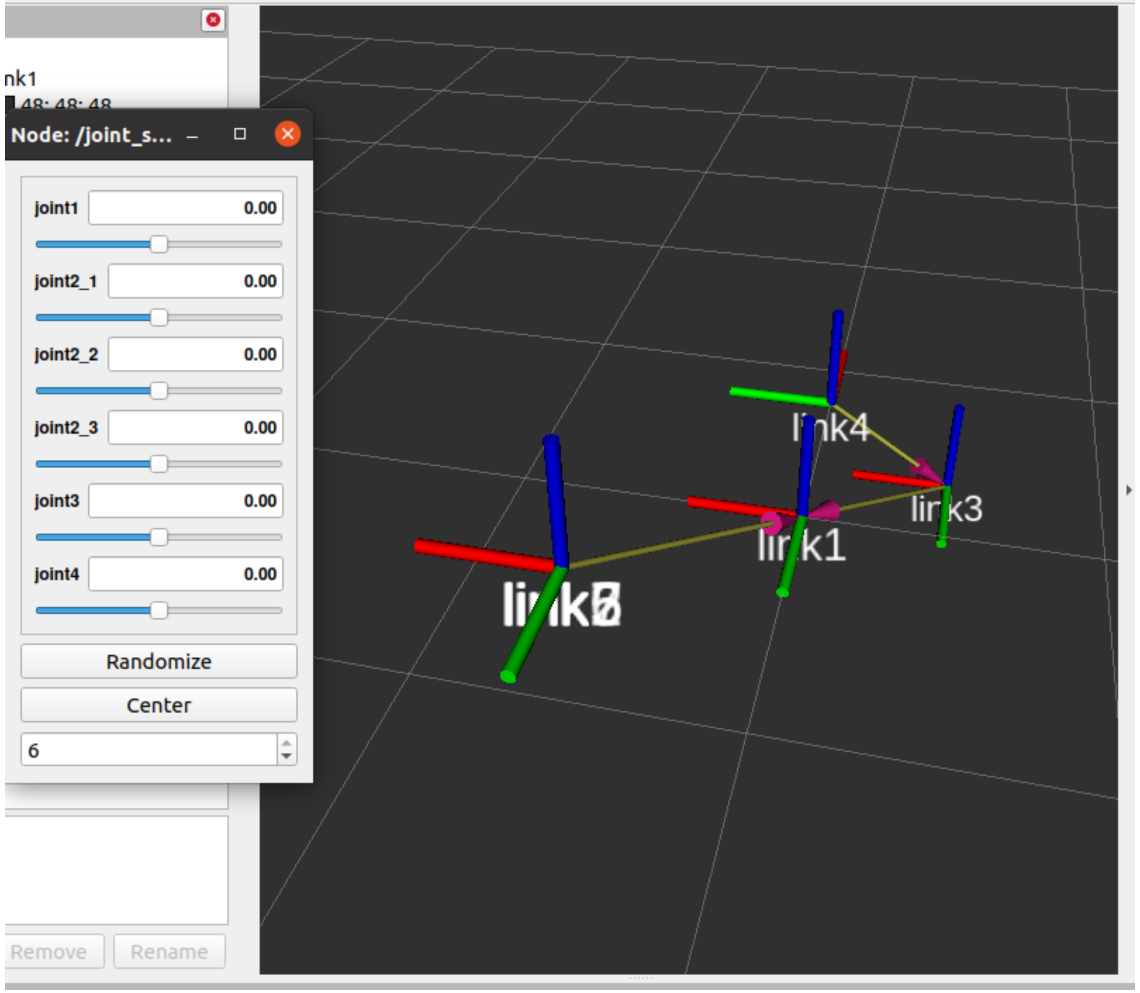
```
header:
  seq: 777
  stamp:
    secs: 1695199400
    nsecs: 576087474
  frame_id: ''
name:
  - joint1
  - joint2_1
  - joint2_2
  - joint2_3
  - joint3
  - joint4
position: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
velocity: []
effort: []
```

