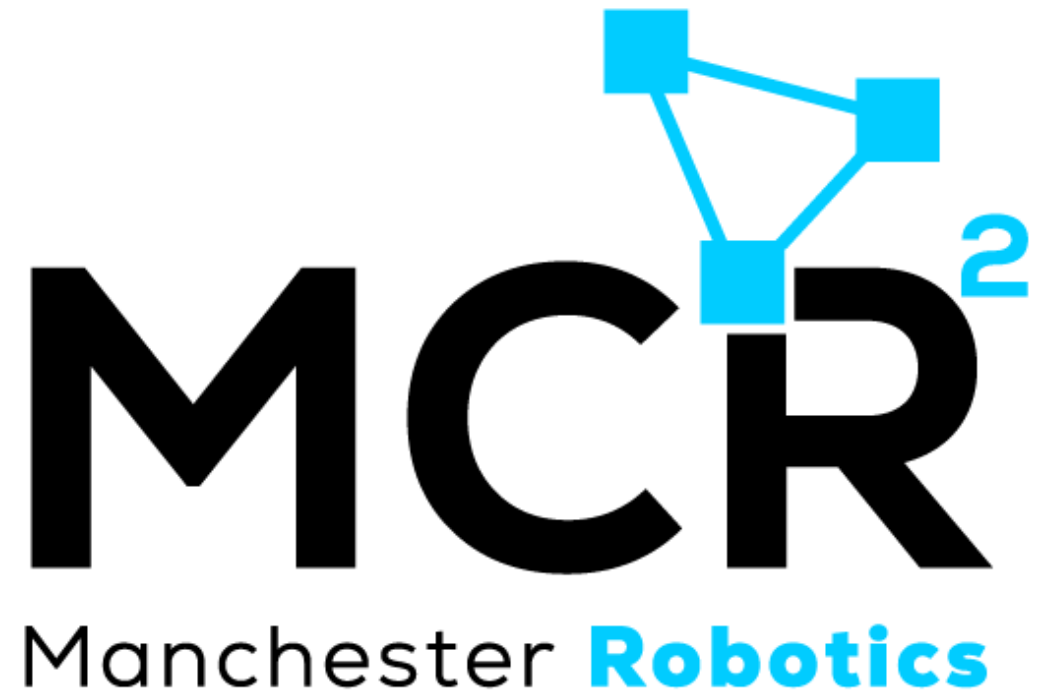


*{Learn, Create, Innovate};*

# ROS

*Robot  
Modelling/Visualisation Tools*



Robot  
Descriptions

*URDF Files*

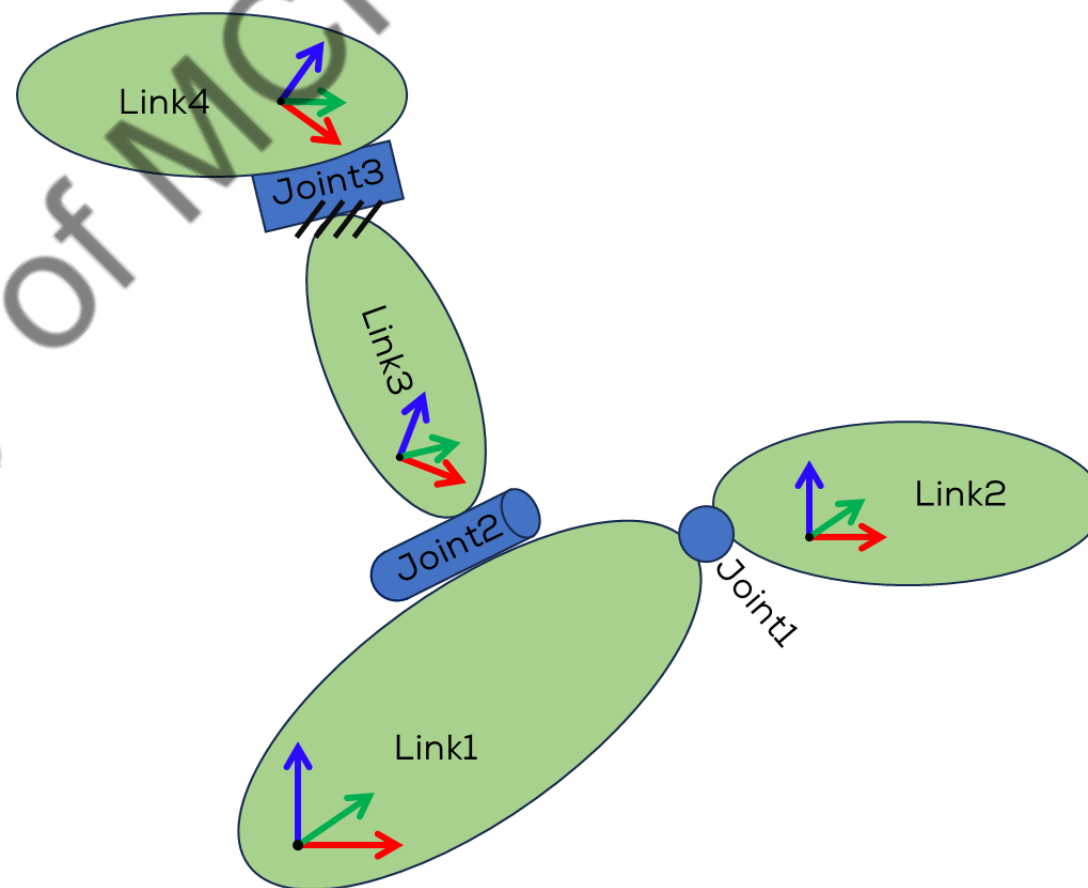
*{Learn, Create, Innovate};*

Property of MCR<sup>2</sup>

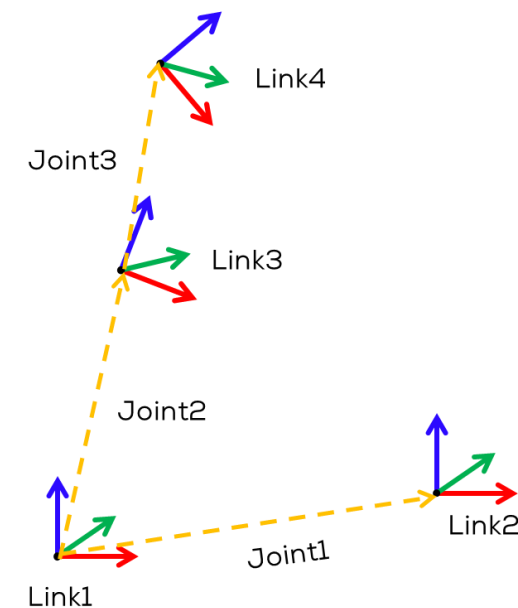
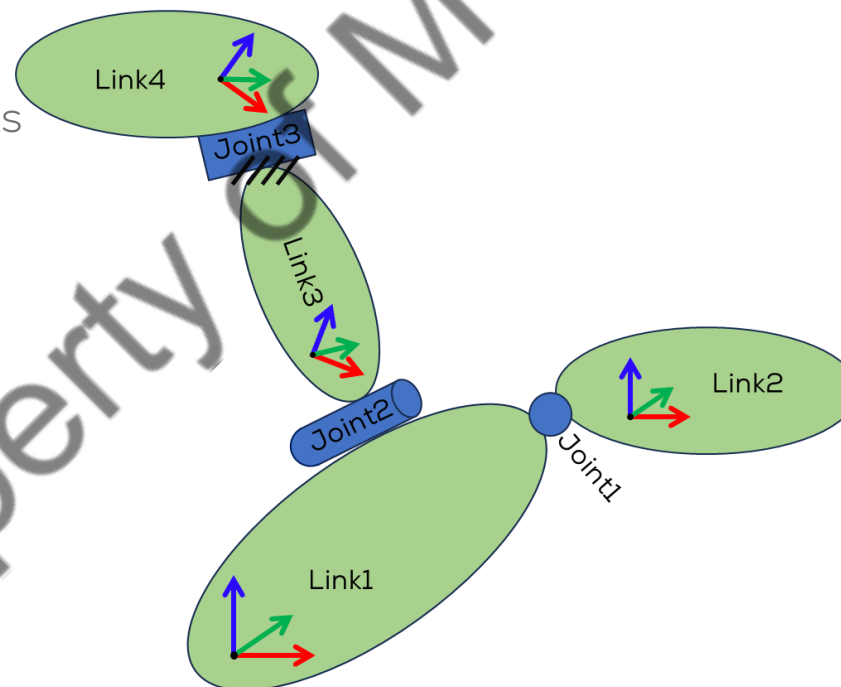


Manchester **Robotics**

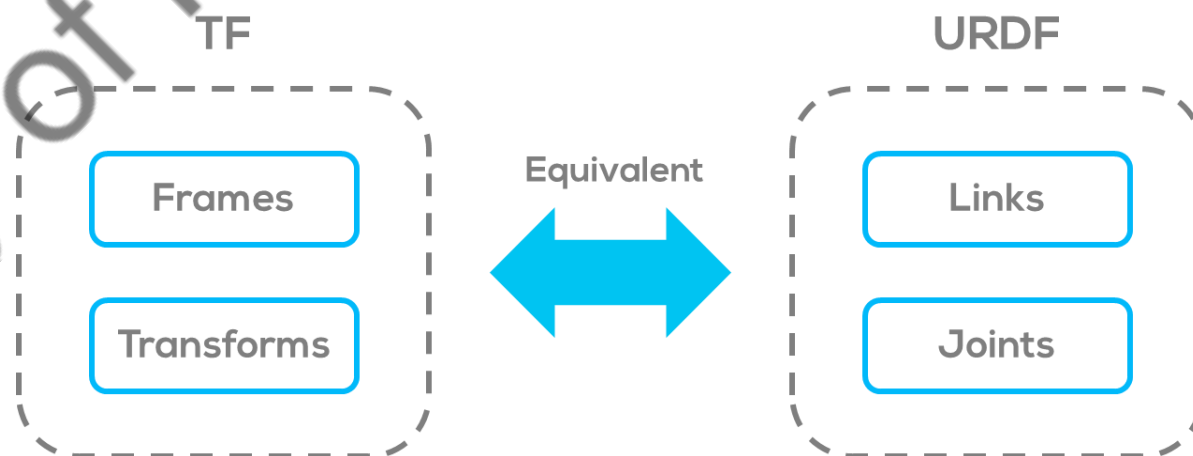
- URDF (Unified Robot Description Format) files are XML-based files used to describe a robot's structure, kinematics, dynamics, and visual appearance in the context of robotics and simulation.
- URDF is commonly used in robotics frameworks like ROS (Robot Operating System) to represent robots and their components.



- URDF files 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 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



- **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"?>`.
- **Robot:** 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** represent a component or a rigid body of the robot. It describes the visual, inertial, and collision properties of the link.
- **Joints** represents the kinematic connection between two links. They specify the type of joint (e.g., revolute, prismatic) and its properties (e.g., limits, axis, origin).

```

XML { <?xml version="1.0"?>
Robot { <robot name="multiple_joints_example">
Links {
    <link name="base_footprint">
    </link>

    <link name="base_link">
      <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
          <mesh filename="file://mesh"/>
        </geometry>
      </visual>
      <collision>
        <geometry>
          <box size="0 0 0"/>
        </geometry>
      </collision>
      <inertial>
        <mass value="1"/>
        <xacro:box_inertia mass="1" x="0" y="0" z="0" />
      </inertial>
    </link>

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

  </robot>

```

Robot  
Descriptions

*Robot State Publisher*

*{Learn, Create, Innovate};*

Property of MCR<sup>2</sup>



Manchester **Robotics**

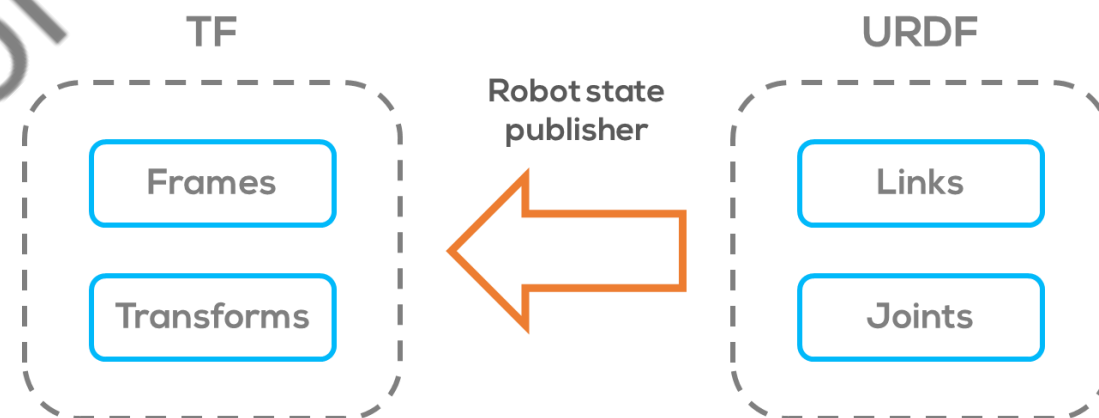


# Robot State Publisher



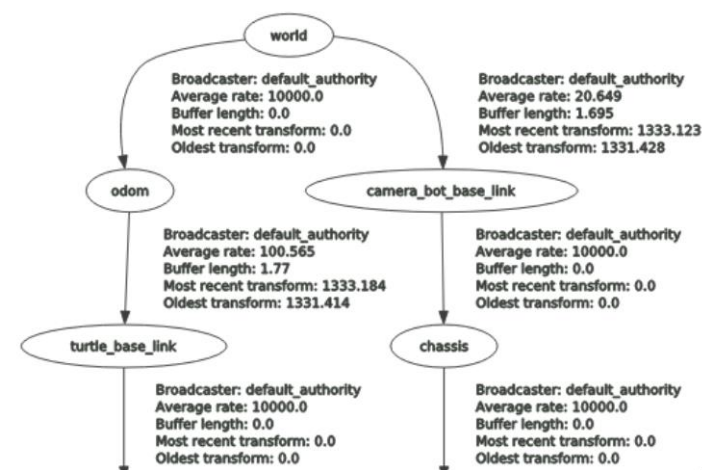
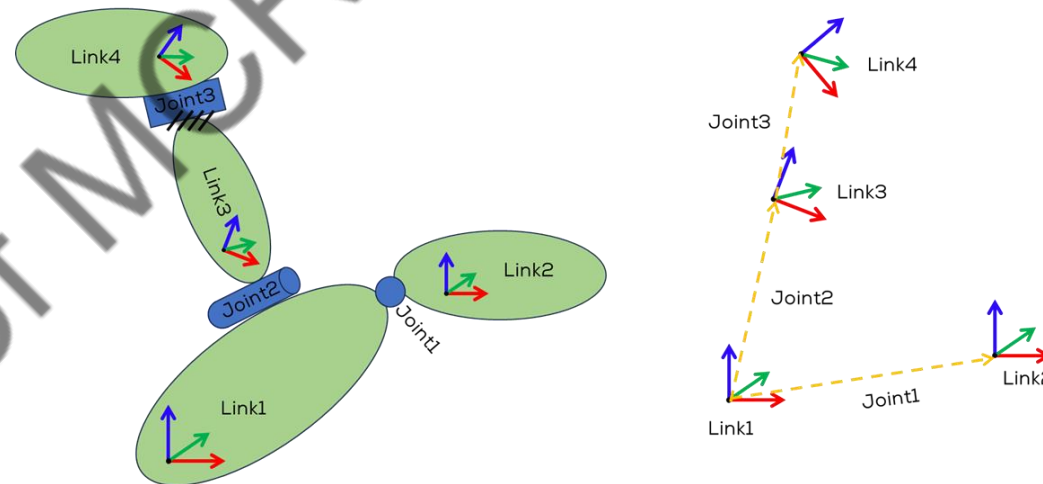
## Robot State Publisher

- URDF files require a translator, so that ROS can use them.
- Different translators have been developed to “translate” URDF files into TF functions (tf2 package).
- ROS has different packages to manage URDF files.
- To transform the URDF files to frames and transforms, ROS uses a package called *“robot\_state\_publisher”*.





