ROS2 Differential Robot --A Brief Overview

This documentation teaches the step-by-step process to create your own 2-wheeled differential robot using ROS2-humble. For continuity, the document will be divided into three major subdivisions, **Design and Package creation**, **Controllers and Sensors, and Localization.**

Github link to the files:https://github.com/parapara29/differential_ros2

These are the basic requirements of your system you need before you can get started:

- Ubuntu 22.04 LTS (https://releases.ubuntu.com/jammy/)
- ROS2 Humble (https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html), pass on the -full tag with the installation command like so:

```
sudo apt install ros-humble-desktop-full
```

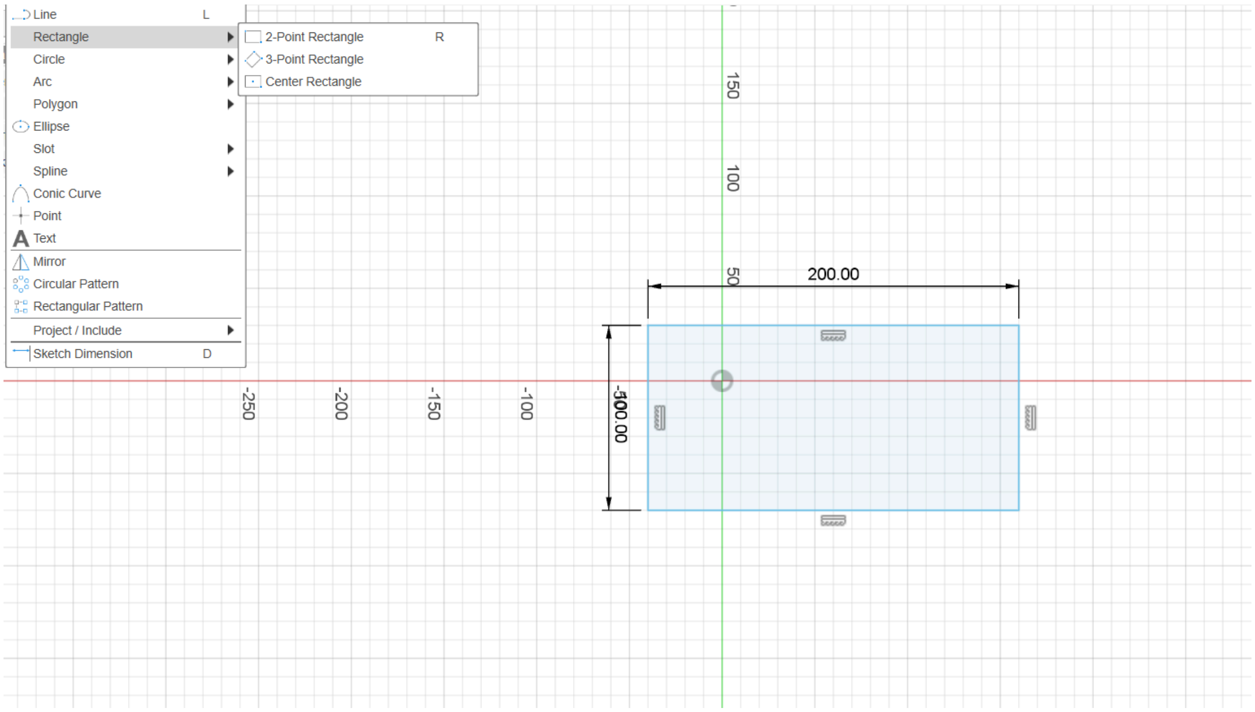
Let's get started!

Design and Package creation

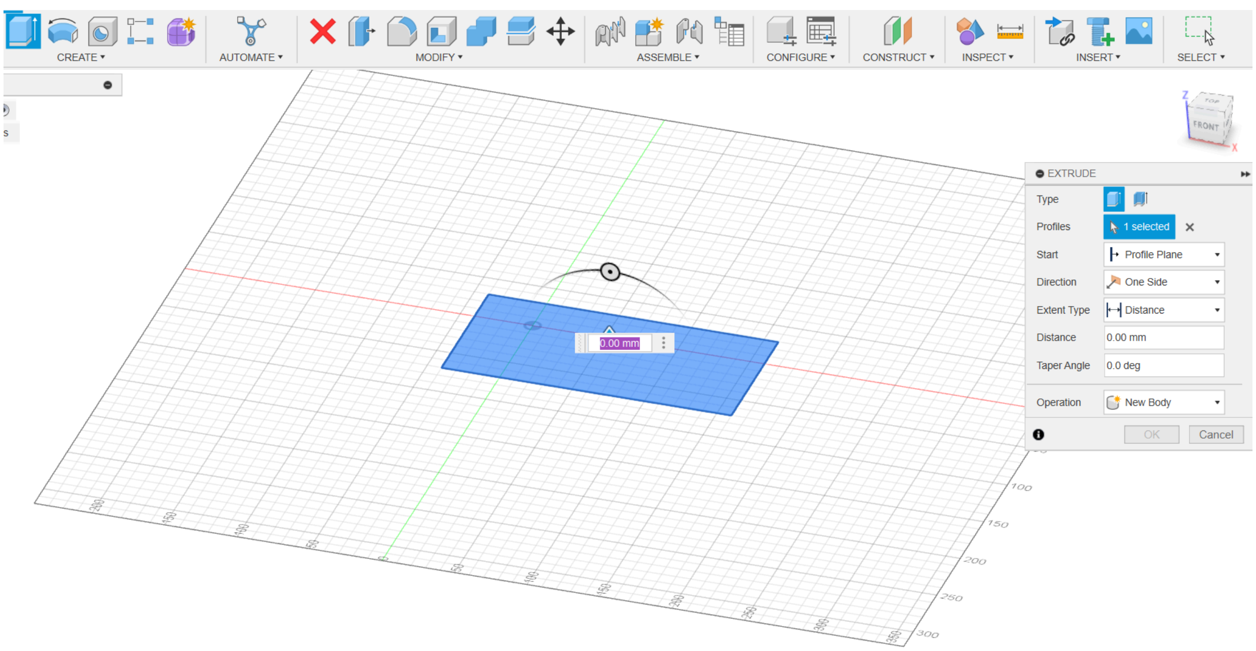
So the first step for visualizing your robot via ROS2, you need a good URDF. Fusion 360 was used to design, and export valid ROS2 files for this purpose, however, you can go forward with any software you prefer to make your model and you can use valid extensions for that software for exporting the file to URDF.

Following are the steps to design and export a ROS2 pkg from Fusion 360:

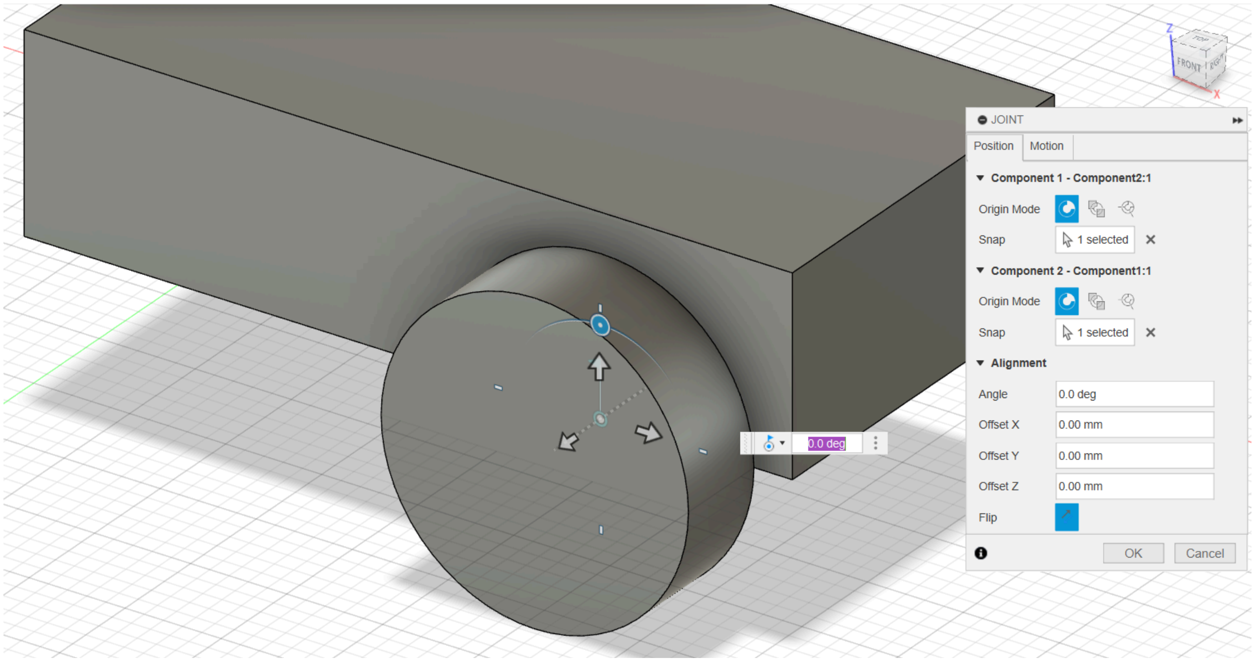
- First, let's get started with the design of the robot. We will be going for quite a basic look. Two wheels and a castor wheel on a rectangle box should do.
- Before that, make sure the Z-axis is pointing upwards in fusion and you use the x and y plane to design, and place the robot.
- Make a rectangular box in fusion according to your dimensional requirements, like so:



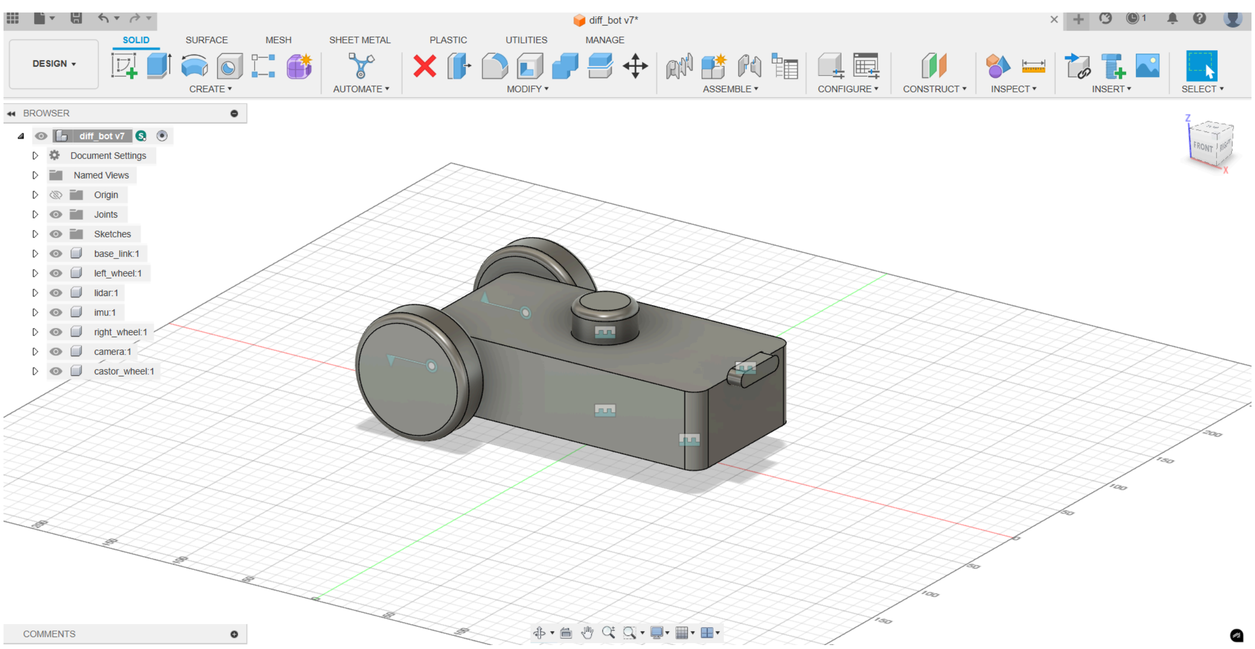
Extrude it in Z direction.



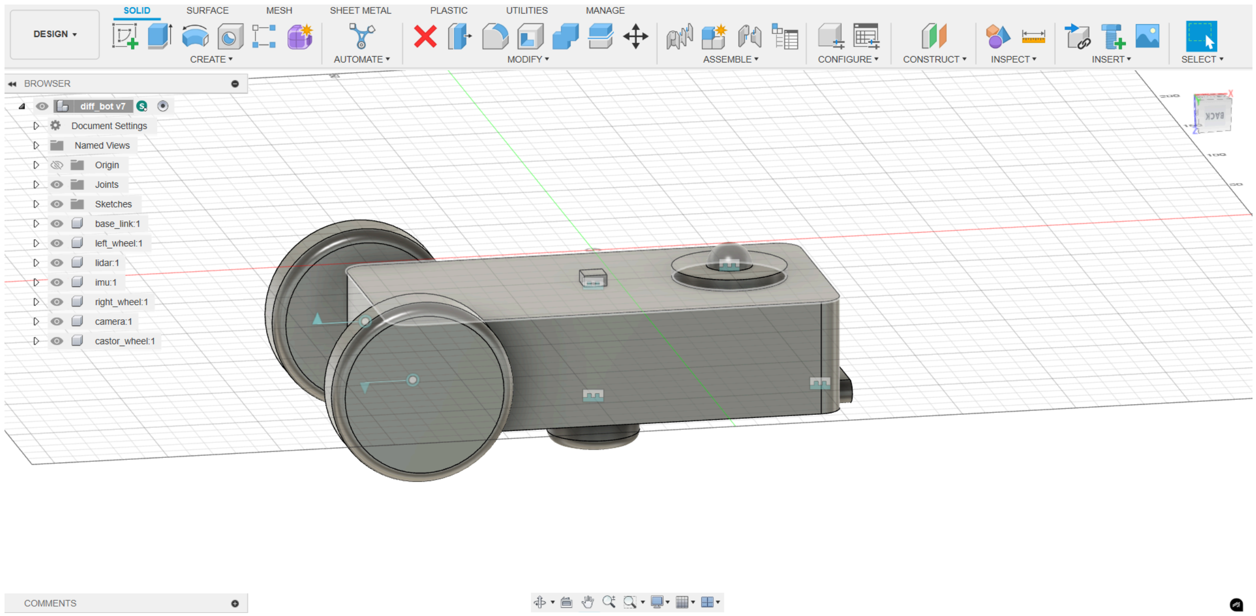
Satisfied with your rectangle, rename it as base_link. Design 2 cylindrical wheels, and attach them to the the rectangular box via revolute joint type, like so:



Now add design and add your sensors, namely, camera (simple rectangle attached via rigid joint on the top face of the base_link will do), lidar (a cylinder attached via rigid joint on the top face of the base_link would do), IMU, a small rectangle at the bottom of the base link would do. All in all the final robot would look like this:

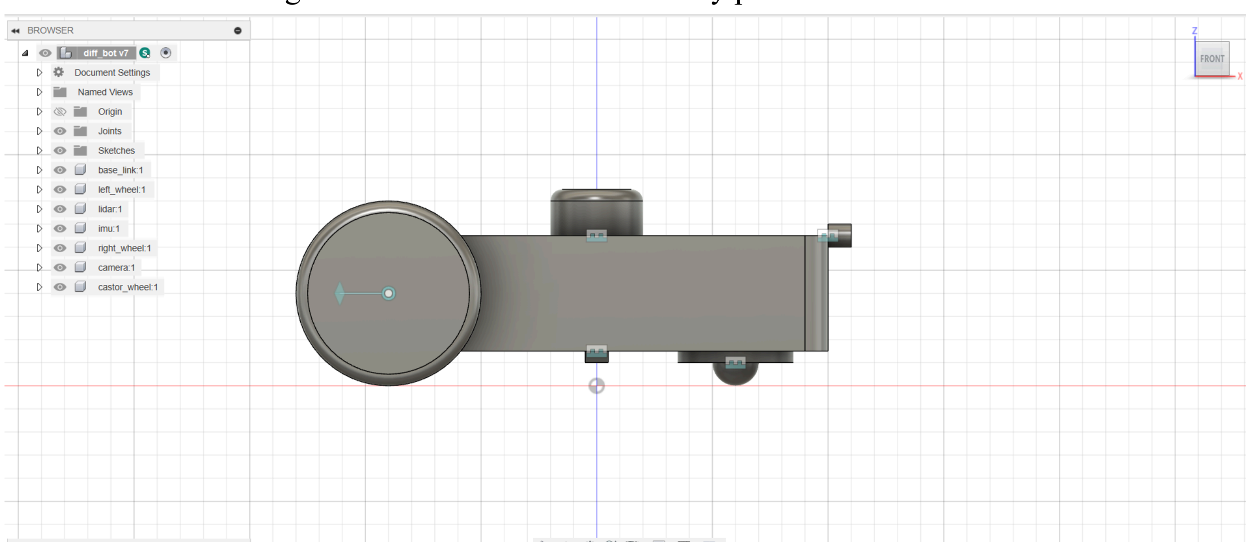


I made it a little bougie by adding fillets.



