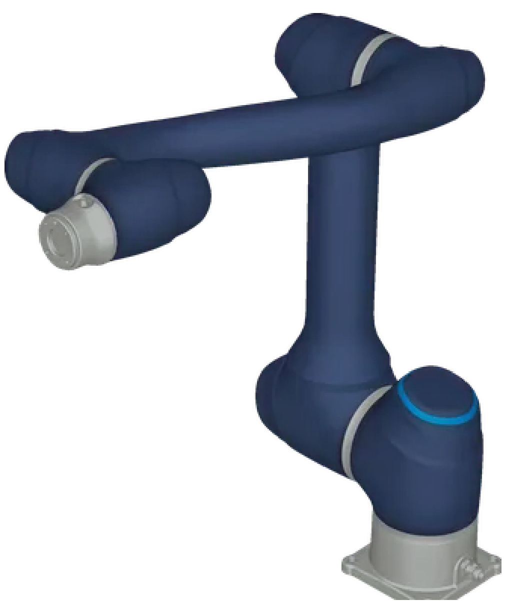
# Simulate a 6DoF Robotic Arm in Gazebo and ROS2

In this tutorial, you will learn how to simulate a robotic arm from scratch. We will be using a 6DoF robotic arm from Doosan Robotics. Gazebo and ROS2 are the software to perform this simulation. All the code, URDFs and configuration files can be found and downloaded in my Github Repository

- Github Repository (LINK)



## 1 Outcomes after this section

- Correct installation of ROS2 control and Gazebo packages
- Correct configuration of your files for ROS2 (control and simulation)
- Visualize your robotic arm in RVIZ 2
- Simulate your robotic arm in Gazebo
- Make your robot execute a simple trajectory

## 2 Prerequisites

In order to succeed with this tutorial, please make sure you have installed and ready the following points:

- Ubuntu 20.04
- Full installation of ROS2 Foxy

## 3 About the Robotic Arm

We are going to use a 6 DoF robotic arm from Doosan Robotics. Each part that makes up this robot for simulation is building using a COLLADA file (a Digital Asset Exchange file format for 3D applications) and the URDFs. Thanks to Doosan Robotics and its repository we could obtain all these files ( LINK )

We are going to simulate the A0912 robotic arm. You can have a look at this robot features here (LINK)

## 4 Installation Packages and Run the Simulation

Let's install the packages and run the simulation so you can see what we will get with this repository. The first thing you have to do is create your workspace, in a new terminal write:

Install some dependencies and necessary packages:

Then you need to download and build  in your workspace the following packages:

- ros2_control framework (LINK)

- gazebo_ros2_control (LINK)

*cd ros2_ws/src*
*git clone -b foxy https://github.com/ros-simulation/gazebo_ros2_control.git*
*cd ..*
*colcon build*

Make sure to install these packages FROM SOURCE, and **point to the correct branch** (FOXY) otherwise, some problems could appear. Now you can clone our repository in your workspace and build the whole workspace.

*cd ros2_ws/src*
*git clone https://github.com/dvalenciar/robotic_arm_environment.git*
*cd ..*
*colcon build*

Now open a new terminal and write the following lines:

*. install/setup.bash*

*ros2 launch my_doosan_pkg my_doosan_gazebo.launch.py*

Open another new terminal and write the following lines:

*cd ros2_ws*

*. install/setup.bash*

*ros2 launch my_doosan_pkg my_doosan_rviz.launch.py*

If the installation was correct and you did not get any error, Gazebo will be open and you should be able to see a beautiful simulation of the robotic arm.  Also, Rviz will be launched, and you can move the robot using the joint state publisher window


00:30 / 00:32

Note: The robot will not move in Gazebo yet because we are not using any controller; only the robot in Rviz will move. I will explain this in few paragraphs later.

## 5 Step by Step Configuration

Now that you have seen part of the results and the robot working, it is time to **replicate the entire process** from scratch, step by step, with all the necessary settings, configuration files and certain "tricks", which I have discovered over many hours of practice.

### 5.1 Create your ROS package

We will start creating the ROS package where you will use for your simulation. This package will contain all the code, configuration files and launchers, but first thing first, let's create a python package.