- Once the transforms gets published, it is available to all components in the system that also use tf2.
- The package takes the joint and links of the robot as input and publishes a transform tree model of the robot.





# Robot State Publisher



- 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.

```
urdf_file_name = 'continuos_ex.urdf'
urdf = os.path.join(
    get_package_share_directory('joints_act'),
    'urdf',
    urdf_file_name)

with open(urdf, 'r') as infp:
    robot_desc = infp.read()

robot_state_pub_node = Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    name='robot_state_publisher',
    output='screen',
    parameters=[{'robot_description': robot_desc}],
    arguments=[urdf]
)
```

URDF

@Yd

{Learn, Create, Innovate};

Property of MCR<sup>2</sup>



Manchester **Robotics**

## Joints

Joints represents the kinematic connection between two links.

- Joints are defined within `<robot>` using the `<joint>` element. Each joint element have attributes like name, type, and contain child elements to define properties like limits, axis, and origin.

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

<robot name="multiple_joints_example">

  <link name="base_footprint">
  </link>

  <link name="base_link">
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <mesh filename="file:///mesh"/>
      </geometry>
    </visual>
    <collision>
      <geometry>
        <box size="0 0 0"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="1"/>
      <xacro:box_inertia mass="1" x="0" y="0" z="0" />
    </inertial>
  </link>

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

</robot>
```

Joint

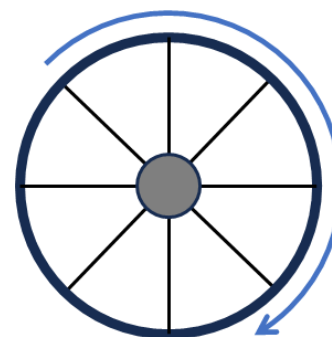
- **Revolute** - A rotational motion, on a single axis, with minimum/maximum angle limits.
- **Continuous** - A rotational motion, on a single axis, with no limit (e.g. a wheel).
- **Prismatic** - A linear sliding motion, on a single axis, with minimum/maximum position limits.
- **Fixed** - The child 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.



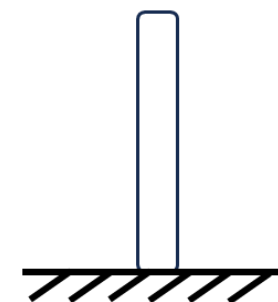
Revolute



Prismatic



Continuous



Fixed



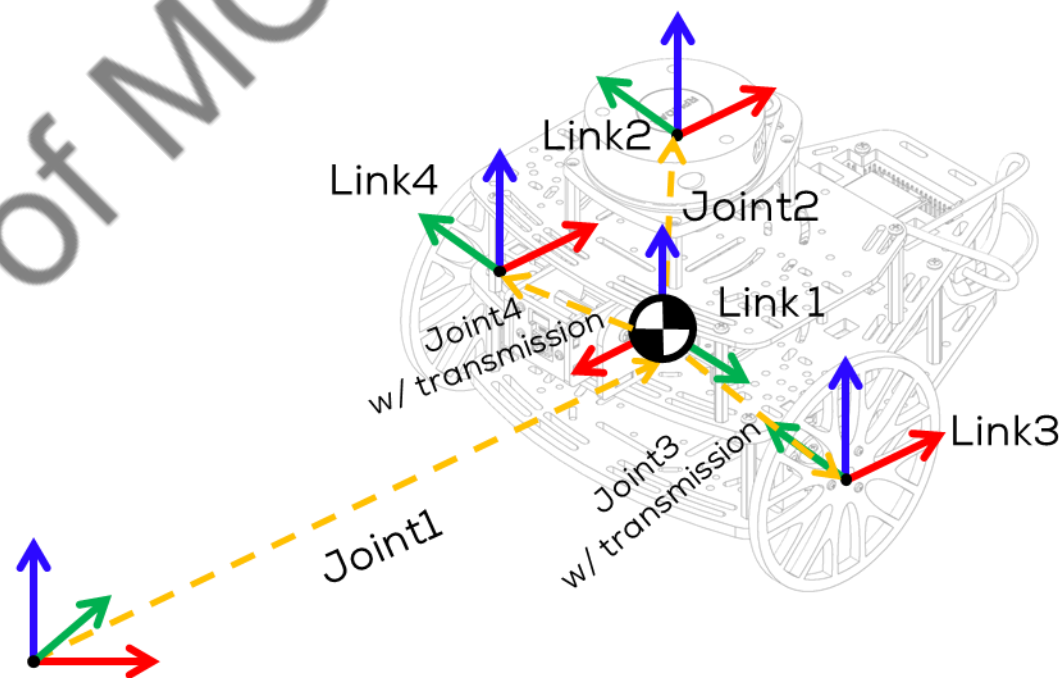
# Syntax



```
<!--Declare Joints to be used-->
<!--Declare Joint element 'name' and 'type' (revolute, continuos, prismatic, fixed)-->
<joint name="joint1" type="revolute">
  <parent link="link1"/>          <!--parent: Parent link name -->
  <child link="link2"/>          <!--child: Child link name -->
  <!--origin: This is the transform from the parent link to the child link xyz: xyz offset rpy:rotation offset (radians)-->
  <origin xyz="0.5 1 0.5" rpy="0 0 0" />
  <!--Rotation/Translation axis for the joints(revolute,continuos/prismatic) noy used with Fixed Joints -->
  <axis xyz="0 0 1 " />
  <!--Establish the limits of a joint Lower/Upper: lower/upper limits
  for revolute/prismatic joints in rad or m, Effort: maximum joint force (torque or force)
  depend on the joint, Velocity: Max velocity of the joint (rad/s or m/s) -->
  <limit lower="-1.57" upper="1.57" effort="5.0" velocity="0.1"/>
</joint>
```

## Extras

- **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.



# Activity 1

*Creating a simple URDF  
file*

*{Learn, Create, Innovate};*

Prope

MCR<sup>2</sup>

Manchester **Robotics**





# Activity 1



1. Make a new package called "joints\_act" with the following library packages

geometry\_msgs, rclpy, sensor\_msgs, std\_msgs, tf2\_ros\_py, ros2launch, robot\_state\_publisher, joint\_state\_publisher

```
$ ros2 pkg create --build-type ament_python joints_act --  
license Apache-2.0 --node-name joint_pub --dependencies  
sensor_msgs geometry_msgs python3-numpy rclpy tf2_ros_py  
ros2launch std_msgs robot_state_publisher  
joint_state_publisher --description URDF Examples --  
maintainer-name 'Mario Martinez' --maintainer-email  
mario.mtz@manchester-robotics.com
```

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 inside the package

```
mkdir launch && touch launch/fixed_launch.py
```

```
<?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



- Open the “setup.py” file
- Add the libraries to be used

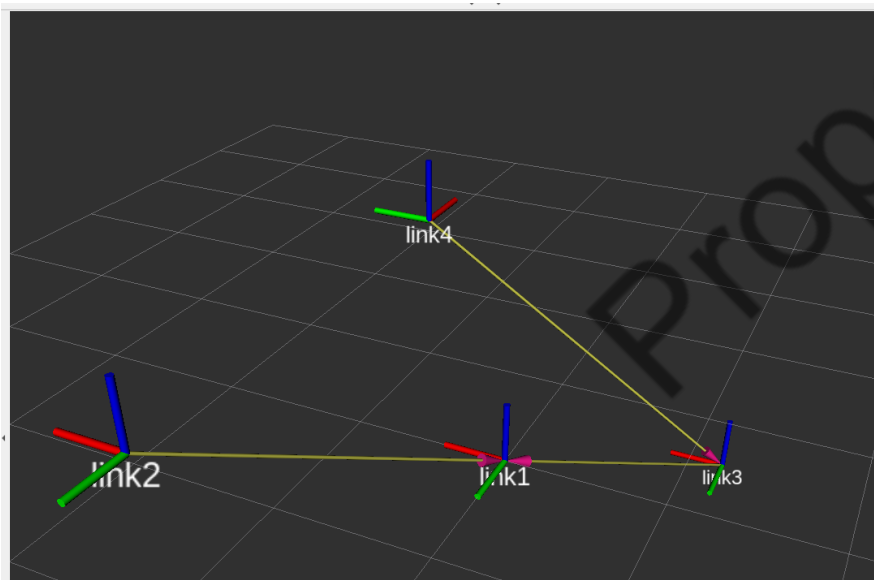
```
from setuptools import find_packages, setup
import os
from glob import glob
```

- Configure the data\_files section as follows

```
data_files=[
    ('share/ament_index/resource_index/packages',
     ['resource/' + package_name]),
    ('share/' + package_name, ['package.xml']),
    (os.path.join('share', package_name, 'launch'), glob(os.path.join('launch', '*launch.[pxy][yma]*'))),
    (os.path.join('share', package_name, 'config'), glob(os.path.join('config', '*.[yma]*'))),
    (os.path.join('share', package_name, 'rviz'), glob(os.path.join('rviz', '*.rviz'))),
    (os.path.join('share', package_name, 'meshes'), glob(os.path.join('meshes', '*.stl'))),
    (os.path.join('share', package_name, 'urdf'), glob(os.path.join('urdf', '*.urdf'))),
],
```

# Activity 1

5. Paste the following code inside the launch file, compile and launch (give execution permissions)
6. Open RVIZ
7. Make the fixed frame to be "link 1"
8. Add the marker
  - Press "Add"
  - >>By display type>>TF



```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    urdf_file_name = 'fixed_ex.urdf'
    urdf = os.path.join(
        get_package_share_directory('joints_act'),
        'urdf',
        urdf_file_name)

    with open(urdf, 'r') as infp:
        robot_desc = infp.read()

    robot_state_pub_node = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        parameters=[{'robot_description': robot_desc}],
        arguments=[urdf]
    )

    l_d = LaunchDescription([robot_state_pub_node])

    return l_d
```

# Activity 2

*Creating a movable joint  
using URDF*

*{Learn, Create, Innovate};*

MCR<sup>2</sup>

Manchester **Robotics**

# Activity 2

1. Create a new “urdf” named “continuous\_ex.urdf” inside the previously created package “joints\_act”

```
$ touch urdf/continuous_ex.urdf
```

2. Paste the following code inside the “continuous\_ex.urdf” file.
  - Code can be found on GitHub.
3. Create a launch file for this node (next slide)

```
$ touch launch/continuous_launch.py
```

5. Add the TF view
  - Press Add
  - >>By display type>>TF
6. Run the tf\_tree...

```
$ ros2 run 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"?>

<!--Start a Robot description-->
<robot name="continuous_joint_example">

  <!--Declare Links to be used-->
  <link name="link1" />
  <link name="link2" />

  <!--Declare Joints to be used-->
  <joint name="joint1" type="continuous">
    <parent link="link1"/>
    <child link="link2"/>
    <origin xyz="0.5 1 0.5" rpy="0 0 0" />
    <axis xyz="0 0 1" />
  </joint>
</robot>
```



# Activity 2



```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():

    urdf_file_name = 'continuos_ex.urdf'
    urdf = os.path.join(
        get_package_share_directory('joints_act'),
        'urdf',
        urdf_file_name)

    with open(urdf, 'r') as infp:
        robot_desc = infp.read()

    robot_state_pub_node = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        parameters=[{'robot_description': robot_desc}],
        arguments=[urdf]
    )

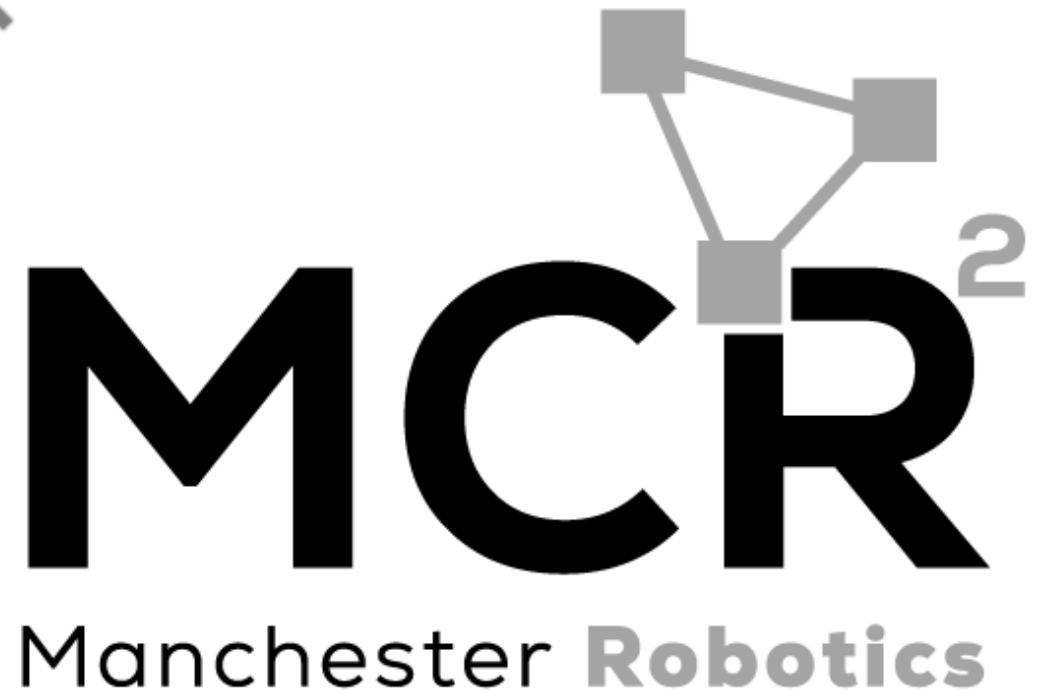
    l_d = LaunchDescription([robot_state_pub_node])
    return l_d
```