Bottom view of the robot, for the castor, a simple hemisphere attached via rigid joint will work.

- The robot should be right at the center and above the x-y plane like so:



- Remember the dimensions of your robot! You will need them for inverse kinematics.

- After you are finished and satisfied with your design, you can install the fusion2urdf2-ros2 extension in your fusion 360 by following the steps given in this link: https://github.com/dheena2k2/fusion2urdf-ros2 . Some basic tips, make sure you don't have nested components, and even after it, if you are facing issues exporting your robot to ros2, just turn the fusion design history off. Notice the component tree for my case. This is the ideal to have to get the correct export.

- Time to visualize the robot in gazebo via ROS2!

Follow these steps to optimize your package and visualize it in a gazebo environment:

- First off, let's get started with making a workspace where all the code for our robot will be developed. Pass the following command:

```
mkdir -p <your_preferred_ws_name>/src
```

This will make a folder and a sub folder in your home directory.

- You can now paste the package generated by the fusion extension into the src folder of your workspace. This is where all the packages, nodes, and services for your robot will be developed!

- cd into the src directory of your workspace by passing the command:

```
cd <your_preferred_ws_name>/src
```

- Your folder structure should look something like this, the package name for my case was diff_bot_description, however, it can vary for your case.

```
.
├── diff_bot_description
│   ├── config
│   │   ├── diff_controller.yaml
│   │   ├── display.rviz
│   │   └── ekf.yaml
│   ├── diff_bot_description
│   │   ├── __init__.py
│   │   └── odom_tf.py
│   ├── launch
│   │   ├── display.launch.py
│   │   └── gazebo.launch.py
│   ├── meshes
│   │   ├── base_link.stl
│   │   ├── camera_1.stl
│   │   ├── castor_wheel.stl
│   │   ├── imu_1.stl
│   │   ├── left_wheel_1.stl
│   │   ├── lidar_1.stl
│   │   └── right_wheel_1.stl
│   ├── package.xml
│   ├── resource
│   │   └── diff_bot_description
│   ├── setup.cfg
│   ├── setup.py
│   ├── test
│   │   ├── test_copyright.py
│   │   ├── test_flake8.py
│   │   └── test_pep257.py
│   └── urdf
│       ├── diff_bot.gazebo
│       ├── diff_bot.trans
│       ├── diff_bot.xacro
```

| └── materials.xacro

- Since the ROS2 was updated a lot after the inception of this package, we will have to make some changes, to get rid of some warnings. You will find a setup.cfg file in your robot package, where you will need to edit line 2 from script-dir to script_dir and line 4 from install-scripts to install_scripts.

- You will need to add an empty link to your urdf like shown in the following code. Add it to .xacro file in your urdf directory:

```
<link name="base_footprint">
   </link>
<joint name="base_footprint_joint" type="fixed">
       <parent link="base_footprint"/>
       <child link="base_link"/>
       <origin xyz="0 0 0" rpy="0 0 0"/>
   </joint>
```

- After this step, open up your terminal, cd into the workspace, and build your package by passing:

```
cd <your robot ws>
colcon build
```

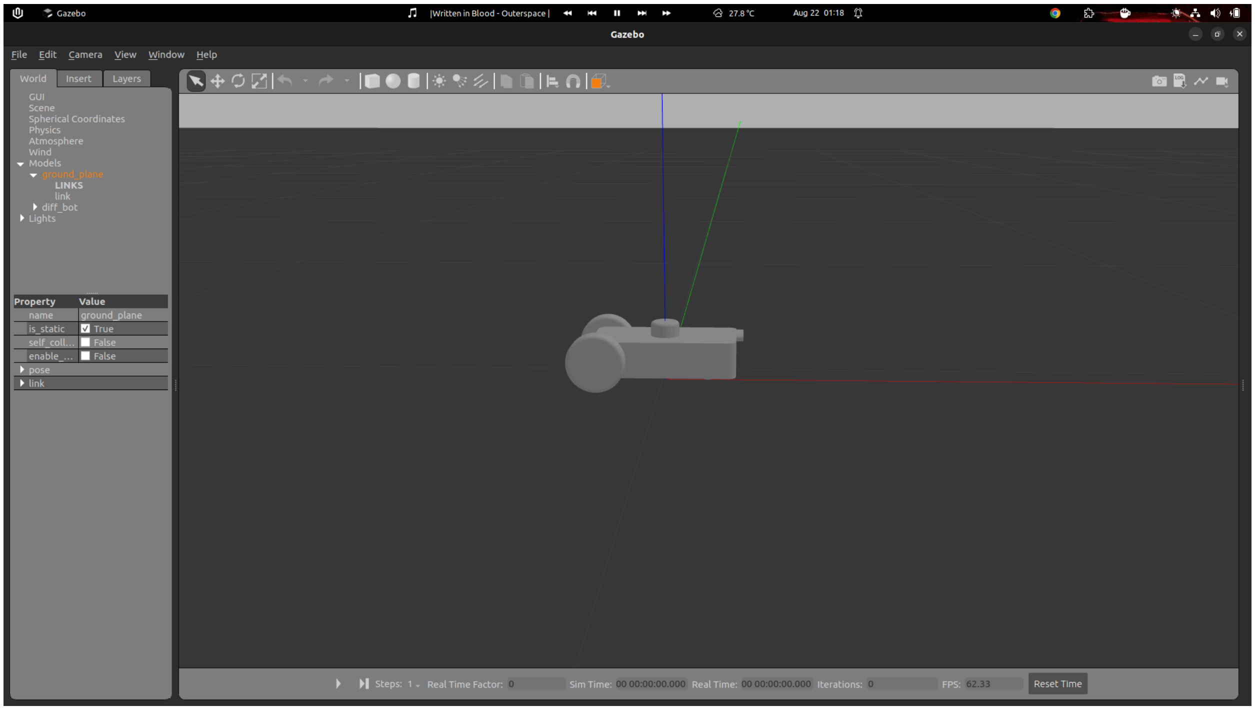
- Source your workspace by passing:

```
source install/setup.bash
```

- You can now visualize your robot in gazebo by launching the existing launch file by running the command:

```
ros2 launch diff_bot_description gazebo.launch
```

You should see the following output on your screen:



Initially, the gazebo environment will be paused, which you can see by the pause sign at the bottom left corner. Press it to start the simulation, and just check if all the connections in your model are proper and nothing is flying anywhere (a very common problem when exporting to URDF, trust me).

Once you have verified everything, let's add the cameras and other sensors to your robot model and get some data from that. Also, about time we start moving the robot around!