*cd /ros2_ws/src/*

*ros2 pkg create --build-type ament_python my_doosan_pkg --dependencies rclpy*

*cd ..*

*colcon build --packages-up-to my_doosan_pkg*

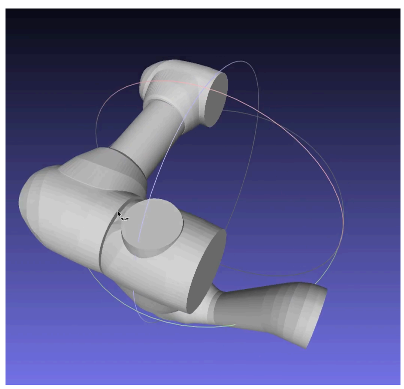
Now that you have created your packages, we will be creating the necessary folders that will contain the descriptions and config files.

*cd my_doosan_pkg*
*mkdir config description launch rviz worlds*

Have a look inside in your my_doosan_pkg folder you should have something like this


### 5.2 Add the meshes files

As I mentioned previously, we are going to use a 6 DoF robotic arm from Doosan Robotics, therefore, we need a 3D prototype (i.e. a 3D model in CAD) of each part that makes up this robot. The simulator needs these files to create a solid and realistic simulation. This tutorial will be using DAE files.



I have provided you all the DAE files (which I have previously sort and downloaded from Doonsa Robotic Repository) so you have to create a new folder called /meshes inside the /description folder and **add** the complete /a0912_blue folder which contains all necessary meshes files. You can obtain these files from the repository here ([LINK]).


### 5.2 Write the XACRO-URDF files

Let's make things more interesting. Now we are going to create our URDF-Xacro files. These files are extremely important because contain a tree structure telling about how to hold the robot body in terms of links and joints. Basically, these files tell the simulator how and where to locate each link and joint. Also, it is here where you import your meshes (the DAE files previously saved in the meshes folder). So, inside your /description folder you need to create a new folder called xacro. Inside that /xacro folder you have to create a file with the extension .xacro (for our case you have called this file macro.a0912.blue.xacro).

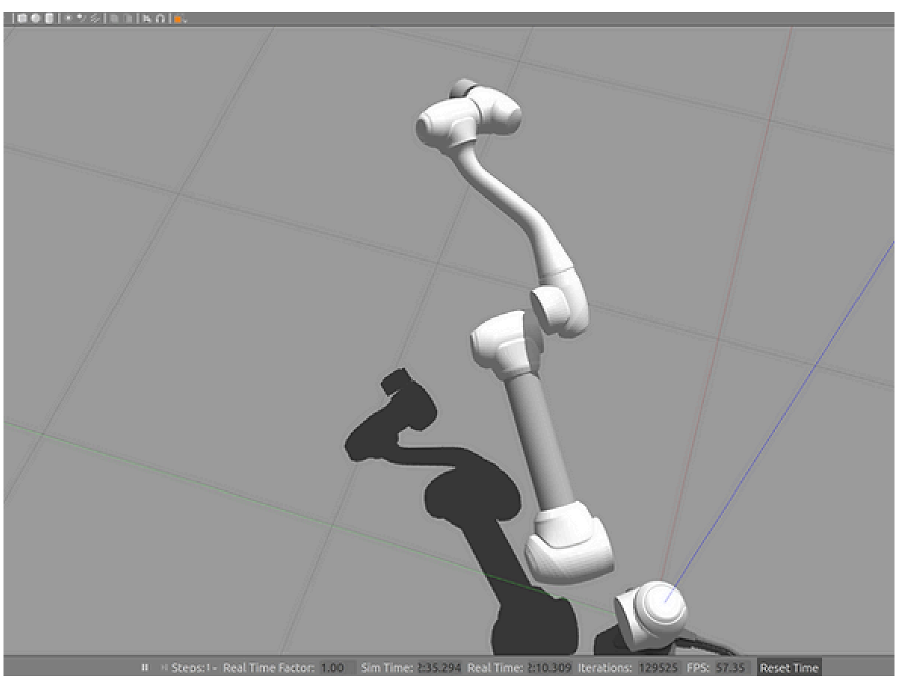
NOTE: Xacro files and URDF files are not differents; xacro is just another way of defining a URDF, not an alternative to it. The biggest difference is the use of macros making certain things easier.