# Robot Descriptions

@\_Ydl dQdUFer\YXUb

{Learn, Create, Innovate};

Property of MCR<sup>2</sup>





# Joint State Publisher



## 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_states` 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/)
```

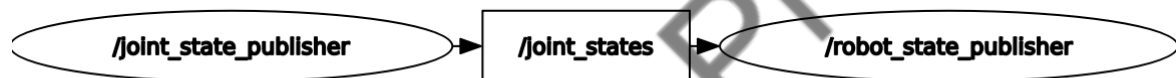




# Joint State Publisher



- 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: []
```

8. Install the Joint State Publisher

```
$ sudo apt install ros-<ROS_DISTRO>-joint-state-publisher-gui
#change <ROS_DISTRO> with your ros distribution e.g., "humble"
```

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

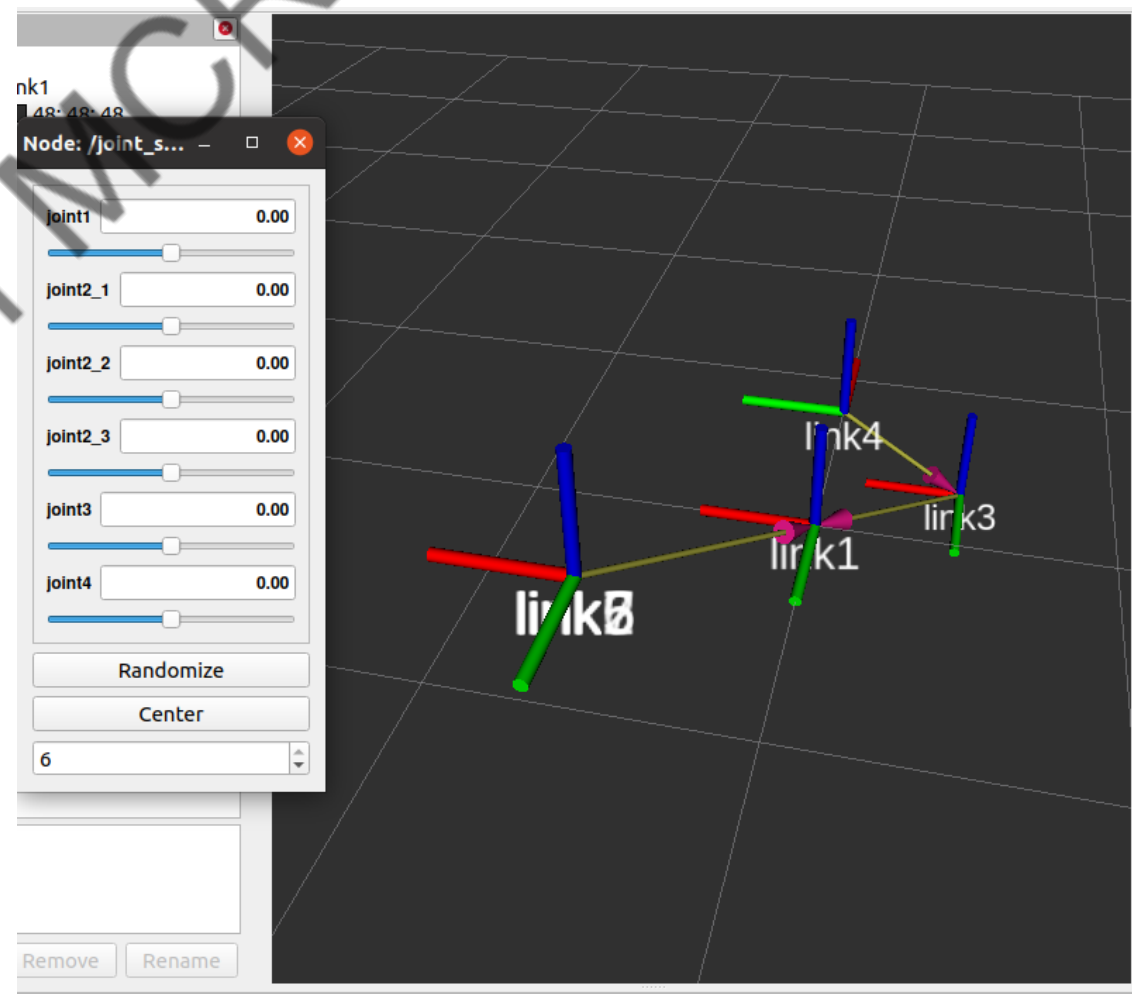
```
# Define joint_state_publisher node (for simulation)
joint_state_publisher_node = Node(
    package='joint_state_publisher_gui',
    executable='joint_state_publisher_gui',
    output='screen'
)

l_d = LaunchDescription([robot_state_pub_node,
    joint_state_publisher_node])
```

10. Compile and relaunch the activity.

11. Use the GUI to move the joints!

12. Add More continuous joints to this activity!





# Activity 2: Extension



1. Create a new “urdf” named “joints\_ex.urdf” inside the previously created package “joints\_act”

```
$ touch urdf/joints_ex.urdf
```

2. Define all the different joints that are available

- Code can be found on GitHub.

3. Create a launch file for this node (similar to the previous one), change the URDF File name at the top.

```
$ touch launch/joints_launch.py
```

4. Add the joint\_state publisher to the launch file

```
$ ros2 run rviz2 rviz2
```

5. Add the TF view

- Press Add
- >>By display type>>TF

6. Run the tf\_tree...

```
<?xml version="1.0"?>
<robot name="multiple_joints_example">

  <!--Declare Links to be used-->
  <link name="link1" />
  <link name="link2" />
  <link name="link3" />
  <link name="link4" />
  <link name="link5" />
  <link name="link6" />

  <!--Declare Joints to be used-->
  <joint name="joint_fixed" type="fixed">
    <parent link="link1"/>
    <child link="link2"/>
    <origin xyz="1 1 1" rpy="0 0 0" />
  </joint>

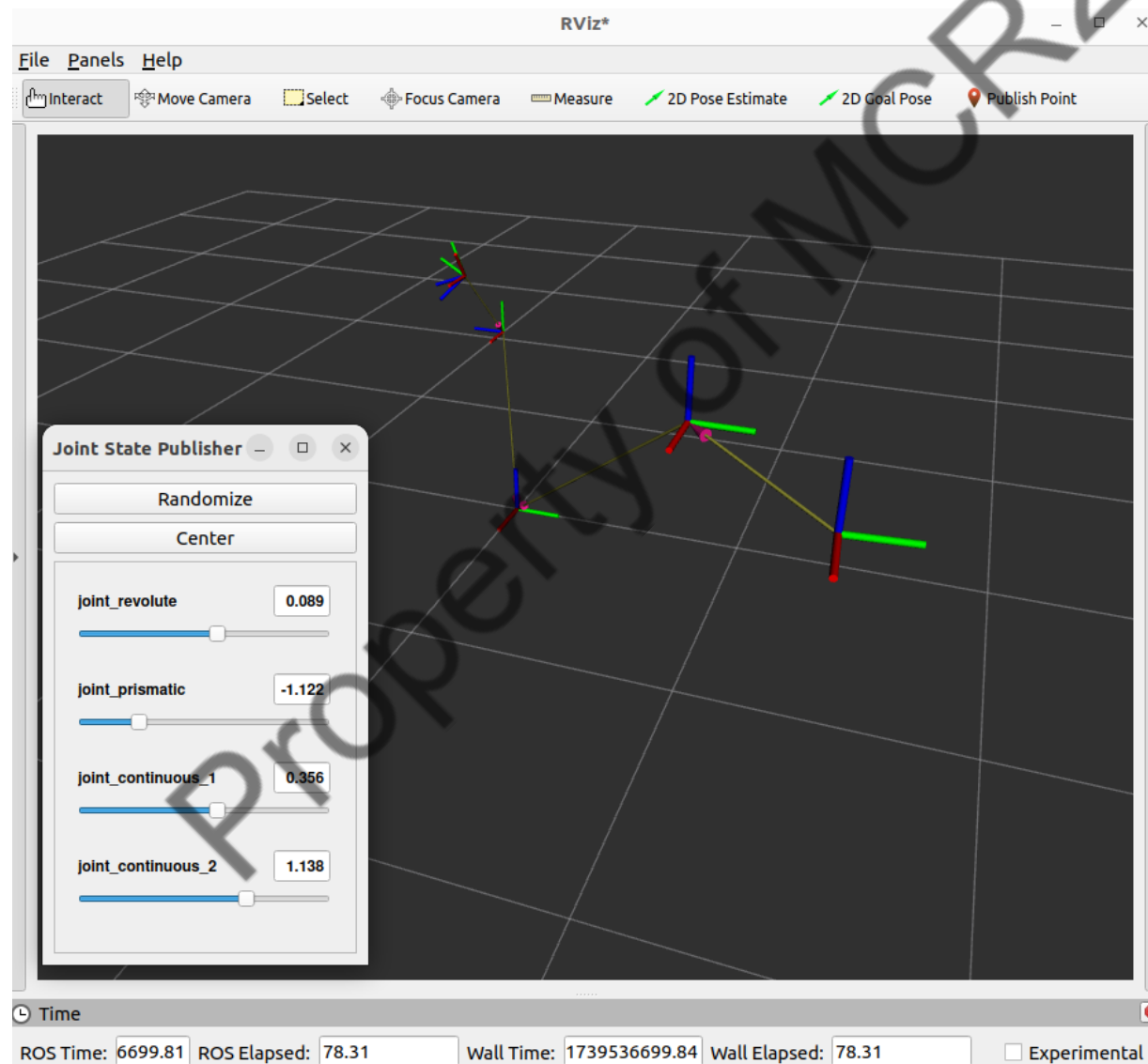
  <joint name="joint_revolute" type="revolute">
    <parent link="link2"/>
    <child link="link3"/>
    <origin xyz="0.5 0.5 0" rpy="0 0 0" />
    <axis xyz="0 0 1" />
    <limit lower="-0.785" upper="0.785" effort="5.0" velocity="0.1"/>
  </joint>

  ... #Define the rest of the joints here ....

</robot>
```



# Activity 2: Extension



# Joint control

*Making your own Joint  
State Publisher*

*{Learn, Create, Innovate};*

Property of MCR<sup>2</sup>



Manchester **Robotics**



# Joint state publisher



- As stated previously, the robot state publisher, will not publish the joints unless something is published into the /joint\_states topic.
- The joint state publisher does this by publishing positional values of the joints to the /joint\_states topic using a JointState message. This allows the user to test the joints before releasing the model.
- The JointState message holds the data that describes the state of a torque-controlled joint.
- Each joint is uniquely identified by its name
- ROS associate the joint name with the correct states.
- The state of each joint (continuous, revolute or prismatic) is defined by:
  - the position of the joint (rad or m),
  - the velocity of the joint (rad/s or m/s)
  - the effort applied in the joint (Nm or N).
- The header specifies the time and frame at which the joint states were recorded.
- This message consists of a multiple arrays, one for each part of the joint state.
- All arrays in this message should have the same size, or be empty.

Header header

```
string[] name
float64[] position
float64[] velocity
float64[] effort
```

# Activity 3

*Making your own joint  
state publisher*

*{Learn, Create, Innovate};*

MCR<sup>2</sup>

Manchester **Robotics**



# Activity 3



## Description

- In this activity, the user will create a node to control the position of the joints declared in the file "joints\_ex.urdf".
- The position will be controlled by publishing a JointState message into the /joint\_states topic.

```
# EXAMPLE
from sensor_msgs.msg import JointState

# Declare the output Messages
contJoints = JointState()

# Declare the output Messages
self.ctrlJoints = JointState()
    self.ctrlJoints.header.stamp = self.get_clock().now().to_msg()
    self.ctrlJoints.name = ["joint_revolute", "joint_prismatic"]
    self.ctrlJoints.position = [0.0] * 2
    self.ctrlJoints.velocity = [0.0] * 2
    self.ctrlJoints.effort = [0.0] * 2
```

- For this activity, the URDF model "joints\_ex.urdf" will be used. This file can be found in the course repository. (Previous activity)
- The URDF contains 4 joints with the following names  
["joint\_revolute", "joint\_prismatic", "joint\_continuous\_1", "joint\_continuous\_2"]
- Please be aware that the JointMessage contains arrays, and they must be the same size as the number of joints to be controlled.