Controllers and Sensors

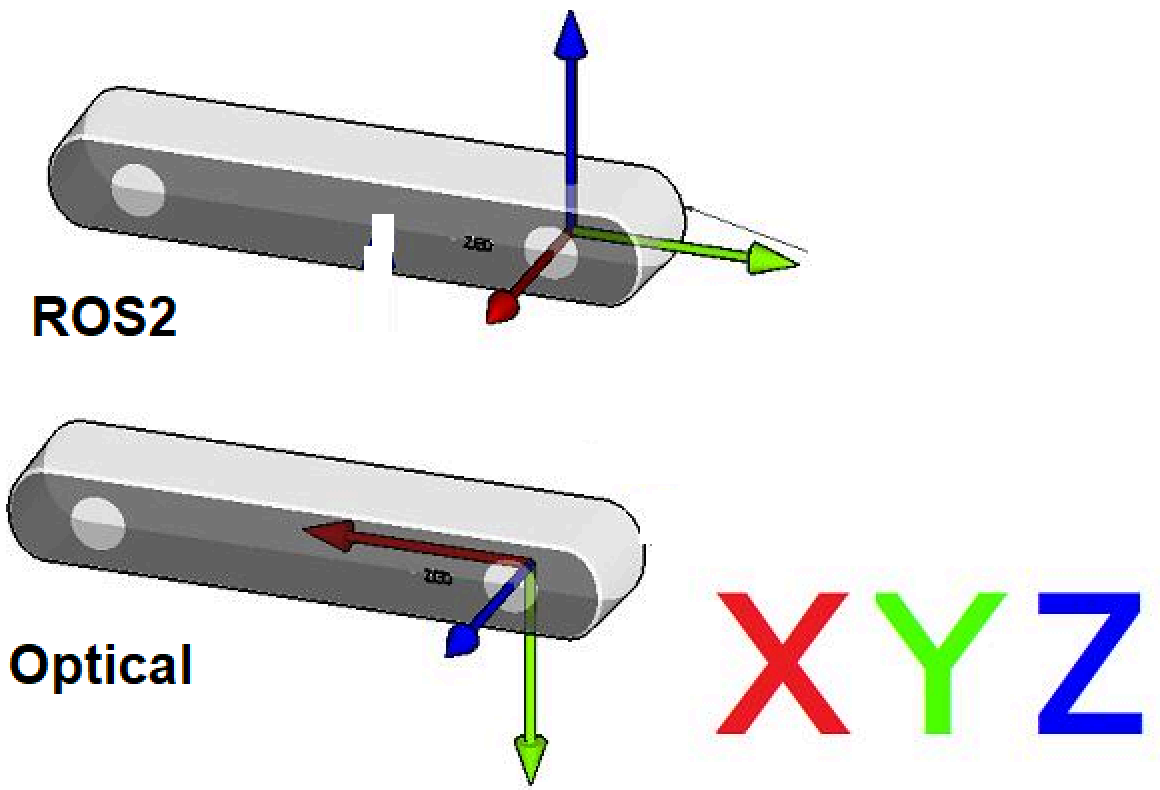
Let's start with the most basic of sensors, a camera. Since you already added a camera model and attached it to your robot, we need to tell the gazebo to treat the camera link as a camera. To do so, make the following edits in your .gazebo file, which you will find in the urdf directory of your robot description package. Since we will be also using the same camera link as a depth camera, we can import the depth camera gazebo plugin directly, which publishes the image and the point clouds.

```xml
<!-- CAMERA SETUP -->
<gazebo reference="camera_link">
       <material>${body_color}</material>

       <sensor name="camera" type="depth">
           <pose> 0 0 0 0 0 0 </pose>
           <visualize>true</visualize>
           <update_rate>10</update_rate>
           <camera>
               <horizontal_fov>1.089</horizontal_fov>
               <image>
                   <format>B8G8R8</format>
                   <width>640</width>
                   <height>480</height>
               </image>
               <clip>
                   <near>0.05</near>
                   <far>8.0</far>
               </clip>
           </camera>
           <plugin name="camera_controller"
filename="libgazebo_ros_camera.so">
               <frame_name>camera_link_optical</frame_name>
               <min_depth>0.1</min_depth>
               <max_depth>100.0</max_depth>
           </plugin>
       </sensor>
   </gazebo>
```
.

Notice how we have added camera_link_optical as the frame name instead of the actual camera link name.

The reason for this is that ROS2 and ROS in general, use the following convention:
X: Forward, Y: Left, Z: Up
However, almost all computer vision applications use the following optical frame convention:
X: Right, Y: Down, Z: Forward

Refer to the following image (https://github.com/IntelRealSense/realsense-ros)



Hence we add a new link to our URDF with no inertial or mass properties called camera_link_optical, and attach it to the original camera link by rotating it in negative 90 degrees along the x and z axis.

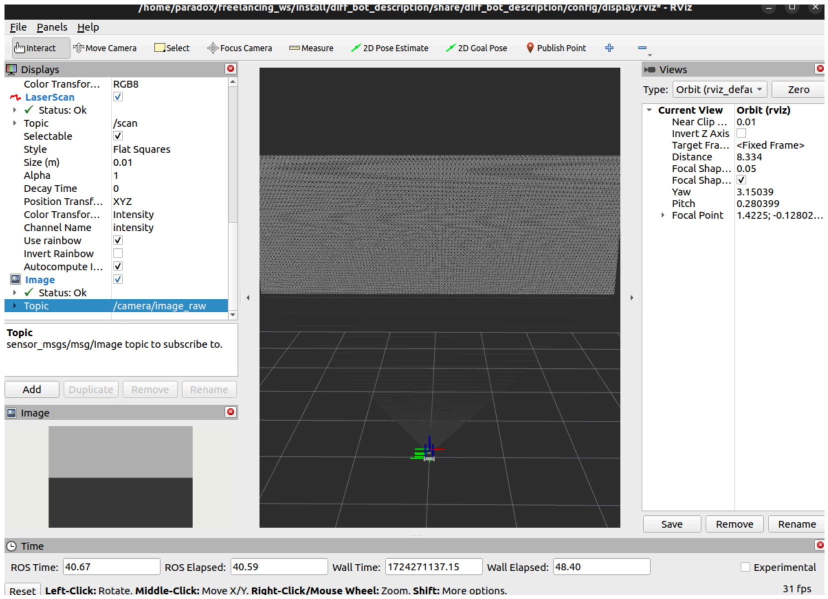
Add the following code to your .xacro file, which has all your link properties and joints information.

```
<joint name="camera_optical_joint" type="fixed">
   <origin xyz="0 0 0" rpy="${-pi/2} 0 ${-pi/2}" />
   <parent link="camera_1" />
   <child link="camera_link_optical" />
</joint>

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

Once you’ve added this, you are done with your camera setup! Build your package again and then launch the gazebo.launch file once more. Open RVIZ2, by passing the following command in the terminal:

```
rviz2
```

and then add Image and PointCloud2 display plugins. RVIZ itself will recommend specific topics. You should see a display screen like so:



The image output and the point cloud output should be pointing in front of the robot. The output feels a little empty no? We will be adding an environment to brighten up the robot's surroundings.

For this, you will be making slight changes to your gazebo.launch file. We will add a prebuilt environment that comes with turtlebot3. Check this link for reference: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/

Add the destination to the world file like so:

```
world = os.path.join(
       get_package_share_directory('turtlebot3_gazebo'),
       'worlds',
       'turtlebot3_house.world')
```

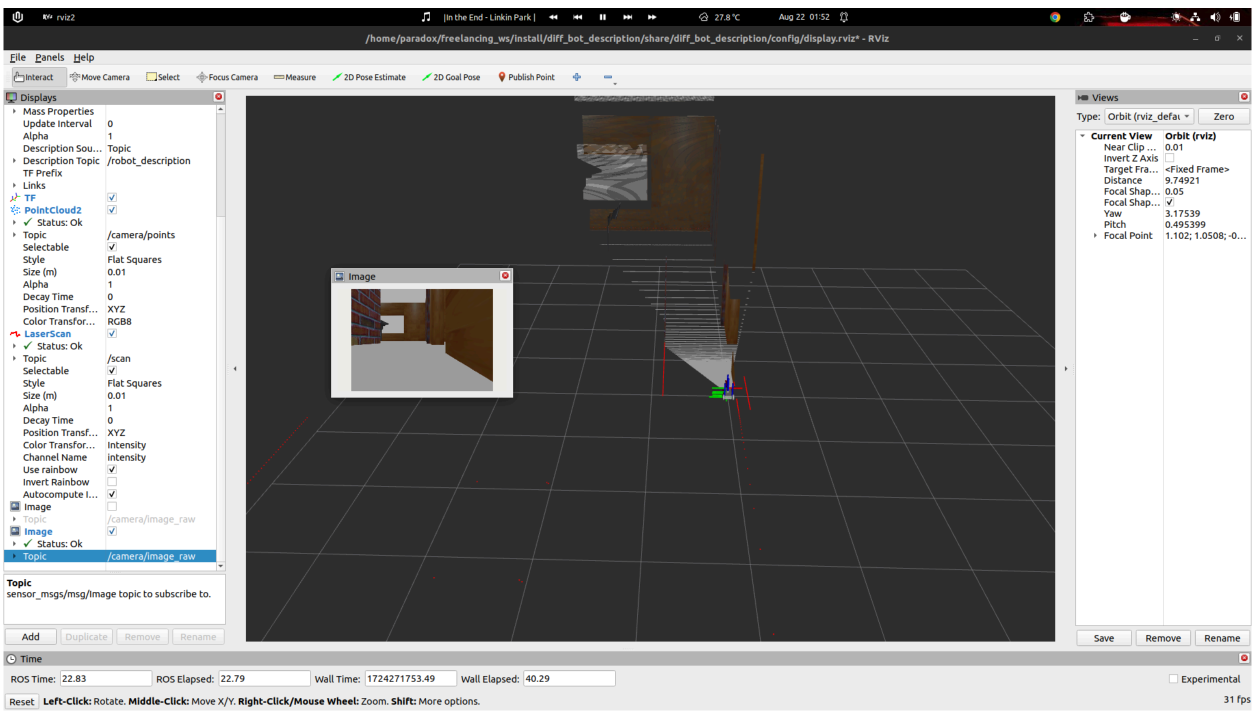
and add this world file as an argument to your gazebo server node like so:

```
gazebo_server = IncludeLaunchDescription(
       PythonLaunchDescriptionSource([
           PathJoinSubstitution([
               FindPackageShare('gazebo_ros'),
```

```
                'launch',
                'gzserver.launch.py'
            ])
        ]),
        launch_arguments={
            'pause': 'true',
            'world': world,
        }.items()
)
```

Here you can also see the argument pause:true, you can change it to false if you get tired of pressing the play button again and again.

Let's build the package and see the output once more.



Better no?
And if you feel the camera image output is too small, you can change the image height and width in the image tags like so:

```
<image>
<format>B8G8R8</format>
```

```
<width>640</width>
<height>480</height>
</image>
```

Let's start with the IMU sensor then. Same as with the camera, we will need to tell Gazebo to import the imu sensor plugin and refer to a specific link as a reference. Add the following code in the .gazebo file of your urdf directory like so:

```
<gazebo reference="imu_link">
 <gravity>true</gravity>
 <sensor name="imu_sensor" type="imu">
   <always_on>true</always_on>
   <update_rate>100</update_rate>
   <visualize>true</visualize>
   <topic>__default_topic__</topic>
   <plugin filename="libgazebo_ros_imu_sensor.so" name="imu_plugin">
     <ros>
         <remapping>~/out:=imu_data</remapping>
     </ros>
     <topicName>imu</topicName>
     <bodyName>imu_link</bodyName>
     <updateRateHZ>10.0</updateRateHZ>
     <gaussianNoise>0.0</gaussianNoise>
     <xyzOffset>0 0 0</xyzOffset>
     <rpyOffset>0 0 0</rpyOffset>
     <frameName>imu_1</frameName>

<initial_orientation_as_reference>false</initial_orientation_as_reference>
   </plugin>
   <pose>0 0 0 0 0 0</pose>
 </sensor>
</gazebo>
```

You don't really need to touch any code blocks here except for the offset parts and the remapping part. Remap to whatever imu topic name you are comfortable with and set the xyz, rpy offsets according to your robot design.

Finally, let’s add a lidar and complete the holy trinity of sensor configuration!

To add the lidar, same as before add the following code to import the lidar sensor plugin:

```
<gazebo reference="lidar_link">
   <sensor name="laser" type="ray">
       <pose> 0 0 0 0 0 0 </pose>
       <visualize>true</visualize>
       <update_rate>30</update_rate>
       <ray>
           <scan>
               <horizontal>
                   <samples>720</samples>
                   <min_angle>-3.14</min_angle>
                   <max_angle>3.14</max_angle>
               </horizontal>
           </scan>
           <range>
               <min>0.2</min>
               <max>8</max>
           </range>
       </ray>
       <plugin name="laser_controller"
filename="libgazebo_ros_ray_sensor.so">
           <ros>
               <remapping>~/out:=scan</remapping>
           </ros>
           <output_type>sensor_msgs/LaserScan</output_type>
           <frame_name>lidar_link</frame_name>
       </plugin>
   </sensor>
</gazebo>
```

No need to touch anything here as well, other than the min-max ranges, set it according to your requirements. You can set the minimum range such that the interferences by the robot design to the lidar (if any) are ignored. Also you can set the remapping output topic to your preferences.

Please note, I added the generic code blocks, so wherever you see <sensor name>_link, you need to add the actual link name that you are using in your description package. For actual implementation, you can refer to the code provided along with.

Build your package and check if you are getting all the correct data from your sensors. You can echo the imu topic to make sure it's running by passing the following in the terminal:

```
ros2 topic echo /imu_data
```

Change the topic name according to your package.

Lidar and Camera, you can visualize in RVIZ to verify.

Let's move around the robot, shall we?

So for moving the robot around, what we basically want to achieve is to send commands via keyboard/joystick, and the robot responds according to that. So how do we do that?

Welcome to the stage, ros2_control.

I won't go into too many details on how it works, the official documentation (https://control.ros.org/rolling/index.html) covers it pretty extensively, and there are some good YouTube tutorials out there (https://www.youtube.com/watch?v=4QKsDf1c4hc, Articulated robotics, my fave), and to be honest, we are here to make it work not learn.

So, the basic understanding you need to have is, if you are controlling a robot via ros2_control (preferred because we can port it to a real robot as well), you need to have hardware(mock hardware that is simulated by gazebo) controlled by some controllers . Put the following code in the .xacro file of your urdf directory:

```
<!--  differential controller setup -->
<ros2_control name="GazeboSystem" type="system">
 <hardware>
   <plugin>gazebo_ros2_control/GazeboSystem</plugin>
 </hardware>
 <joint name="left_joint">
   <command_interface name="velocity"/>
   <state_interface name="velocity"/>
   <state_interface name="position"/>
 </joint>
 <joint name="right_joint">
```

```
    <command_interface name="velocity"/>
    <state_interface name="velocity"/>
    <state_interface name="position"/>
  </joint>
</ros2_control>
```

In the hardware tags of this code, you can see we are calling a gazebo plugin that can simulate the hardware. After that, you can see the joint tags which have command and state interfaces. This is how ros2_control will control the joints, via these interfaces. Since we are using a 2 wheel differential, we need only add our two continuous joints, however, if you use a four-wheel robot, you need to mention all the joints.

After this comes another gazebo plugin:

```
<gazebo>
  <plugin filename="libgazebo_ros2_control.so" name="gazebo_ros2_control">
  <!-- <robot_param>robot_description</robot_param>
      <robot_param_node>robot_state_publisher</robot_param_node> -->
    <parameters>$(find
diff_bot_description)/config/diff_controller.yaml</parameters>
  </plugin>
</gazebo>
```

This plugin basically integrates the gazebo sim with the ros2_control. You can see in the parameter tags, there's a file called diff_controller.yaml. This does not come with the urdf exporter tool but is something you need to add yourself. Make a new .yaml file with the following and add it to the config directory.

```
controller_manager:
  ros__parameters:
    update_rate: 30
    use_sim_time: true

    joint_broad:
      type: joint_state_broadcaster/JointStateBroadcaster

    diff_controller:
      type: differential_bot_controller/DifferentialBotController
```

```yaml
diff_controller:
 ros__parameters:
   left_wheel_names: ["left_joint",] #These are JOINT names, not WHEEL
names(fucking confusing, ik)
   right_wheel_names: ["right_joint",]
   wheel_separation: 0.2
   # wheels_per_side: 1 # actually 2, but both are controlled by 1 signal
   wheel_radius: 0.075

   wheel_separation_multiplier: 1.0
   left_wheel_radius_multiplier: 1.0
   right_wheel_radius_multiplier: 1.0

   publish_rate: 50.0
   odom_frame_id: odom
   base_frame_id: base_footprint
   pose_covariance_diagonal : [0.001, 0.001, 0.001, 0.001, 0.001, 0.01]
   twist_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.01]

   open_loop: false
   enable_odom_tf: false

   cmd_vel_timeout: 0.5
   #publish_limited_velocity: true
   use_stamped_vel: false
   #velocity_rolling_window_size: 10

   # Velocity and acceleration limits
   # Whenever a min_* is unspecified, default to -max_*
   linear.x.has_velocity_limits: true
   linear.x.has_acceleration_limits: true
   linear.x.has_jerk_limits: false
   linear.x.max_velocity: 10.0
   linear.x.min_velocity: -10.0
   linear.x.max_acceleration: 10.0
   linear.x.max_jerk: 0.0
   linear.x.min_jerk: 0.0

   angular.z.has_velocity_limits: true
   angular.z.has_acceleration_limits: true
```