Let's start by writing the first link (base) and the first joint in our xacro file. As you can see in the next image there is a tag called **"mesh"** inside the visual and collision tag and that is where the mesh file is added for each link. You have to do this for all the links and joints that compose your robot. This could be a very difficult and tedious job, however, the majority of the commercial robots include this description or can be auto-generated using external software. For our case, you can obtain the XACRO-URDF file complete here (LINK).

```
<!-- Base -->
<link name="base_0">
    <inertial>
        <mass value="3.635"/>
        <origin xyz="4e-06   -0.001016     0.08181"/>
        <inertia ixx="0.02216" ixy="5.8001e-06" ixz="6.1941e-06" iyy="0.024835"
            iyz="-5.3034e-05"
            izz="0.021063"/>
    </inertial>
    <visual>
        <origin rpy="0 0 0" xyz="0 0 0" />
        <geometry>
            <mesh filename="file:///$(find my_doosan_pkg)/description/meshes/a0912_blue/A0912_0_0.dae" scale="0.001 0.001 0.001" />
        </geometry>
    </visual>
    <collision>
        <geometry>
            <mesh filename="file:///$(find my_doosan_pkg)/description/meshes/a0912_blue/A0912_0_0.dae" scale="0.001 0.001 0.001" />
        </geometry>
    </collision>
</link>


<!-- joint 1 -->
<joint name="joint1" type="revolute">
    <parent link="base_0" />
    <child link="link1" />
    <origin rpy="0 0 0" xyz="0 0 0.195" />
    <axis xyz="0 0 1" />
    <limit effort="411" lower="-6.2832" upper="6.2832" velocity="3.1416"/>
    <dynamics damping="60.0" friction="4.0" />
</joint>
```

Something that I included manually in this xacro file is the **damping and friction parameters** (these parameters were not included in the original Doosan xacro-urdf). But after a long analysis, I discovered that these parameters help the robot to look more solid during the simulation, also prevent the robot from making strange movements, especially in the joints. The next image is an example of what could happen if you do not include the damping and friction in your description file. So make sure to try and select the correct values. See the URDF page for more details about damping and friction. (LINK).



### 5.3 Control plugins and configurations

Another very important thing that the original Doosan repository does not include (at least I could not find it) is the use of the necessary plugins for ros2_control and gazebo_ros2_control inside the urdf-xacro files (perhaps that is the reason why they mention the original repository does not run with the latest version of ros2_control). Therefore, I created and add this file, making this tutorial full compatible with the latest ros2_control version.

So inside the /xacro folder you create a new file and called it macro.gazebo_config_control.xacro. This file will include ros2_control tag and defines a system. We will call this file from our main xacro file (macro.a0912.blue.xacro). So let's add the ros2_control tag and the necessary configuration (command interface and possible state interfaces)for each joint.

```
<ros2_control name="GazeboSystem" type="system">
    <hardware>
        <plugin>gazebo_ros2_control/GazeboSystem</plugin>
    </hardware>

    <joint name="joint1">
        <command_interface name="position">
            <param name="min">-6.2832</param>
            <param name="max">6.2832</param>
        </command_interface>
        <state_interface name="position"/>
        <state_interface name="velocity"/>
        <state_interface name="effort"/>
    </joint>

    <joint name="joint2">
        <command_interface name="position">
            <param name="min">-6.2832</param>
            <param name="max">6.2832</param>
        </command_interface>
        <state_interface name="position"/>
        <state_interface name="velocity"/>
        <state_interface name="effort"/>
    </joint>

    <joint name="joint3">
        <command_interface name="position">
            <param name="min">-2.7925</param>
            <param name="max">2.7925</param>
        </command_interface>
        <state_interface name="position"/>
        <state_interface name="velocity"/>
        <state_interface name="effort"/>
    </joint>
```

Also, inside this file, you have to load the gazebo plugin and the configuration file of the controller. You can find the complete file here (LINK)

```
<gazebo>
    <plugin name="gazebo_ros2_control" filename="libgazebo_ros2_control.so">
        <robot_sim_type>gazebo_ros2_control/GazeboSystem</robot_sim_type>
        <parameters>$(find my_doosan_pkg)/config/simple_controller.yaml</parameters>
    </plugin>
</gazebo>
```

Note: this is one of the main differences with respect to ROS1; that is, in ROS1 the configuration yaml file was loaded in the launch file while in ROS2 it is loaded in the description URDF-xacro file.