# Partial Code



```
class FramePublisher(Node):

    def __init__(self):
        super().__init__('joints_publisher')

        #Publisher
        self.publisher = self.create_publisher(JointState,
        '/joint_states', 10)

        #Create a Timer
        timer_period = 0.1 #seconds
        self.timer = self.create_timer(timer_period, self.timer_cb)

        #initialise Message to be published
        self.ctrlJoints = JointState()
        self.ctrlJoints.header.stamp = self.get_clock().now().to_msg()
        self.ctrlJoints.name = ["joint_revolute", "joint_prismatic",
        "joint_continuous_1", "joint_continuous_2"]
        self.ctrlJoints.position = [0.0] * 4
        self.ctrlJoints.velocity = [0.0] * 4
        self.ctrlJoints.effort = [0.0] * 4

        ... #Variables
```

```
#Timer Callback
def timer_cb(self):
    time = self.get_clock().now().nanoseconds/1e9

    self.ctrlJoints.header.stamp = self.get_clock().now().to_msg()
    self.ctrlJoints.position[0] = -0.785 + 2*0.785*self.i
    self.ctrlJoints.position[1] = -2+4.0*self.i
    self.ctrlJoints.position[2] = 0.5*time
    self.ctrlJoints.position[3] = 0.5*time

    ... # Rest of the code

    self.publisher.publish(self.ctrlJoints)
```



# Activity 3



1. Create a node named "continuousJoint.py" inside the previously created package "joints\_act"

```
touch joints_act/joint_pub.py
```

2. Give execution permission and add it to the setup.py (if haven't done so)
3. Copy the code in GitHub for this node.
4. Make a new launch called "jointPub\_launch.py" file similar to the one done for the previous activity (copy it) replace the name of the URDF file at the top.

```
touch launch/jointPub_launch.py
```

5. Give execution permissions

6. Replace the joint state publisher with your state publisher

```
# Define joint_state_publisher node (for simulation)
joint_state_publisher_node = Node(
    package='joints_act',
    executable='joint_pub',
    output='screen'
)
```

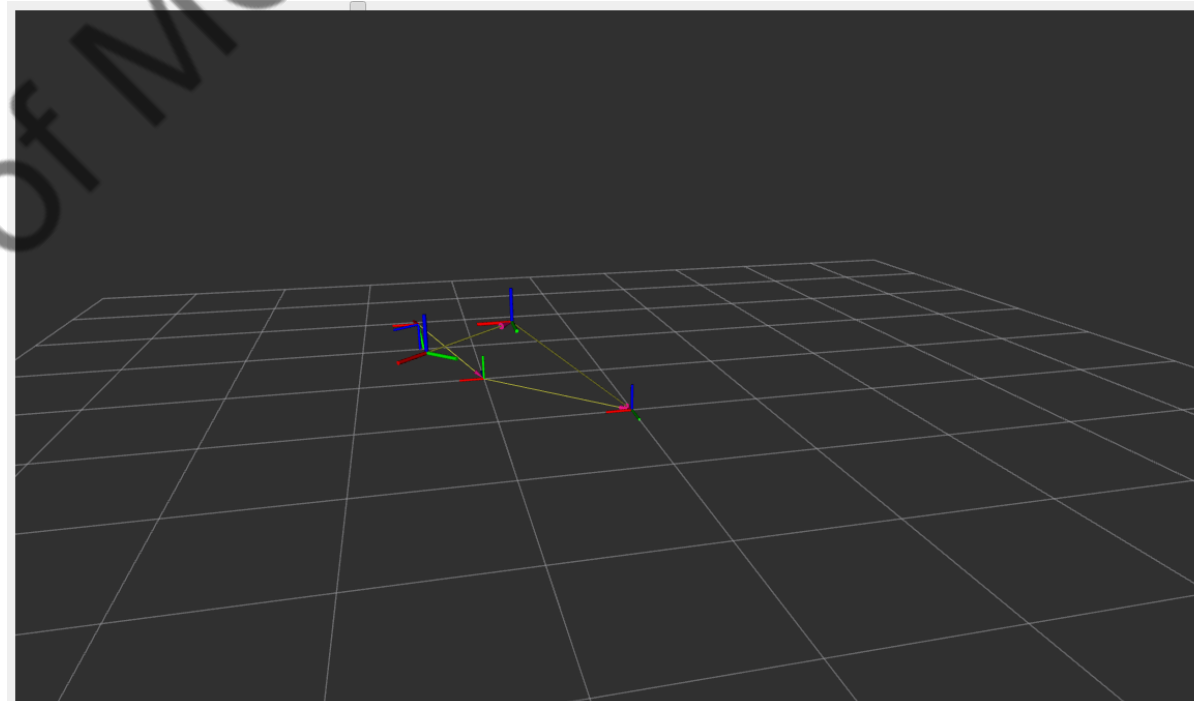
7. Compile the node.
8. Launch the node
9. Add the TF view
  - Press Add
  - >>By display type>>TF
10. Run the tf\_tree...



# Activity 3



- Inside the timer of the node, the positional values for each joint (4 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.
- The publishing of the message is done as a regular ROS message.



# URDF

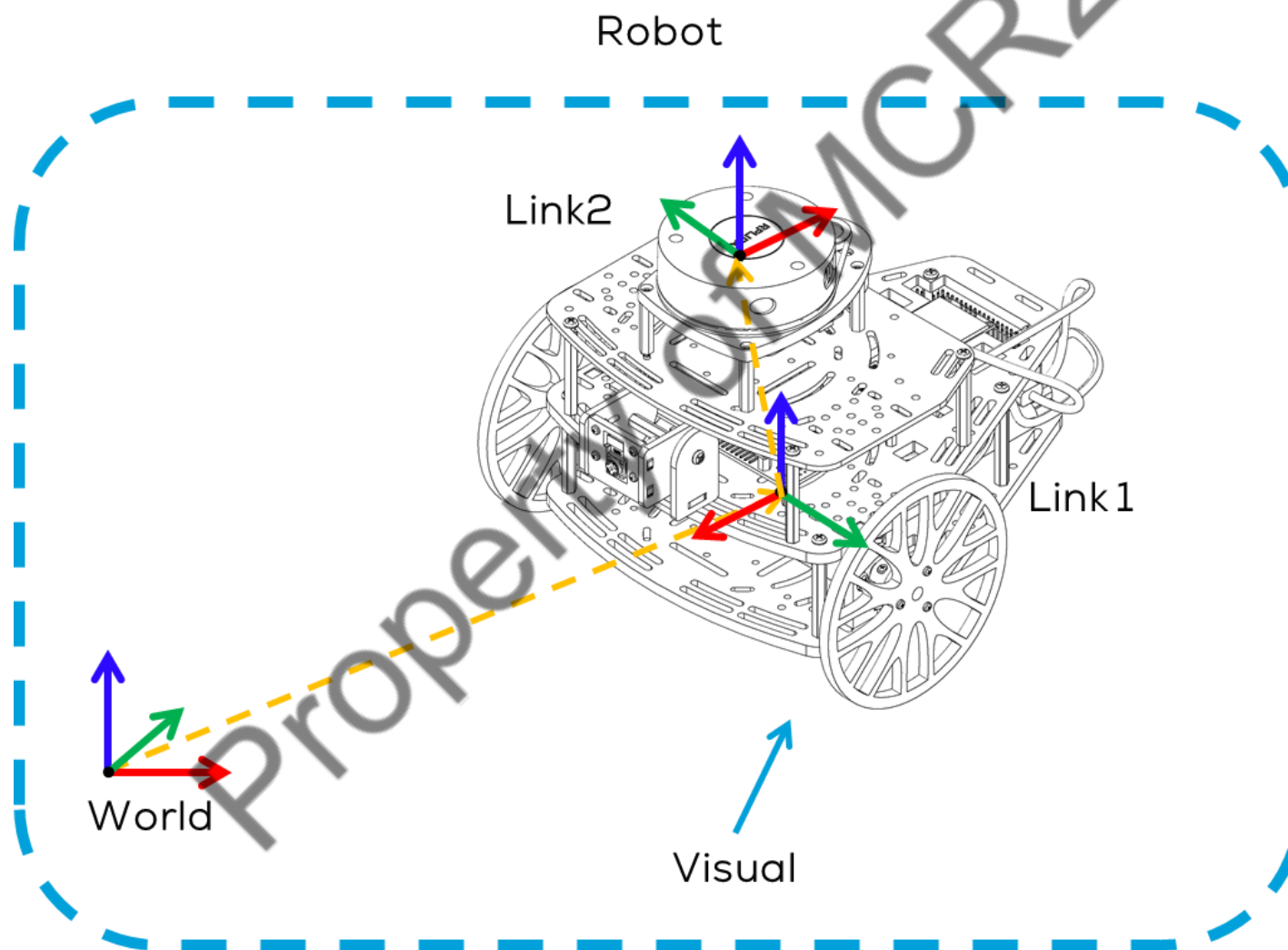
*Links*

*{Learn, Create, Innovate};*

Property of MCR<sup>2</sup>

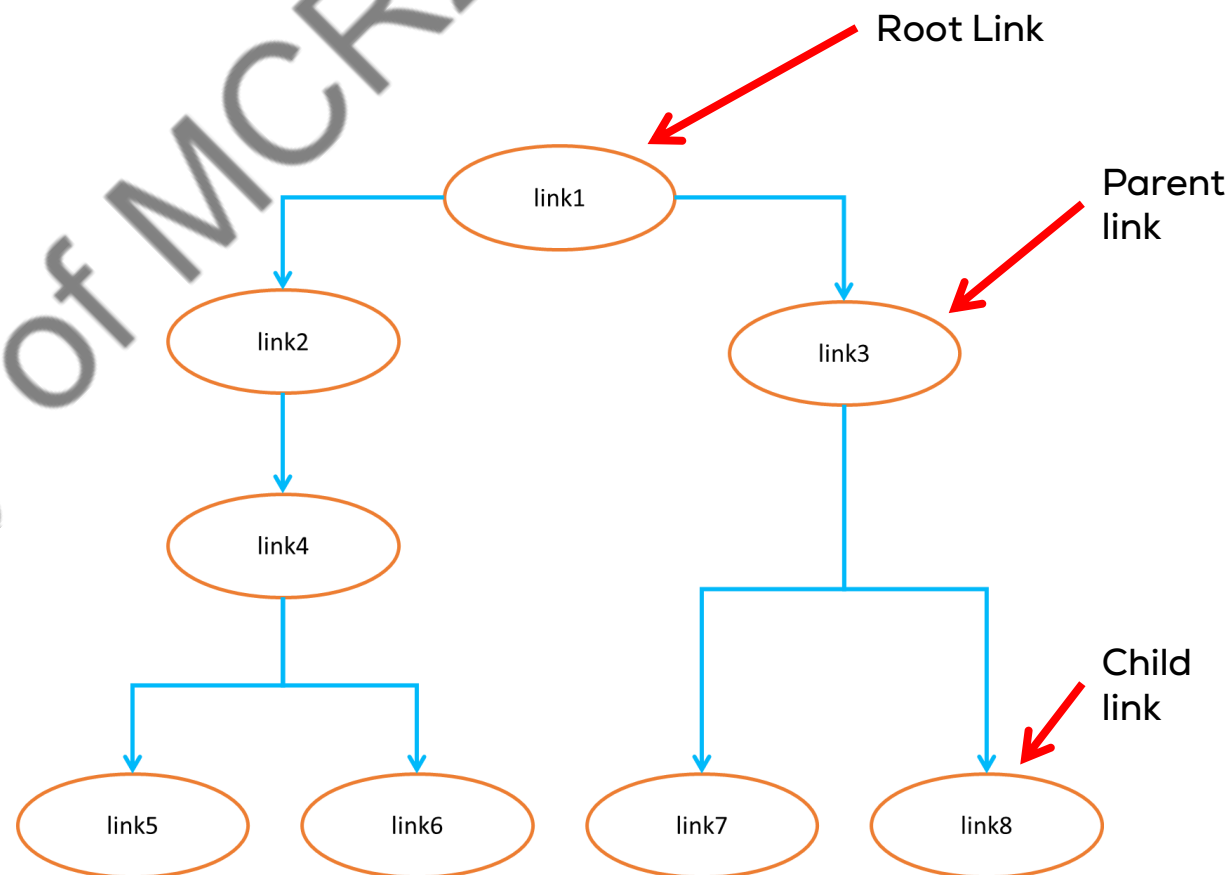


Manchester **Robotics**



## Links description

- As stated previously, links represent a rigid body or a robot component.
- Links describe the visual, inertial, and collision properties of the link.
- Each link may have one or more visual and collision elements associated with it.
- Each link cannot have more than one parent (defined in joints).
- Root Link can have no parents.
- Only one root link is allowed per model.
- Root Link is usually modified using a joint state publisher.



## Visual

- Defines the visual appearance of a link.
- The inbuilt shapes are cylinder, box, and sphere.
- The visual can also set a mesh from a CAD file in format .dae or .stl.
- Origin: Sets the pose of the visual part relative to the link.
- Material: material properties (e.g., color, texture)

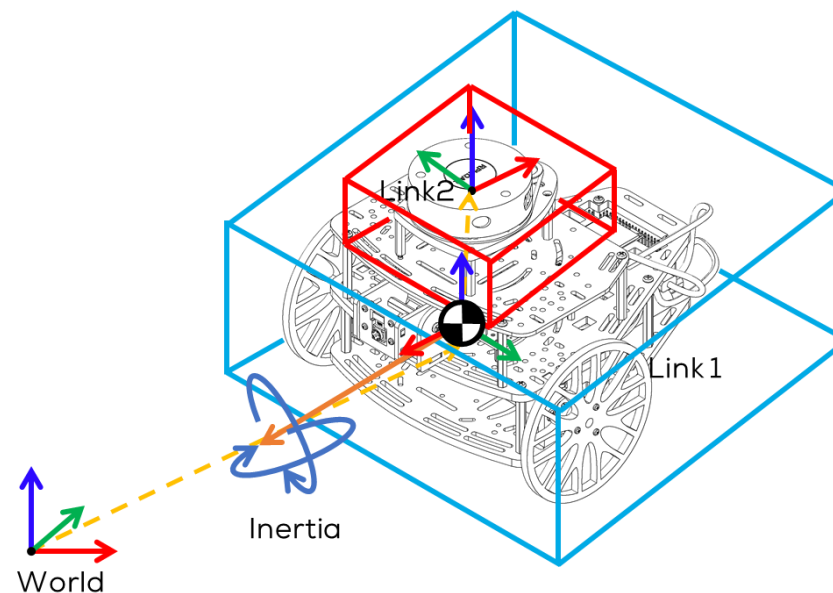
```
<link name="wheel_left_link">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <mesh filename="file://mesh.stl"/>
    </geometry>
    <material name="yellow"/>
  </visual>
</link>
```

```
<link name="link1">
  <visual>
    <origin xyz="0 0 0" rpy="0 1.57 0"/>
    <geometry>
      <cylinder length="0.4" radius="0.1" />
    </geometry>
    <material name="Grey">
      <color rgba="0.67 0.67 .67 1.0"/>
    </material>
  </visual>
</link>
```