```
  angular.z.has_jerk_limits: false
  angular.z.max_velocity: 50.0
  angular.z.min_velocity: -50.0
  angular.z.max_acceleration: 5.0
  angular.z.min_acceleration: -5.0
  angular.z.max_jerk: 0.0
  angular.z.min_jerk: 0.0
```

These are basically the parameters you put for your robot that are used by the controller (either yours or controllers already developed by ros2 devs)

The joint broadcaster is pretty simple cause as the name suggests, it broadcasts the joint states. However the second controller, diff_controller, is of the type: differential_bot_controller/DifferentialBotController

This is basically a second package that you can either develop yourself or use the already pre-built packages that come with ros2 according to your robot configuration. Check https://github.com/ros-controls/ros2_controllers/tree/master to see the number of controllers already developed.

For the purposes of demonstration, I made a second package in the robot workspace called differential_bot_controller, which has all the source files for controlling the individual joints of the robot and is **heavily** influenced by the prebuilt diff_drive_controller that comes with the ros2 packages. Just wanted to show you how it's done. To understand how to make your own ros2_control, you can check the following link: https://github.com/ros-controls/ros2_control_demos

After all is done, some final changes are required for your gazebo.launch file, before you can start playing around with the robot.

Add the following code to your launch file:

```
diff_drive_spawner = Node(
       package='controller_manager',
       executable='spawner',
       arguments=['diff_controller', '--controller-manager',
'/controller_manager'],
       parameters=[
           {'use_sim_time': use_sim_time},
```

```
        ],
    )

    joint_broad_spawner = Node(
        package='controller_manager',
        executable='spawner',
        parameters=[
            {'use_sim_time': use_sim_time},
        ],
        arguments=['joint_broad', '--controller-manager',
'/controller_manager'],
    )
```

Edit the return LaunchDescription section to add these two new nodes as well. Finally, you need to make some changes in the friction settings of your wheels, by changing the friction of the castor wheel to zero so that it doesn't constrain the wheels and acts like a real castor wheel and increase the friction of wheels so you can get more traction on the environment, for better motion.

```
<gazebo reference="castor_wheel">
 <material>${body_color}</material>
 <mu1>0.0</mu1>
 <mu2>0.0</mu2>
 <self_collide>true</self_collide>
</gazebo>
<gazebo reference="right_wheel_1">
 <material>${body_color}</material>
 <mu1>5.0</mu1>
 <mu2>5.0</mu2>
 <self_collide>true</self_collide>
</gazebo>

<gazebo reference="left_wheel_1">
 <material>${body_color}</material>
 <mu1>5.0</mu1>
 <mu2>5.0</mu2>
 <self_collide>true</self_collide>
</gazebo>
```

mu1 is friction along x and mu2 is friction along y, change the parameters until you get a motion you are satisfied by.

Build the workspace and launch the file. A successful launch file will have the following messages waiting for you in the terminal:

```
[spawner-2] [INFO] [1724271803.284999905] [spawner_diff_controller]:
Loaded diff_controller
[gzserver-3] [INFO] [1724271803.291614710] [controller_manager]:
Loading controller 'joint_broad'
[gzserver-3] [INFO] [1724271803.346071474] [controller_manager]:
Configuring controller 'diff_controller'
[spawner-1] [INFO] [1724271803.348996128] [spawner_joint_broad]:
Loaded joint_broad
[gzserver-3] [INFO] [1724271803.459740214] [controller_manager]:
Configuring controller 'joint_broad'
[gzserver-3] [INFO] [1724271803.460078937] [joint_broad]: 'joints' or
'interfaces' parameter is empty. All available state interfaces will
be published
[spawner-2] [INFO] [1724271813.274221352] [spawner_diff_controller]:
Configured and activated diff_controller
[spawner-1] [INFO] [1724271813.367207723] [spawner_joint_broad]:
Configured and activated joint_broad
```

If you do list out the topics, you can see a cmd_vel-like topic mentioned in the terminal like so:

```
/diff_controller/cmd_vel_unstamped
```

This is the topic that basically takes the linear and angular commands and passes it to the differential controller. To move the robot, you can try running the following in a new terminal:

```
ros2 run teleop_twist_keyboard teleop_twist_keyboard --remap
/cmd_vel:=/diff_controller/cmd_vel_unstamped
```

Try pressing the keyboard keys like the terminal output shows and see if the robot is moving around.

If yes, Congratulations! You are one step closer to making your robot move autonomously!!

If not, please check the topic listings and terminal outputs from your launch files and see if you can pinpoint the error. But don't worry, you will be able to get it!

It's been a long read, but we are almost there. Let's try to localize the robot so that we can view the robot moving in RVIZ, which is important for autonomous navigation.

Localization

So, for localization, we can use the EKF node that comes with the ros2 packages to get the odom topic and even the transformation from odom to base_footprint. The documentation and the comments on it are pretty easy to follow. You just need to add the topics from which you'd like the linear, angular, and velocity data. For example, you can fuse the robot distance moved according to the wheel encoder data, angular motion via the imu, and the laser scan data to get an odom estimate. However, we still won't get an accurate odom estimate in simulation. You can see the implementation in the code for your own reference, but I won't recommend to use that.

The ekf.yaml file that is in the config folder has extensive comments explaining everything. You just need to add the following in your launch file:

```
robot_localization = Node(
   package='robot_localization',
   executable='ekf_node',
   name='ekf_filter_node',
   output='screen',
   parameters=[robot_localization_file_path,
   {'use_sim_time': use_sim_time}])
```

Add the node to your return LaunchDescription parameter at the end to launch the node. The robot_localization_file_path is basically the file path to the ekf.yaml file.

So how to get the accurate odometry data I hear you ask?

The best way is the p3d plugin that gazebo provides, which sends the robot's position with respect to the gazebo world. Add the following in your .gazebo file:

```xml
<gazebo>
   <plugin name="libgazebo_ros_p3d" filename="libgazebo_ros_p3d.so">
   <topicName>odom</topicName>
       <ros>
           <namespace>/p3d_robot</namespace>
           <remapping>odom:=odom</remapping>
           </ros>
           <frame_name>world</frame_name>
           <body_name>base_footprint</body_name>
           <update_rate>50.0</update_rate>
           <gaussian_noise>0.01</gaussian_noise>
       </plugin>
     </gazebo>
```

This will output the odom data on the topic /p3d_robot/odom.

However, one downside to this is that you won't have the transformation from odom to base_footprint so you cannot visualize it in RVIZ….yet.

We can make our own node for transforming frames. You can make a new python file in your package_name/package_name folder. Add the following code:

```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import TransformStamped
from nav_msgs.msg import Odometry
import tf2_ros
class OdomToTransform(Node):

   def __init__(self):
       super().__init__('odom_to_transform_publisher')
       self.tf_broadcaster = tf2_ros.TransformBroadcaster(self)

       self.odom_subscriber = self.create_subscription(
           Odometry,
           '/p3d_robot/odom',
           self.odometry_callback,
           10
       )
       self.timer_period = 0.1  # Set the publishing rate (in seconds)
```

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

    def odometry_callback(self, msg):
        self.last_odom_msg = msg

    def publish_odom_transform(self):
        try:
            odom_msg = self.last_odom_msg  # Get the latest stored odometry
message
        except AttributeError:
            return  # No odometry message received yet

        odom_to_base_transform = TransformStamped()
        odom_to_base_transform.header.stamp = odom_msg.header.stamp
        odom_to_base_transform.header.frame_id = 'odom'
        odom_to_base_transform.child_frame_id = 'base_footprint'

        # Set the translation from odometry data
        odom_to_base_transform.transform.translation.x =
odom_msg.pose.pose.position.x
        odom_to_base_transform.transform.translation.y =
odom_msg.pose.pose.position.y
        odom_to_base_transform.transform.translation.z =
odom_msg.pose.pose.position.z

        # Set the rotation from odometry data
        odom_to_base_transform.transform.rotation =
odom_msg.pose.pose.orientation

        # Publish the transform
        self.tf_broadcaster.sendTransform(odom_to_base_transform)



def main(args=None):
    rclpy.init(args=args)
    odom_to_transform_publisher = OdomToTransform()
    rclpy.spin(odom_to_transform_publisher)
    rclpy.shutdown()
```

```
if __name__ == '__main__':
    main()
```

This code basically subscribes to the p3d_robot/odom topic and publishes the transform from odom frame to base_footprint frame according to the odom data. Save the file, and in your setup.py file in the same package, add this file as a node by passing the following lines in the entry points section:

```
entry_points={
        'console_scripts': ['odom_transform =
diff_bot_description.odom_tf:main',
        ],
    },
```

The line “odom_transform = diff_bot_description.odom_tf:main” is basically node_name = package_name:python_file_name:main.