Activity 2

8. Change the launch file to add the joint state publisher

```
<?xml version="1.0"?>
<launch>

    <arg name="continuos_ex" default="$(find
joints_act)/urdf/continuos_ex.urdf"/>
    <param name="robot_description" command="cat $(arg continuos_ex)" />
    <node pkg="robot_state_publisher" type="robot_state_publisher"
name="continuos_test_pub" >
    </node>

 <node name="joint_state_publisher_gui" pkg="joint_state_publisher_gui"
type="joint_state_publisher_gui">
    </node>

<node name="rviz" pkg="rviz" type="rviz" required="true" />

</launch>
```

9. Use the GUI to move the joints!



Activity 3

Making your own joint state publisher

{Learn, Create, Innovate};

Activity 3

1. Create a node named “continuosJoint.py” inside the previously created package “joints_act”

```
touch scripts/ continuosJoint.py
```

2. Give execution permission and add it to the CMakeLists.txt.
3. Copy the code in GitHub for this node.
4. Compile the node.
5. Add the node to the launch file and comment the joint_state_publisher.

```
 <!--<node name="joint_state_publisher_gui" 
pkg="joint_state_publisher_gui" type="joint_state_publisher_gui">
    </node>-->

    <node pkg="joints_act" type="continuousJoint.py" 
name="continuous_joint_pub" />
```

6. Run the launch file

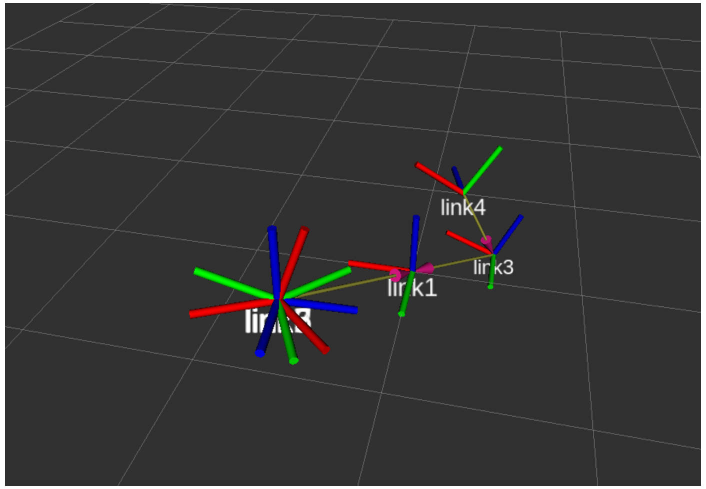
- This funntion, declares the JointState message and its initial values.
- The message allows the joints to be treated as a list, with its corresponding state values.
  - contJoints.name[0]
    - contJoints.position[0]
    - contJoints.velocity[0]
    - contJoints.effort[0]

```
from sensor_msgs.msg import JointState

# Declare the output Messages
contJoints = JointState()

# Declare the output Messages
def init_joints():
    contJoints.header.frame_id = "link1"
    contJoints.header.stamp = rospy.Time.now()
    contJoints.name.extend(["joint1", "joint2_1", "joint2_2", 
"joint2_3", "joint3", "joint4" ])
    contJoints.position.extend([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
    contJoints.velocity.extend([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
    contJoints.effort.extend([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
```

Activity 3

- Inside the main loop of the node, the positional values for each joint (6 joints according to the URDF file) are updated using the ROS Time.
- RVIZ only handles positional values, not velocity or effort values, therefore only the position of each joint in the list of joints is being updated.
- Since they are continuous joints (as declared in the previous URDF), the values must be wrapped to PI, before being published.
- The publishing of the message is done as a regular ROS message.

```
while not rospy.is_shutdown():
           t = rospy.Time.now().to_sec()
           contJoints.header.stamp = rospy.Time.now()
           contJoints.position[0] = wrap_to_Pi(t)
           contJoints.position[1] = wrap_to_Pi(0.5*t)
           contJoints.position[2] = wrap_to_Pi(0.5*t)
           contJoints.position[3] = wrap_to_Pi(0.5*t)
           contJoints.position[4] = wrap_to_Pi(0.1*t)
           contJoints.position[5] = wrap_to_Pi(t)

           joint_pub.publish(contJoints)

           loop_rate.sleep()
```



Thank you

{Learn, Create, Innovate};

MCR^2
Manchester Robotics

T&C

Terms and conditions

{Learn, Create, Innovate};

Terms and conditions