## Collision

- Defines the collision properties of a link.
- Collision properties can be different from the visual properties of a link (simpler collision models).
- Multiple <collision> tags can exist for the same link.
- The union of the geometry they define forms the collision representation of the link.
- Origin: Sets the pose of the visual part relative to the link frame.
- Geometry: Geometrical description of the collision body  
The inbuilt shapes are cylinder, box, and sphere.

```
<link name="wheel_left_link">
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder length="1" radius="1" />
    </geometry>
  </collision>
</link>
```

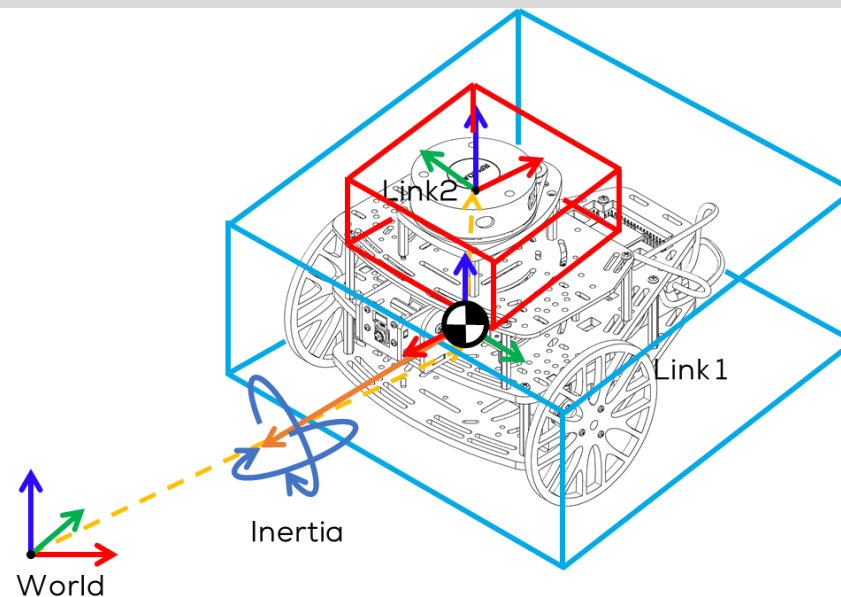




## Inertial

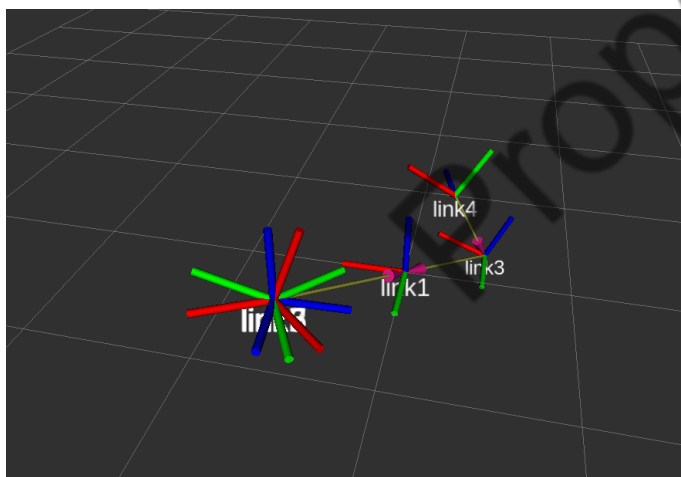
- Defines the link's mass, the position of its centre of mass, and its central inertia properties.
- Origin: Position and orientation of the link's centre of mass relative to the link frame "L".
- Mass: The mass of the link.
- Inertia: This link's moments of inertia  $ixx$ ,  $iyy$ ,  $izz$  and products of inertia  $ixy$ ,  $ixz$ ,  $iyz$  about C (centre of mass) for the unit vectors  $\hat{C}_x$ ,  $\hat{C}_y$ ,  $\hat{C}_z$  fixed in the centre-of-mass frame C. Relative to  $\hat{L}_x$ ,  $\hat{L}_y$ ,  $\hat{L}_z$  is specified by the rpy values in the <origin> tag.

```
<link name="wheel_left_link">
  <inertial>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.4"
      iyz="0.0" izz="0.2"/>
  </inertial>
</link>
```



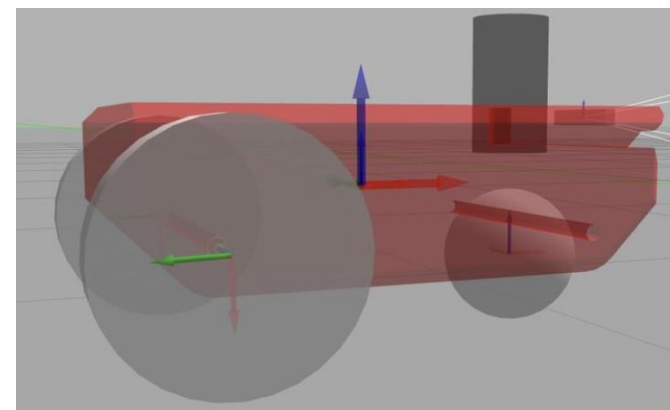
## RVIZ

- In RVIZ Links are represented by sets of coordinates.
- RVIZ only requires a name for a link.
- RVIZ only uses the information in the collision and the visual descriptions of the URDF file (both optional).



## Gazebo

- The Gazebo simulation environment, requires a plugin to transform the URDF files into SDF files (Gazebo's native file type).
- Gazebo requires information about the visual, inertial, and collision properties, alongside other physics parameters for each link, such as friction, stiffness, and damping.



# Links: Syntax

```
<link name="wheel_left_link">

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <mesh filename="file://mesh.stl"/>
    </geometry>
    <material name="yellow"/>
  </visual>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder length="1" radius="1" />
    </geometry>
  </collision>

  <inertial>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.4" iyz="0.0" izz="0.2"/>
  </inertial>

</link>
```

# Activities

*Activity 4 and Activity 5*

*{Learn, Create, Innovate};*

MCR<sup>2</sup>

Manchester **Robotics**



# Activity 4



1. For the following activities, make a new package called “link\_act” with the following library

```
$ ros2 pkg create --build-type ament_python link_act --node-name puzzle_drone --license Apache-2.0 --dependencies sensor_msgs geometry_msgs robot_state_publisher joint_state_publisher_gui std_msgs ros2launch tf2_ros_py python3-numpy python3-transforms3d rclpy --description 'URDF Link Examples' --maintainer-name 'Mario Martinez' --maintainer-email mario.mtz@manchester-robotics.com
```

- Create the following folders inside the package
  - launch, meshes, urdf
- Create 2 launch files named inside the “launch” folder (do not forget to give executable permissions):
  - cylinder\_launch.py
  - puzzledrone\_launch.py
- Download from GitHub and add the three .stl files to the meshes folder
  - base\_210mm.stl
  - propeller\_ccw\_puller\_5in.stl
  - propeller\_cw\_puller\_5in.stl
- Create two URDF Files inside the “urdf” folder
  - cylinder.urdf
  - puzzle\_drone.urdf

```
src/link_act/  
├── launch  
│   ├── cylinder_launch.py  
│   └── puzzledrone_launch.py  
├── LICENSE  
├── link_act  
│   ├── __init__.py  
│   └── puzzle_drone.py  
├── meshes  
│   ├── base_210mm.stl  
│   ├── propeller_ccw_puller_5in.stl  
│   └── propeller_cw_puller_5in.stl  
├── package.xml  
├── resource  
│   └── link_act  
├── rviz  
│   └── puzzledrone.rviz  
├── setup.cfg  
├── setup.py  
├── test  
│   ├── test_copyright.py  
│   ├── test_flake8.py  
│   └── test_pep257.py  
└── urdf  
    ├── cylinder.urdf  
    └── puzzle_drone.urdf
```



# Activity 1



- Open the “setup.py” file
- Add the libraries to be used

```
from setuptools import find_packages, setup
import os
from glob import glob
```

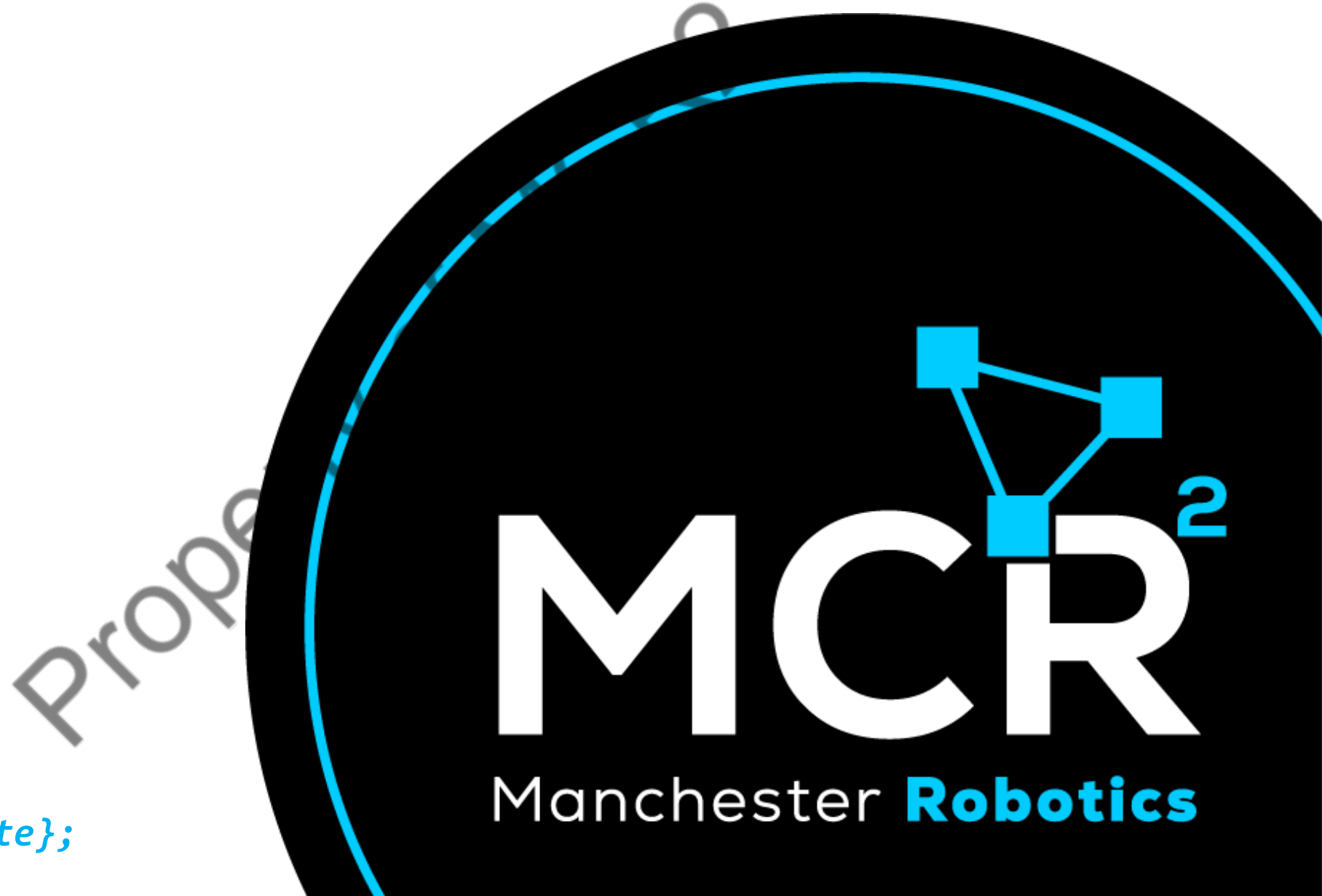
- Configure the data\_files section as follows

```
data_files=[
    ('share/ament_index/resource_index/packages',
     ['resource/' + package_name]),
    ('share/' + package_name, ['package.xml']),
    (os.path.join('share', package_name, 'launch'), glob(os.path.join('launch', '*launch.[pxy][yma]*'))),
    (os.path.join('share', package_name, 'config'), glob(os.path.join('config', '*.[yma]*'))),
    (os.path.join('share', package_name, 'rviz'), glob(os.path.join('rviz', '*.rviz'))),
    (os.path.join('share', package_name, 'meshes'), glob(os.path.join('meshes', '*.stl'))),
    (os.path.join('share', package_name, 'urdf'), glob(os.path.join('urdf', '*.urdf'))),
],
```

8 Self Yd o

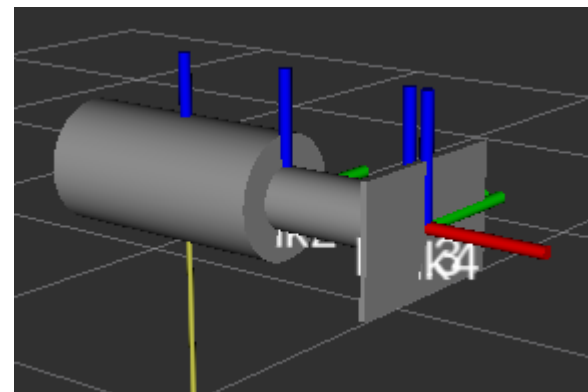
*Links in RVIZ*

*{Learn, Create, Innovate};*



## Description

- In this activity, the user will create a link and a joint state publisher that depicts a pneumatic cylinder moving continuously.
- The position will be controlled by publishing a JointState message into the /joint\_states topic.
- Open the urdf file (inside the urdf folder) named "cylinder.urdf"
- Create 5 links, and 4 joints with the information depicted in the following figure.