Basically telling ros2 to refer to that specific python file in that specific package whenever the ros2 run package_name node_name is invoked.

Finally, in your launch file, add the node like so:

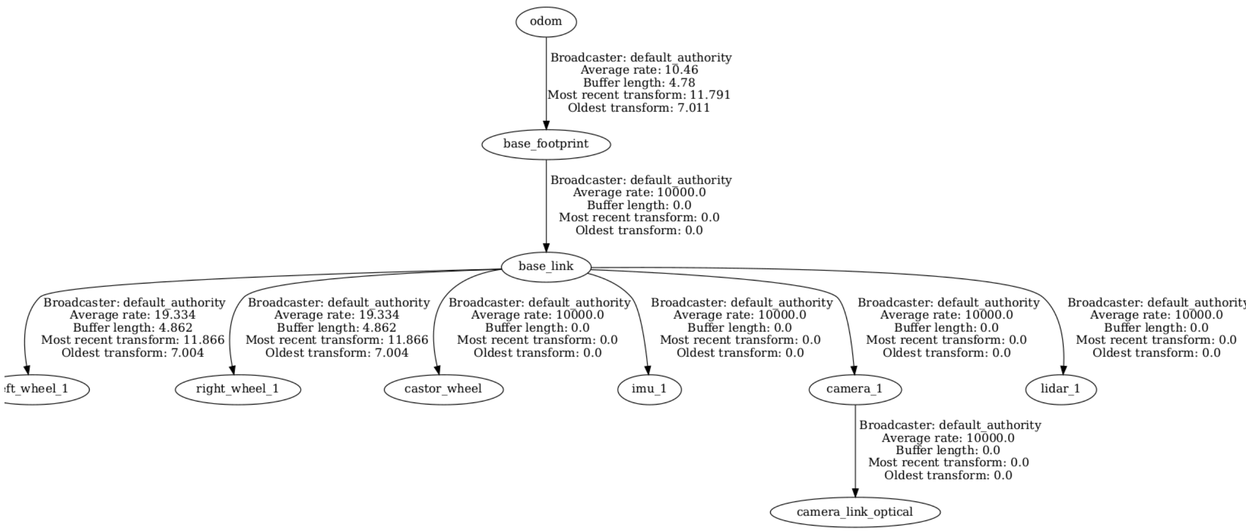
```
odom_node = Node(
        package='diff_bot_description',
        executable='odom_transform',
        name = 'odom_transformer'
    )
```

Add the node in the return LaunchDescription section, save all the files, and build the workspace.

Finally, launch your gazebo file and try visualizing the robot in rviz and see the difference for yourself between ekf and p3d. Your tf tree, which you can get by running:

```
ros2 run tf2_tools view_frames
```

Should look like this:



Congratulations, you have a working robot for your reference now!!!

From here on out, these are some extra stuff I added that maybe you can use, or try out to get a better idea of ROS2. Test it out for yourself, and **learn**!

Topic Remappings

Finally, we can add separate packages in Python and C++, to remap sensor topics.

To create a python package, cd into the src folder of you workspace, and run the following command:

```
ros2 pkg create --build-type ament_python republisher_py
--dependencies rclpy sensor_msgs
```

For C++, run:

```
ros2 pkg create --build-type ament_cmake republisher_cpp
--dependencies rclcpp sensor_msgs
```

Now, we just need to make a node in python and C++ that subscribes to all the sensor data and republishes them.

For python, in the folder republisher_py/republisher_py, make a new file called republisher.py and add the following code:

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan, Imu, Image, PointCloud2

class RepublisherNode(Node):
   def __init__(self):
       super().__init__('republisher_node')

       # Subscriptions
       self.scan_subscriber = self.create_subscription(LaserScan, '/scan',
self.scan_callback, 10)
       self.imu_subscriber = self.create_subscription(Imu, '/imu_data',
self.imu_callback, 10)
       self.image_subscriber = self.create_subscription(Image,
'/camera/image_raw', self.image_callback, 10)
       self.points_subscriber = self.create_subscription(PointCloud2,
'/camera/points', self.points_callback, 10)

       # Publishers
       self.scan_publisher = self.create_publisher(LaserScan,
'/republished/scan', 10)
       self.imu_publisher = self.create_publisher(Imu,
'/republished/imu_data', 10)
       self.image_publisher = self.create_publisher(Image,
'/republished/image_raw', 10)
       self.points_publisher = self.create_publisher(PointCloud2,
'/republished/points', 10)

   def scan_callback(self, msg):
```

```
        self.scan_publisher.publish(msg)
        self.get_logger().info('Republished /scan data')

    def imu_callback(self, msg):
        self.imu_publisher.publish(msg)
        self.get_logger().info('Republished /imu_data')

    def image_callback(self, msg):
        self.image_publisher.publish(msg)
        self.get_logger().info('Republished /camera/image_raw')

    def points_callback(self, msg):
        self.points_publisher.publish(msg)
        self.get_logger().info('Republished /camera/points')


def main(args=None):
    rclpy.init(args=args)
    node = RepublisherNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Add the node in your entry points as described above.

For making a C++ node, in your src folder of the republisher_cpp directory, make a new file republisher.cpp and add the following code:

```
#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/laser_scan.hpp"
#include "sensor_msgs/msg/imu.hpp"
#include "sensor_msgs/msg/image.hpp"
#include "sensor_msgs/msg/point_cloud2.hpp"

class RepublisherNode : public rclcpp::Node
```

```
{
public:
   RepublisherNode() : Node("republisher_node")
   {
       // Subscriptions
       scan_subscriber_ =
this->create_subscription<sensor_msgs::msg::LaserScan>(
           "/scan", 10, std::bind(&RepublisherNode::scan_callback, this,
std::placeholders::_1));
       imu_subscriber_ = this->create_subscription<sensor_msgs::msg::Imu>(
           "/imu_data", 10, std::bind(&RepublisherNode::imu_callback,
this, std::placeholders::_1));
       image_subscriber_ =
this->create_subscription<sensor_msgs::msg::Image>(
           "/camera/image_raw", 10,
std::bind(&RepublisherNode::image_callback, this, std::placeholders::_1));
       points_subscriber_ =
this->create_subscription<sensor_msgs::msg::PointCloud2>(
           "/camera/points", 10,
std::bind(&RepublisherNode::points_callback, this,
std::placeholders::_1));

       // Publishers
       scan_publisher_ =
this->create_publisher<sensor_msgs::msg::LaserScan>("/republished/scan",
10);
       imu_publisher_ =
this->create_publisher<sensor_msgs::msg::Imu>("/republished/imu_data",
10);
       image_publisher_ =
this->create_publisher<sensor_msgs::msg::Image>("/republished/image_raw",
10);
       points_publisher_ =
this->create_publisher<sensor_msgs::msg::PointCloud2>("/republished/points
", 10);
   }

private:
   void scan_callback(const sensor_msgs::msg::LaserScan::SharedPtr msg)
   {
```

```
        scan_publisher_->publish(*msg);
        RCLCPP_INFO(this->get_logger(), "Republished /scan data");
    }

    void imu_callback(const sensor_msgs::msg::Imu::SharedPtr msg)
    {
        imu_publisher_->publish(*msg);
        RCLCPP_INFO(this->get_logger(), "Republished /imu_data");
    }

    void image_callback(const sensor_msgs::msg::Image::SharedPtr msg)
    {
        image_publisher_->publish(*msg);
        RCLCPP_INFO(this->get_logger(), "Republished /camera/image_raw");
    }

    void points_callback(const sensor_msgs::msg::PointCloud2::SharedPtr
msg)
    {
        points_publisher_->publish(*msg);
        RCLCPP_INFO(this->get_logger(), "Republished /camera/points");
    }

    rclcpp::Subscription<sensor_msgs::msg::LaserScan>::SharedPtr
scan_subscriber_;
    rclcpp::Subscription<sensor_msgs::msg::Imu>::SharedPtr imu_subscriber_;
    rclcpp::Subscription<sensor_msgs::msg::Image>::SharedPtr
image_subscriber_;
    rclcpp::Subscription<sensor_msgs::msg::PointCloud2>::SharedPtr
points_subscriber_;

    rclcpp::Publisher<sensor_msgs::msg::LaserScan>::SharedPtr
scan_publisher_;
    rclcpp::Publisher<sensor_msgs::msg::Imu>::SharedPtr imu_publisher_;
    rclcpp::Publisher<sensor_msgs::msg::Image>::SharedPtr image_publisher_;
    rclcpp::Publisher<sensor_msgs::msg::PointCloud2>::SharedPtr
points_publisher_;
};

int main(int argc, char **argv)
```

```
{
   rclcpp::init(argc, argv);
   auto node = std::make_shared<RepublisherNode>();
   rclcpp::spin(node);
   rclcpp::shutdown();
   return 0;
}
```