The configuration file containing the settings of the controller. Therefore, inside the folder /config you will create a new XML file and called simple_controller.yaml.

```yaml
controller_manager:
  ros__parameters:
    update_rate: 100 # Hz


    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster


    joint_trajectory_controller:
      type: joint_trajectory_controller/JointTrajectoryController


joint_trajectory_controller:
  ros__parameters:
    joints:
      - joint1
      - joint2
      - joint3
      - joint4
      - joint5
      - joint6

    write_op_modes:
      - joint1
      - joint2
      - joint3
      - joint4
      - joint5
      - joint6

    interface_name: position

    command_interfaces:
      - position

    state_interfaces:
      - position
      - velocity

    state_publish_rate: 50.0 # Defaults to 50
    action_monitor_rate: 20.0 # Defaults to 20

    allow_partial_joints_goal: false # Defaults to false
    hardware_state_has_offset: true
    deduce_states_from_derivatives: true
```

This file is self-explanatory; it contains the type of controller and the joints to which the controller will be applied. For our case, we will be working with Joint Trajectory Controller --> Position Controller. You can find this YAML file here (LINK).

## 5.4 Launch files and Setup configurations

We have reached the point where we began to create our code to simulate the robotic arm in Gazebo. So we will make a launch file, which will help us start all the necessary files and packages. Inside the /launch folder, we have to create a new file; we have named this file my_doosan_gazebo_controller.launch.py

We start importing all the necessary libraries and then reading the xacro file from the share directory of the package. Then, we will be executing is robot_state_publisher, which will publish all the states of the robot to the TF and put them on the parameter server. The next step will load the gazebo world and spawn our robot. Finally, we will load and start our controllers (previously defined in the config/*.yaml file). The following image is a complete example of our launch file, but in case you need it, you can find this file here (LINK)

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

def generate_launch_description():

    robot_model = 'a0912'
    #robot_model = 'm1013'

    xacro_file = get_package_share_directory('my_doosan_pkg') + '/description'+'/xacro/'+ robot_model +'.urdf.xacro'

    # Robot State Publisher
    robot_state_publisher = Node(package    ='robot_state_publisher',
                                 executable ='robot_state_publisher',
                                 name       ='robot_state_publisher',
                                 output     ='both',
                                 parameters =[{'robot_description': Command(['xacro', ' ', xacro_file])
                                }])

    # Spawn the robot in Gazebo
    spawn_entity_robot = Node(package     ='gazebo_ros',
                              executable  ='spawn_entity.py',
                              arguments   = ['-entity', 'my_doosan_robot', '-topic', 'robot_description'],
                              output      ='screen')

    # Gazebo
    world_file_name = 'my_empty_world.world'
    world = os.path.join(get_package_share_directory('my_doosan_pkg'), 'worlds', world_file_name)
    gazebo_node = ExecuteProcess(cmd=['gazebo', '--verbose', world,'-s', 'libgazebo_ros_factory.so'], output='screen')

    # load and START the controllers in launch file

    load_joint_state_broadcaster = ExecuteProcess(
                                        cmd=['ros2', 'control', 'load_controller', '--set-state', 'start','joint_state_broadcaster'],
                                        output='screen')

    load_joint_trajectory_controller = ExecuteProcess(
                                        cmd=['ros2', 'control', 'load_controller', '--set-state', 'start', 'joint_trajectory_controller'],
                                        output='screen')

    return LaunchDescription([robot_state_publisher, spawn_entity_robot, gazebo_node, load_joint_state_broadcaster, load_joint_trajectory_controller  ])
```

When you are working with python packages in ROS2, you need to configure the **setup.py** file in order to ROS find the folders and files that you just created. Therefore open your setup.py ([LINK]) and add the following lines within the data_files brackets:

Finally, save all these files then build the package and launch it as follow:

*cd ros2_ws*

*colcon build --packages-up-to my_doosan_pkg*

*. install/setup.bash*

*ros2 launch my_doosan_pkg my_doosan_gazebo_controller.launch.py*