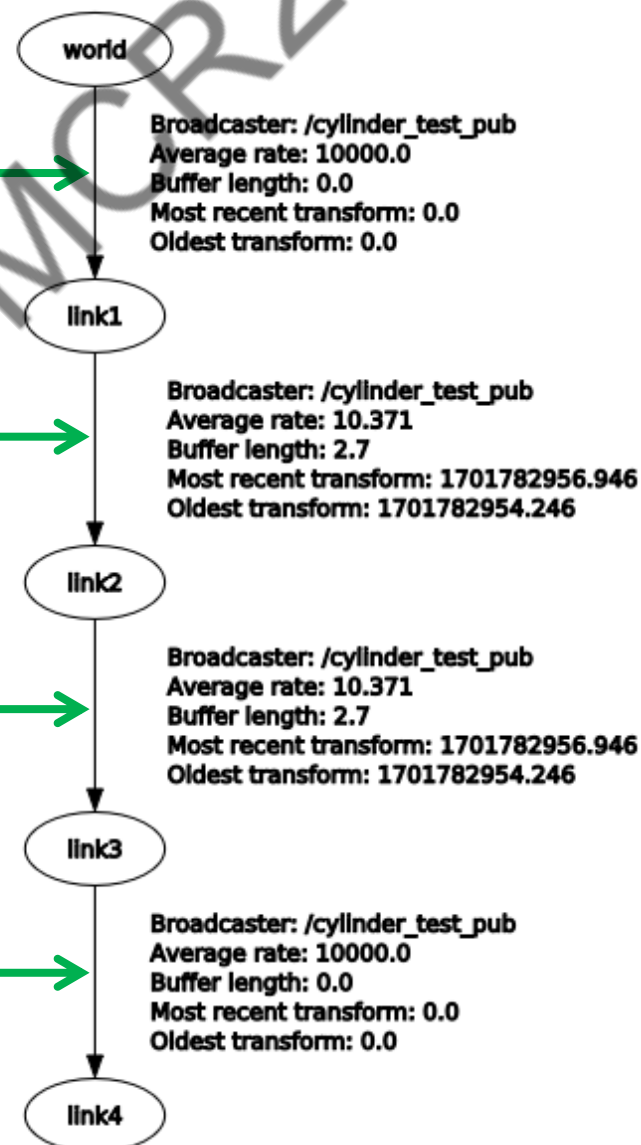
# Activity 4

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

```
<joint name="joint2" type="prismatic">
  <parent link="link1"/>
  <child link="link2"/>
  <origin xyz="0.1 0 0" rpy="0 0 0" />
  <axis xyz="1 0 0" />
  <limit lower="0.0" upper="0.2" effort="5.0" velocity="0.1"/>
</joint>
```

```
<joint name="joint3" type="revolute">
  <origin xyz="0.225 0 0" rpy="0 0 0" />
  <parent link="link2" />
  <child link="link3" />
  <axis xyz="0 1 0" />
  <limit lower="-0.785" upper="0.785" effort="5" velocity="2" />
</joint>
```

```
<joint name="joint4" type="fixed">
  <origin xyz="0.03 0 0" rpy="0 0 0" />
  <parent link="link3" />
  <child link="link4" />
</joint>
```



# Activity 4

- For each link (Link1 – Link 4) use the following description in the URDF file.
- The link “world” must be empty.

```
<link name="world">
</link>

<link name="link1">
  <visual>
    <origin xyz="0 0 0" rpy="0 1.57 0"/>
    <geometry>
      <cylinder length="0.4" radius="0.1" />
    </geometry>
    <material name="Grey">
      <color rgba="0.67 0.67 .67 1.0"/>
    </material>
  </visual>
</link>

<link name="link2">
  <visual>
    <origin xyz="0 0 0" rpy="0 1.57 0"/>
    <geometry>
      <cylinder length="0.4" radius="0.05" />
    </geometry>
    <material name="Grey">
      <color rgba="0.67 0.67 .67 1.0"/>
    </material>
  </visual>
</link>
```

```
<link name="link3">
  <visual>
    <origin xyz="0 0 0" rpy="0 1.57 1.57"/>
    <geometry>
      <cylinder length="0.1" radius="0.025" />
    </geometry>
    <material name="Grey">
      <color rgba="0.67 0.67 .67 1.0"/>
    </material>
  </visual>
</link>

<link name="link4">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="0.01 0.3 0.2" />
    </geometry>
    <material name="Grey">
      <color rgba="0.67 0.67 .67 1.0"/>
    </material>
  </visual>
</link>
```



# Activity 4



## Testing the model

- Open the “cylinder\_launch.py” and add the following.
- Save it and give executable permissions if you haven’t done so.
- Make a lunch file and use the “robot\_state\_publisher” to publish the information about the state of the cylinder.
- Use the Joint State Controller GUI in the launch file to verify if the model is working properly.
- To visualise the model in RVIZ select “world” as Fixed Frame.
- In RVIZ press “Add>>By display type>>RobotModel”. In the RobotModel select s “Description Topic” “/robot\_description”.

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():

    urdf_file_name = 'cylinder.urdf'
    urdf = os.path.join(
        get_package_share_directory('link_act'),
        'urdf',
        urdf_file_name)

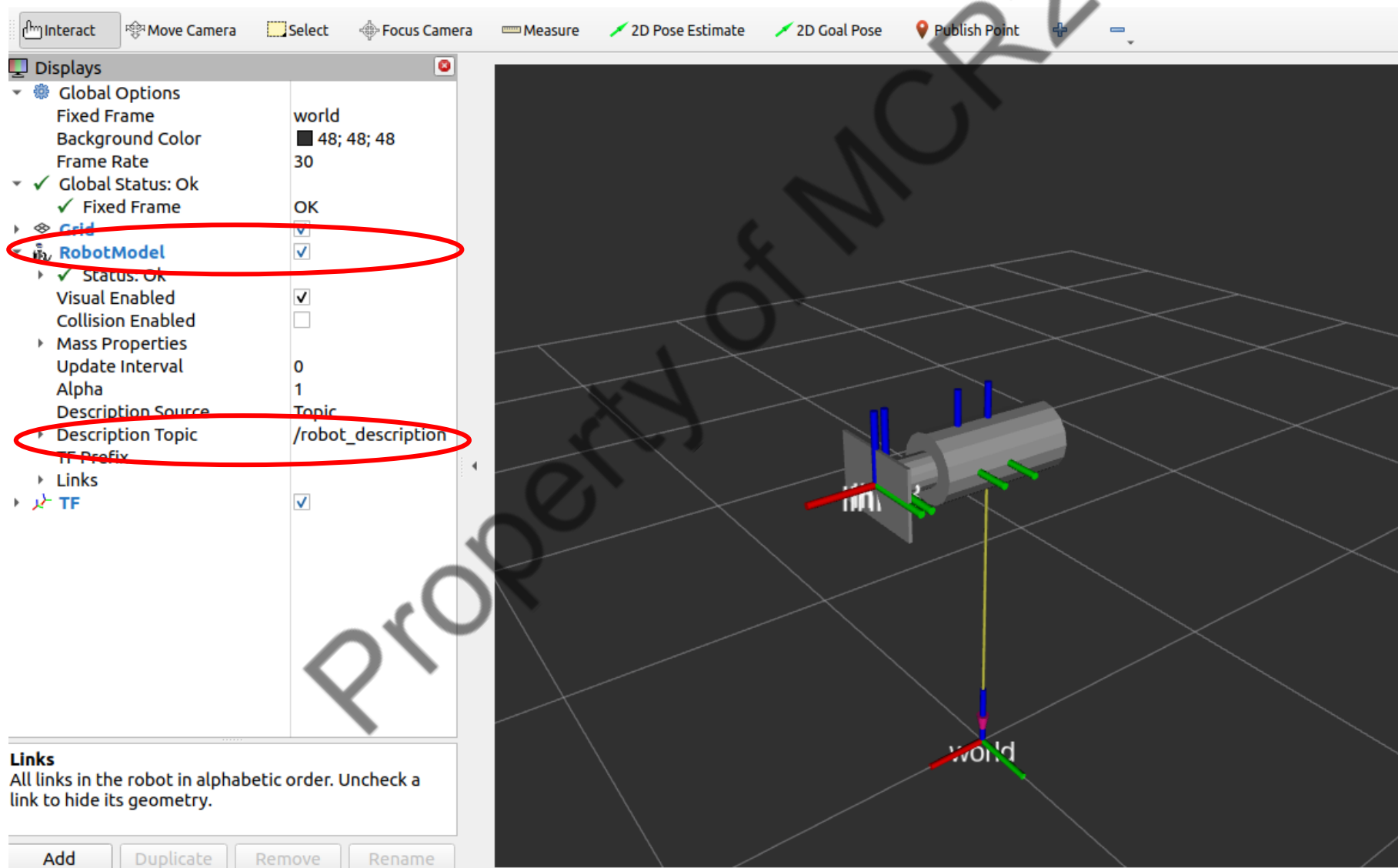
    with open(urdf, 'r') as infp:
        robot_desc = infp.read()

    robot_state_pub_node = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='screen',
        parameters=[{'robot_description': robot_desc}],
        arguments=[urdf]
    )

    # Define joint_state_publisher node (for simulation)
    joint_state_publisher_node = Node(
        package='joint_state_publisher_gui',
        executable='joint_state_publisher_gui',
        output='screen'
    )

    l_d = LaunchDescription([robot_state_pub_node, joint_state_publisher_node])
    return l_d
```

# Activity 4



The screenshot shows the ROS2 GUI interface. The top toolbar includes buttons for Interact, Move Camera, Select, Focus Camera, Measure, 2D Pose Estimate, 2D Goal Pose, and Publish Point. The left sidebar contains a 'Displays' panel with the following settings:

- Global Options**
  - Fixed Frame: world
  - Background Color: 48; 48; 48
  - Frame Rate: 30
- Global Status: Ok**
  - Fixed Frame: OK
- Grid**
  - ☒ Grid
- RobotModel** (highlighted with a red circle)
  - ☒ RobotModel
  - Status: Ok**
    - Visual Enabled: ☒
    - Collision Enabled: ☐
  - Mass Properties**
    - Update Interval: 0
    - Alpha: 1
  - Description Source**
    - Description Source: Topic
    - Description Topic: /robot\_description (highlighted with a red circle)
    - TF Prefix: ☐
  - Links**
    - ☒ TF

At the bottom of the sidebar, there is a 'Links' section with the text: "All links in the robot in alphabetic order. Uncheck a link to hide its geometry." Below this are buttons for Add, Duplicate, Remove, and Rename.

The main 3D view shows a robot model in a dark gray environment with a white grid floor. The robot is a simple gray cylinder with blue and green vertical bars on top. A yellow line connects the robot to a point labeled 'world' on the grid.

8 Self Yd p

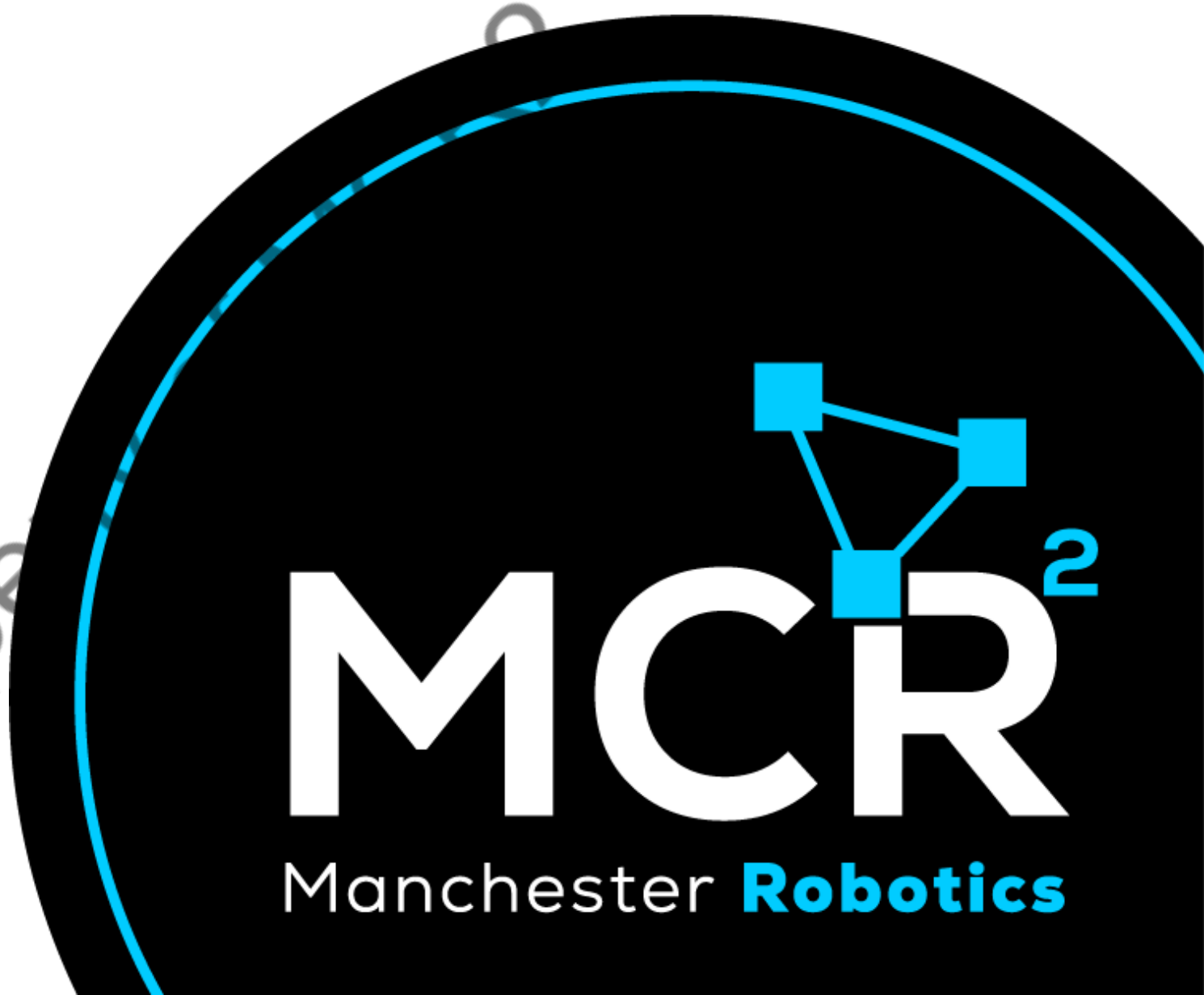
*Putting It All Together*

*{Learn, Create, Innovate};*

Prope

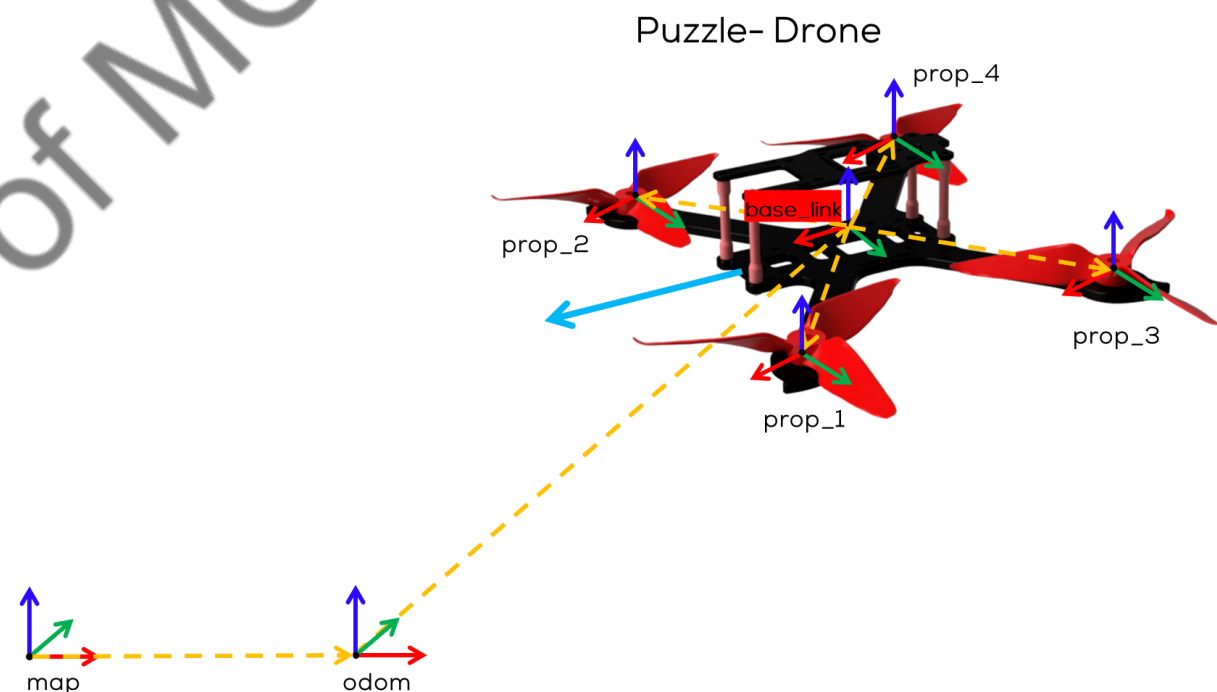
MCR<sup>2</sup>

Manchester **Robotics**



## Description

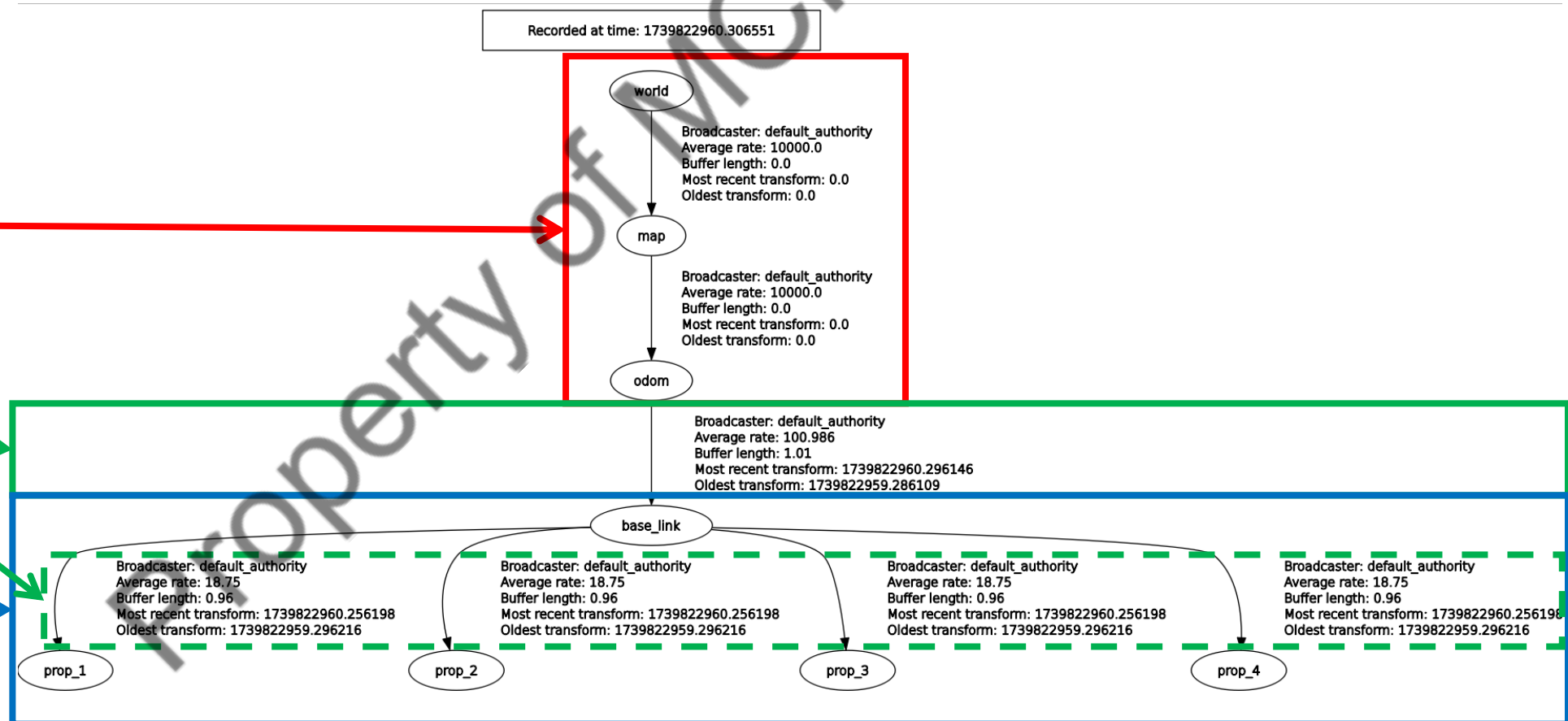
- In this activity, the user will create a link and a joint state publisher that depicts the Puzzedrone.
- The propellers will be controlled by publishing a JointState message into the /joint\_states topic.
- To decrease the complexity and decrease computational power, only the basic parts of the Puzzle Drone will be modelled (excl. motors, cables, PCB, etc.)
- The activity is divided into three sections: URDF, Joint State Publisher and Launch File.



Static Transforms and links to be defined in the launch file

Dynamic Transforms, to be defined and joints to be updated on the node "puzzle\_drone.py" (joint\_state\_pub)

Dynamic (continuous) joints and links to be defined on URDF File



## Instructions

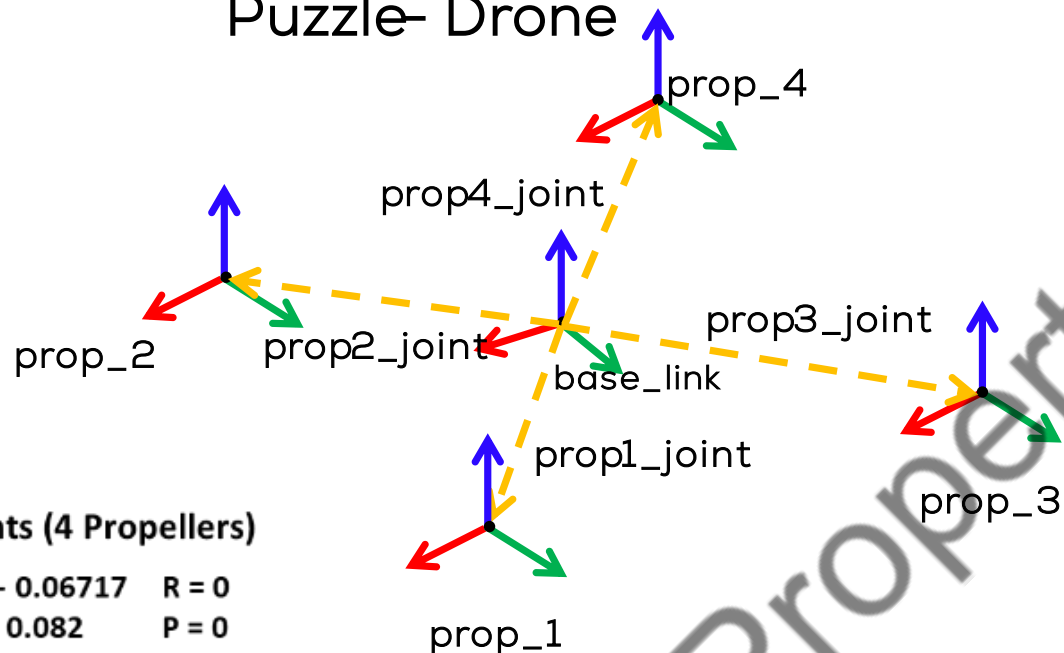
- Use the package previously created in Activity 4.
- Open the urdf file (inside the urdf folder) named “puzzle\_drone.urdf”
- Create the links and joints of the using the following information.
- The information about the joints is described in the figure.
- An example on how to define two links and a join is presented in the following slide, the complete exercise can be found in GitHub.

- Create five links using the following information:

Link name	Mesh to attach	Adjusted Mesh Pose
base_link	base_210mm.stl	x=0 y=0 z=-0.0205 r=1.57 p= 0.0 yaw=1.57
prop_1	propeller_ccw_pul ler_5in.stl	z=-0.004
prop_2	propeller_cw_pul ler_5in.stl	z=-0.004
prop_3	propeller_ccw_pul ler_5in.stl	z=-0.004
prop_4	propeller_cw_pul ler_5in.stl	z=-0.004



## Puzzle- Drone



### Joints (4 Propellers)

$X = \pm 0.06717$     $R = 0$   
 $Y = \pm 0.082$     $P = 0$   
 $Z = -0.0125$     $Y = 0$

```

...
<link name="base_link">
  <visual>
    <origin xyz="0 0 -0.0205" rpy="1.57 0 1.57"/>
    <geometry>
      <mesh filename="package://link_act/meshes/base_210mm.stl"/>
    </geometry>
    <material name="yellow">
      <color rgba="0.8 0.8 0.05 1.0"/>
    </material>
  </visual>
</link>

<link name="prop_1">
  <visual>
    <origin xyz="0 0 -0.004" rpy="0 0 0"/>
    <geometry>
      <mesh filename="package://link_act/meshes/propeller_ccw_puller_5in.stl"/>
    </geometry>
    <material name="yellow">
      <color rgba="0.8 0.8 0.05 1.0"/>
    </material>
  </visual>
</link>

...
<joint name="prop1_joint" type="continuous">
  <parent link="base_link"/>
  <child link="prop_1"/>
  <origin xyz="0.06717 0.082 -0.0125" rpy="0 0 0"/>
  <axis xyz="0 0 1"/>
</joint>

```



# Activity 5: Joint State Publisher



- Make a Ros2 Node, that broadcast a dynamic transform from the “odom” frame to the “base\_link” frame. The transform can vary in time to make the drone describe a circular trajectory.
- Publish the states of the joints "prop1\_joint", "prop2\_joint", "prop3\_joint", "prop4\_joint" in the /joint\_states topic so that the propellers act like they are rotating.
- The following slide shows the template for the Joint State Publisher

```
#Initialise Message to be published

self.ctrlJoints = JointState()
self.ctrlJoints.header.stamp = self.get_clock().now().to_msg()
self.ctrlJoints.name = ["prop1_joint", "prop2_joint",
                        "prop3_joint", "prop4_joint"]
self.ctrlJoints.position = [0.0] * 4
self.ctrlJoints.velocity = [0.0] * 4
self.ctrlJoints.effort = [0.0] * 4
```



# Activity 5: Joint State Publisher



```
import rclpy
from rclpy.node import Node
from tf2_ros import TransformBroadcaster
from geometry_msgs.msg import TransformStamped
from visualization_msgs.msg import Marker
import transforms3d
import numpy as np
```

```
class DronePublisher(Node):
```

```
    def __init__(self):
        super().__init__('frame_publisher')
```

```
        #Drone Initial Pose
```

```
        self.intial_pos_x = 1.0
```

```
        self.intial_pos_y = 1.0
```

```
        self.intial_pos_z = 1.0
```

```
        self.intial_pos_yaw = np.pi/2
```

```
        self.intial_pos_pitch = 0.0
```

```
        self.intial_pos_roll = 0.0
```

```
        #Angular velocity for the pose change and propellers
```

```
        self.omega = 0.5
```

```
        self.omega_prop = 100.0
```

```
        #Define Transformations
```

```
        self.define_TF()
```

```
        #Create Transform Broadcasters
```

```
        #Create Publishers
```

```
        #Create Joint State to be Published
```

```
        #Create a Timer
```

```
        timer_period = 0.01 #seconds
```

```
        self.timer = self.create_timer(timer_period, self.timer_cb)
```

```
        #Timer Callback
```

```
    def timer_cb(self):
```

```
        #Callback to be filled
```

```
    def define_TF(self):
```

```
        #Create Trasnform Messages here
```

```
def main(args=None):
```

```
    rclpy.init(args=args)
```

```
    node = DronePublisher()
```

```
    try:
```

```
        rclpy.spin(node)
```

```
    except KeyboardInterrupt:
```

```
        pass
```

```
    finally:
```

```
        if rclpy.ok(): # Ensure shutdown is only called once
```

```
            rclpy.shutdown()
```

```
        node.destroy_node()
```

```
if __name__ == '__main__':
```

```
    main()
```



# Activity 5: Joint State Publisher



```
def define_TF(self):
    #Create Transform Messages
    self.base_link_tf = TransformStamped()
    self.base_link_tf.header.stamp = self.get_clock().now().to_msg()
    self.base_link_tf.header.frame_id = 'odom'
    self.base_link_tf.child_frame_id = 'base_link'
    self.base_link_tf.transform.translation.x = self.intial_pos_x
    self.base_link_tf.transform.translation.y = self.intial_pos_y
    self.base_link_tf.transform.translation.z = self.intial_pos_z
    q = transforms3d.euler.euler2quat(self.intial_pos_roll,
    self.intial_pos_pitch, self.intial_pos_yaw)
    self.base_link_tf.transform.rotation.x = q[1]
    self.base_link_tf.transform.rotation.y = q[2]
    self.base_link_tf.transform.rotation.z = q[3]
    self.base_link_tf.transform.rotation.w = q[0]
```

- The transform will be declared as follows
- All transforms will be defined inside the function “define\_TF”
- The transforms and joints will be updated using a timer.
- The following slides contain the full code, the code can also be found on GitHub.



# Activity 5: Joint State Publisher



```
import rclpy
from rclpy.node import Node
from tf2_ros import TransformBroadcaster
from geometry_msgs.msg import TransformStamped
from sensor_msgs.msg import JointState
import transforms3d
import numpy as np
```