```
setup.py
import os
from glob import glob
from setuptools import setup

package_name = 'my_doosan_pkg'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    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.py'))),
        (os.path.join('share', package_name, 'rviz'), glob(os.path.join('rviz', '*.rviz'))),
        (os.path.join('share', package_name, 'config'), glob(os.path.join('config', '*.*'))),
        (os.path.join('share', package_name, 'worlds'), glob(os.path.join('worlds', '*.world'))),
        (os.path.join('share', package_name, 'description', 'urdf'), glob(os.path.join('description','urdf', '*.urdf'))),
        (os.path.join('share', package_name, 'description', 'xacro'), glob(os.path.join('description','xacro', '*.xacro'))),
        (os.path.join('share', package_name, 'description', 'meshes', 'a0912_blue'), glob(os.path.join('description','meshes','a0912_blue', '*.dae'))),
        (os.path.join('share', package_name, 'description', 'meshes', 'm1013_white'), glob(os.path.join('description','meshes','m1013_white', '*.dae'))),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='david',
    maintainer_email='david@todo.todo',
    description='TODO: Package description',
```

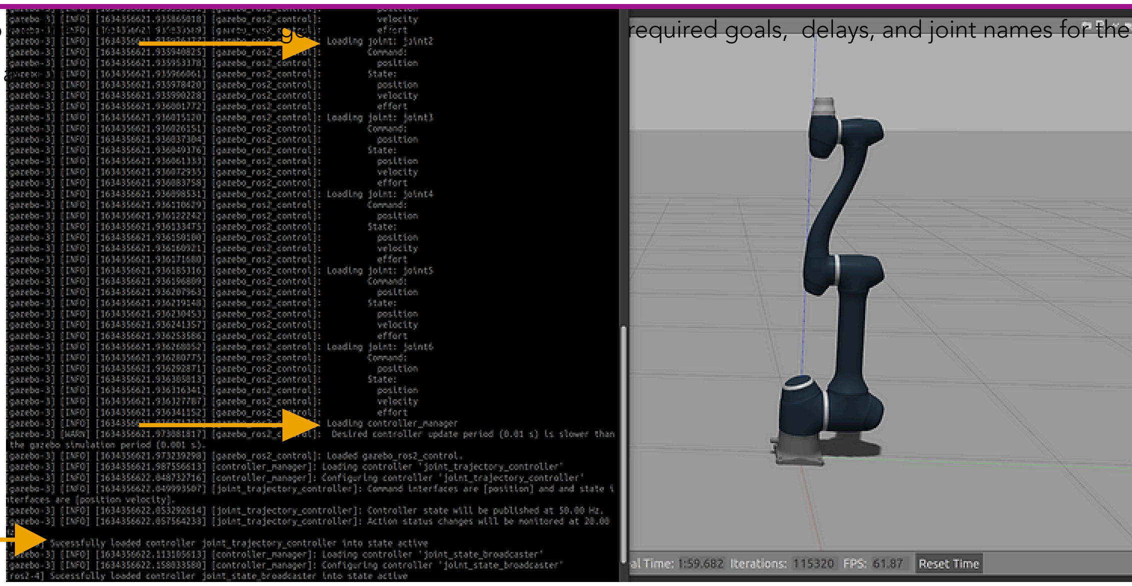
If everything goes well and you do not get errors, you should receive the following response in your terminal, and Gazebo will start with our robot.

## 5.5 Node file for Trajectory

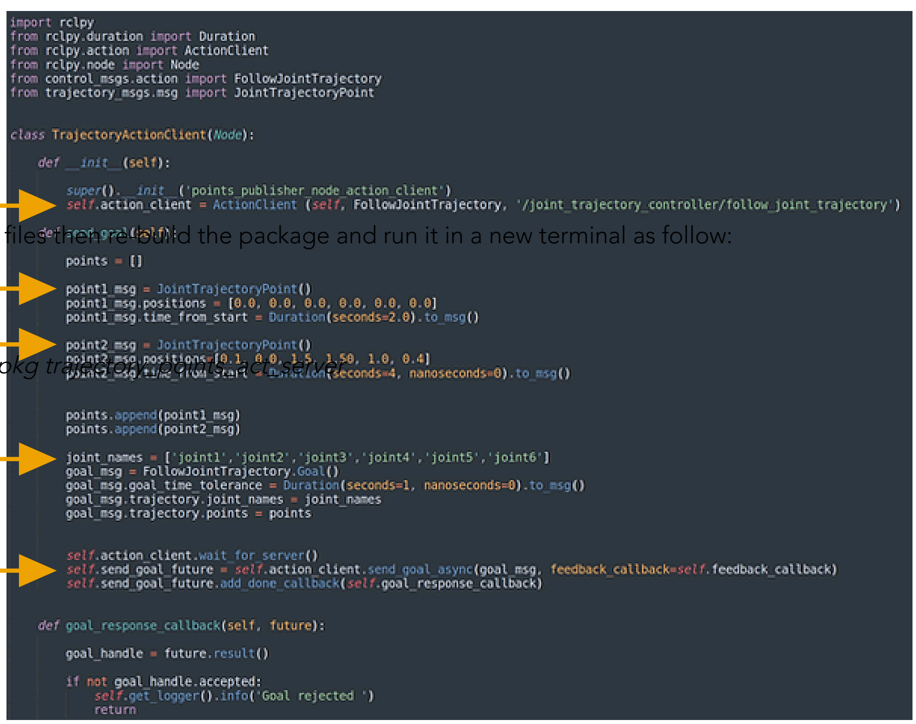
We have reached the final part, and probably the most interesting, of this tutorial where we will move the robot's joint to a specific point. We will write a ROS node using an action-client to send a request to move the joints of our robotic arm. Therefore, inside the /my_doosan_pkg folder, you need to create a new python file. We have called this file

We have to ... required goals, delays, and joint names for the controller

to take the ...



Here as well you have to add this python file in the setup.py file to be executable and ROS can run it