```
class DronePublisher(Node):
```

```
    def __init__(self):
        super().__init__('frame_publisher')
```

```
        #Drone Initial Pose
```

```
        self.intial_pos_x = 1.0
        self.intial_pos_y = 1.0
        self.intial_pos_z = 1.0
        self.intial_pos_yaw = np.pi/2
        self.intial_pos_pitch = 0.0
        self.intial_pos_roll = 0.0
```

```
        #Angular velocity for the pose change and propellers
```

```
        self.omega = 0.5
        self.omega_prop = 100.0
```

```
        #Define Transformations
```

```
        self.define_TF()
```

```
        #initialise Message to be published
        self.ctrlJoints = JointState()
        self.ctrlJoints.header.stamp = self.get_clock().now().to_msg()
        self.ctrlJoints.name = ["prop1_joint", "prop2_joint", "prop3_joint",
                                "prop4_joint"]
        self.ctrlJoints.position = [0.0] * 4
        self.ctrlJoints.velocity = [0.0] * 4
        self.ctrlJoints.effort = [0.0] * 4

        #Create Transform Boradcasters
        self.tf_br_base = TransformBroadcaster(self)
        #Publisher
        self.publisher = self.create_publisher(JointState, '/joint_states', 10)
        #Create a Timer
        timer_period = 0.01 #seconds
        self.timer = self.create_timer(timer_period, self.timer_cb)
```



# Activity 5: Joint State Publisher



#Timer Callback

```
def timer_cb(self):
```

```
    time = self.get_clock().now().nanoseconds/1e9
```

#Create Transform Messages

```
    self.base_link_tf.header.stamp = self.get_clock().now().to_msg()
```

```
    self.base_link_tf.transform.translation.x = self.intial_pos_x +
```

```
0.5*np.cos(self.omega*time)
```

```
    self.base_link_tf.transform.translation.y = self.intial_pos_y +
```

```
0.5*np.sin(self.omega*time)
```

```
    self.base_link_tf.transform.translation.z = self.intial_pos_z
```

```
    q = transforms3d.euler.euler2quat(self.intial_pos_roll,
```

```
self.intial_pos_pitch, self.intial_pos_yaw+self.omega*time)
```

```
    self.base_link_tf.transform.rotation.x = q[1]
```

```
    self.base_link_tf.transform.rotation.y = q[2]
```

```
    self.base_link_tf.transform.rotation.z = q[3]
```

```
    self.base_link_tf.transform.rotation.w = q[0]
```

```
    self.ctrlJoints.header.stamp = self.get_clock().now().to_msg()
```

```
    self.ctrlJoints.position[0] = self.omega_prop*time
```

```
    self.ctrlJoints.position[1] = -self.omega_prop*time
```

```
    self.ctrlJoints.position[2] = self.omega_prop*time
```

```
    self.ctrlJoints.position[3] = -self.omega_prop*time
```

```
    self.tf_br_base.sendTransform(self.base_link_tf)
```

```
    self.publisher.publish(self.ctrlJoints)
```

```
def define_TF(self):
```

#Create Trasnform Messages

```
    self.base_link_tf = TransformStamped()
```

```
    self.base_link_tf.header.stamp = self.get_clock().now().to_msg()
```

```
    self.base_link_tf.header.frame_id = 'odom'
```

```
    self.base_link_tf.child_frame_id = 'base_link'
```

```
    self.base_link_tf.transform.translation.x = self.intial_pos_x
```

```
    self.base_link_tf.transform.translation.y = self.intial_pos_y
```

```
    self.base_link_tf.transform.translation.z = self.intial_pos_z
```

```
    q = transforms3d.euler.euler2quat(self.intial_pos_roll,  
self.intial_pos_pitch, self.intial_pos_yaw)
```

```
    self.base_link_tf.transform.rotation.x = q[1]
```

```
    self.base_link_tf.transform.rotation.y = q[2]
```

```
    self.base_link_tf.transform.rotation.z = q[3]
```

```
    self.base_link_tf.transform.rotation.w = q[0]
```

```
def main(args=None):
```

```
    rclpy.init(args=args)
```

```
    node = DronePublisher()
```

```
    try:
```

```
        rclpy.spin(node)
```

```
    except KeyboardInterrupt:
```

```
        pass
```

```
    finally:
```

```
        if rclpy.ok(): # Ensure shutdown is only called once
```

```
            rclpy.shutdown()
```

```
            node.destroy_node()
```

```
if __name__ == '__main__':
```

```
    main()
```



# Activity 5: Launch File



- Open the previously created launch file “puzzledrone\_launch.py”
- Define two static transforms
  - world -> map (same pose)
  - map -> odom [x= 2, y=1, z=-0.0, r=0, p=0.0, yaw=0]

```
static_transform_node = Node(
    package='tf2_ros',
    executable='static_transform_publisher',
    arguments = ['--x', '2', '--y', '1', '--z', '0.0',
                 '--yaw', '0.0', '--pitch', '0', '--roll', '0.0',
                 '--frame-id', 'map', '--child-frame-id', 'odom']
)

static_transform_node_2 = Node(
    package='tf2_ros',
    executable='static_transform_publisher',
    arguments = ['--x', '0', '--y', '0', '--z', '0.0',
                 '--yaw', '0.0', '--pitch', '0', '--roll', '0.0',
                 '--frame-id', 'world', '--child-frame-id', 'map']
)
```

- Add the robot state publisher

```
urdf_file_name = 'puzzle_drone.urdf'
urdf = os.path.join(
    get_package_share_directory('link_act'),
    'urdf',
    urdf_file_name)

with open(urdf, 'r') as infp:
    robot_desc = infp.read()

robot_state_pub_node = Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    name='robot_state_publisher',
    output='screen',
    parameters=[{'robot_description': robot_desc}],
    arguments=[urdf]
)
```



# Activity 5: Launch File

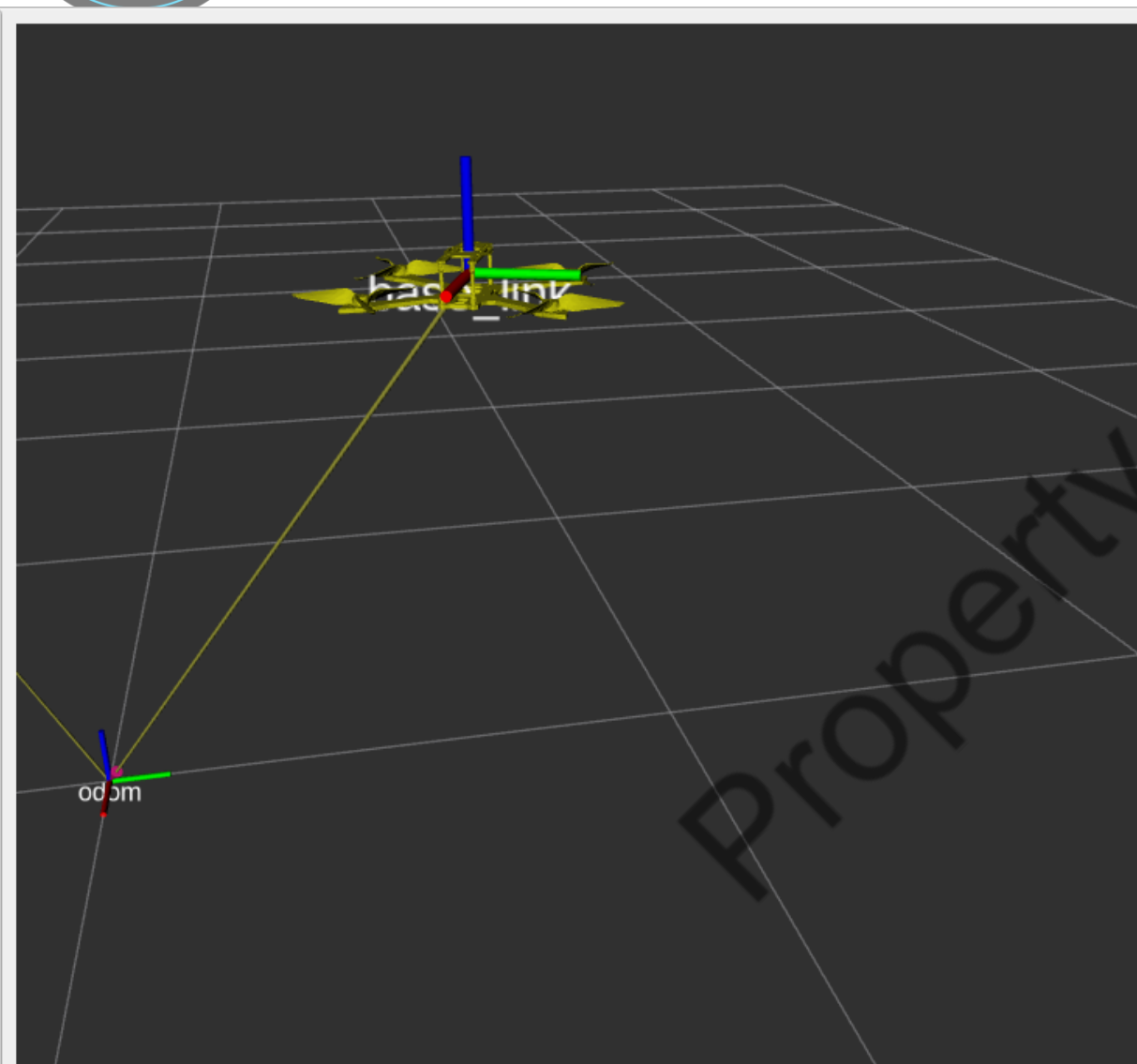
---



- Add the newly developed “puzzle\_drone.py” node to the launch file.
- You can add RVIZ and TF\_Tree to verify the transforms.
- The full Launch File can be found in GitHub.
- Build the package
- Launch the package
- Add the TF's to RVIZ
- To visualise the model in RVIZ select “world” as Fixed Frame.
- In RVIZ press “Add>>By display type>>RobotModel”. In the RobotModel select s “Description Topic” “/robot\_description”.



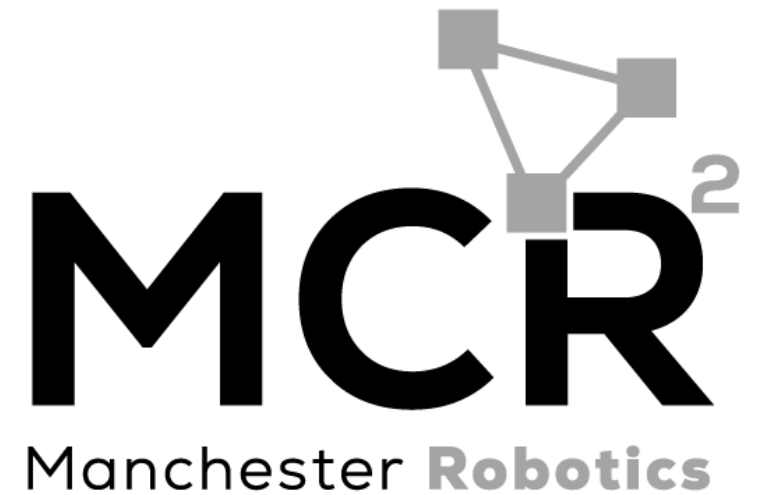
# Activity 5: Results



Thank you

Property of MCR2

*{Learn, Create, Innovate};*

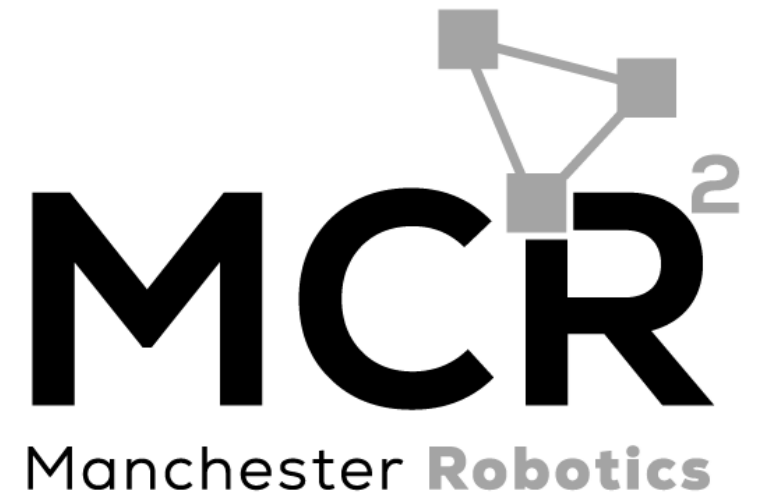


# T&C

*Terms and conditions*

*{Learn, Create, Innovate};*

Property of MCR2





# Terms and conditions

---



- *THE PIECES, IMAGES, VIDEOS, DOCUMENTATION, ETC. SHOWN HERE ARE FOR INFORMATIVE PURPOSES ONLY. THE DESIGN IS PROPRIETARY AND CONFIDENTIAL TO MANCHESTER ROBOTICS LTD. (MCR2). THE INFORMATION, CODE, SIMULATORS, DRAWINGS, VIDEOS PRESENTATIONS ETC. CONTAINED IN THIS PRESENTATION IS THE SOLE PROPERTY OF MANCHESTER ROBOTICS LTD. ANY REPRODUCTION, RESELL, REDISTRIBUTION OR USAGE IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF MANCHESTER ROBOTICS LTD. IS STRICTLY PROHIBITED.*
- *THIS PRESENTATION MAY CONTAIN LINKS TO OTHER WEBSITES OR CONTENT BELONGING TO OR ORIGINATING FROM THIRD PARTIES OR LINKS TO WEBSITES AND FEATURES IN BANNERS OR OTHER ADVERTISING. SUCH EXTERNAL LINKS ARE NOT INVESTIGATED, MONITORED, OR CHECKED FOR ACCURACY, ADEQUACY, VALIDITY, RELIABILITY, AVAILABILITY OR COMPLETENESS BY US.*
- *WE DO NOT WARRANT, ENDORSE, GUARANTEE, OR ASSUME RESPONSIBILITY FOR THE ACCURACY OR RELIABILITY OF ANY INFORMATION OFFERED BY THIRD-PARTY WEBSITES LINKED THROUGH THE SITE OR ANY WEBSITE OR FEATURE LINKED IN ANY BANNER OR OTHER ADVERTISING.*