```python
import rclpy
from rclpy.duration import Duration
from rclpy.action import ActionClient
from rclpy.node import Node
from control_msgs.action import FollowJointTrajectory
from trajectory_msgs.msg import JointTrajectoryPoint

class TrajectoryActionClient(Node):

    def __init__(self):

        super().__init__('points_publisher_node_action_client')
        self.action_client = ActionClient (self, FollowJointTrajectory, '/joint_trajectory_controller/follow_joint_trajectory')

    def send_goal(self):

        points = []

        point1_msg = JointTrajectoryPoint()
        point1_msg.positions = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        point1_msg.time_from_start = Duration(seconds=2.0).to_msg()

        point2_msg = JointTrajectoryPoint()
        point2_msg.positions = [0.1, 0.0, 1.5, 1.50, 1.0, 0.4]
        point2_msg.time_from_start = Duration(seconds=4, nanoseconds=0).to_msg()

        points.append(point1_msg)
        points.append(point2_msg)

        joint_names = ['joint1','joint2','joint3','joint4','joint5','joint6']
        goal_msg = FollowJointTrajectory.Goal()
        goal_msg.goal_time_tolerance = Duration(seconds=1, nanoseconds=0).to_msg()
        goal_msg.trajectory.joint_names = joint_names
        goal_msg.trajectory.points = points


        self.action_client.wait_for_server()
        self.send_goal_future = self.action_client.send_goal_async(goal_msg, feedback_callback=self.feedback_callback)
        self.send_goal_future.add_done_callback(self.goal_response_callback)

    def goal_response_callback(self, future):

        goal_handle = future.result()

        if not goal_handle.accepted:
            self.get_logger().info('Goal rejected ')
            return
```

Finally, save all these files then re-build the package and run it in a new terminal as follow:

*. install/setup.bash*

*ros2 run my_doosan_pkg trajectory_points_act_server*

```
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='david',
    maintainer_email='david@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'trajectory_points_act_server = my_doosan_pkg.joint_points_act_service:main',
        ],
    },
)
```

**Email**

dval035@aucklanduni.ac.nz

# Comments?

Any questions or comments about this post will always be welcome. If you have something to say, please do not hesitate to contact me.

First Name

Last Name

Email *

Feedback or Comments

Submit