Finally, in your CMakeLists.txt file, add the following new configurations,

```
add_executable(republisher_c src/republisher.cpp)
ament_target_dependencies(republisher_c rclcpp sensor_msgs)

install(TARGETS
republisher_c
 DESTINATION lib/${PROJECT_NAME}
)
```

Where republisher_c is the node name.

Finally, build your workspace and you can launch the gazebo.launch file, and then either run the remapping node via python or C++ package by simply running:

```
ros2 run republisher_cpp republisher_c
```

When you list out your topics, you should now see the republished topics.

Trajectory visualisation

To visualize the trajectory taken by the robot, we can use the marker topic in ros2. To make the code, make a new Python file in your package, like you did for the republisher nodes, and then put the following code into it:

```
import rclpy
from rclpy.node import Node
from nav_msgs.msg import Odometry
from visualization_msgs.msg import Marker
from geometry_msgs.msg import Point
```

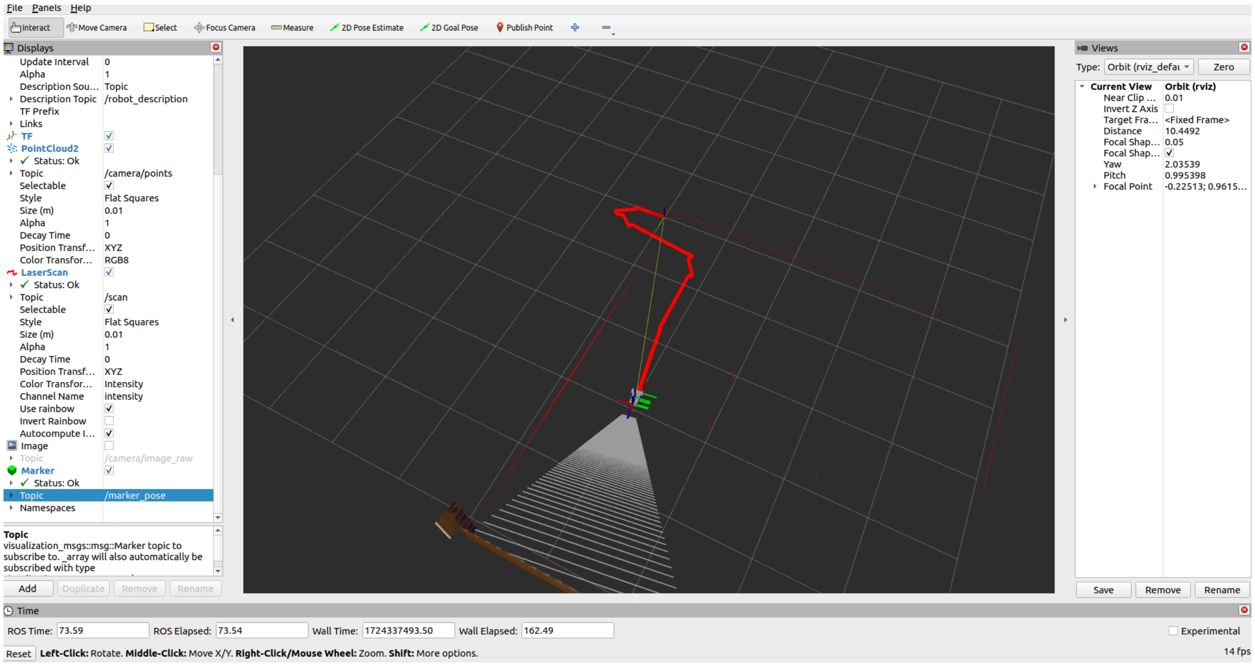
```python
class PathVisualizer(Node):
   def __init__(self):
       super().__init__('path_visualizer')

       # Subscribe to the /odom topic
       self.odom_subscriber = self.create_subscription(
           Odometry,
           '/p3d_robot/odom',
           self.odom_callback,
           10
       )

       # Publisher for the marker topic
       self.marker_publisher = self.create_publisher(Marker,
'/marker_poses', 10)

       # Initialize a Marker object for the path
       self.path_marker = Marker()
       self.path_marker.header.frame_id = "odom"
       self.path_marker.type = Marker.LINE_STRIP
       self.path_marker.action = Marker.ADD
       self.path_marker.scale.x = 0.05  # Line width
       self.path_marker.color.a = 1.0  # Alpha (transparency)
       self.path_marker.color.r = 1.0  # Red color
       self.path_marker.color.g = 0.0  # Green color
       self.path_marker.color.b = 0.0  # Blue color

   def odom_callback(self, msg):
       # Extract the position from the Odometry message
       position = msg.pose.pose.position

       # Create a Point from the position
       point = Point()
       point.x = position.x
       point.y = position.y
       point.z = position.z

       # Append the point to the path marker's points array
       self.path_marker.points.append(point)
```

```
        # Update the header timestamp
        self.path_marker.header.stamp = self.get_clock().now().to_msg()

        # Publish the marker
        self.marker_publisher.publish(self.path_marker)

def main(args=None):
    rclpy.init(args=args)
    node = PathVisualizer()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

After this, you can initialize the code as a node in your python package. To view the path in rviz now, you can add a marker topic. You won't see the suggested topic for it cause we did not run the node for publishing the marker topic. For that, just run:

```
ros2 run <package_name> node_name
```

Move the robot around and see the visualized trajectory!



Launch file

We will now look at how a launch file is structured:

```
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import
PythonLaunchDescriptionSource
from launch.substitutions import PathJoinSubstitution, LaunchConfiguration
import os
import xacro
from ament_index_python.packages import get_package_share_directory
```

These are all important imports that we need to do to work our launch file.

```
def generate_launch_description():
   share_dir = get_package_share_directory('diff_bot_description')
   use_sim_time = LaunchConfiguration('use_sim_time', default='True')
   xacro_file = os.path.join(share_dir, 'urdf', 'diff_bot.xacro')
```

```
    robot_localization_file_path = os.path.join(share_dir, 'config',
'ekf.yaml')
    robot_description_config = xacro.process_file(xacro_file)
    robot_urdf = robot_description_config.toxml()
    rviz_config_dir = os.path.join(share_dir, 'config', 'display.rviz')
    world = os.path.join(
        get_package_share_directory('turtlebot3_gazebo'),
        'worlds',
        'turtlebot3_house.world')

    robot_state_publisher_node = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        parameters=[
            {'robot_description': robot_urdf},
            {'use_sim_time': use_sim_time},
        ]
    )

    joint_state_publisher_node = Node(
        package='joint_state_publisher',
        executable='joint_state_publisher',
        name='joint_state_publisher'
    )

    gazebo_server = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([
            PathJoinSubstitution([
                FindPackageShare('gazebo_ros'),
                'launch',
                'gzserver.launch.py'
            ])
        ]),
        launch_arguments={
            'pause': 'true',
            'world': world,
        }.items()
    )
```

```
    gazebo_client = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([
            PathJoinSubstitution([
                FindPackageShare('gazebo_ros'),
                'launch',
                'gzclient.launch.py'
            ])
        ])
    )


    diff_drive_spawner = Node(
        package='controller_manager',
        executable='spawner',
        arguments=['diff_controller', '--controller-manager',
'/controller_manager'],
        parameters=[
            {'use_sim_time': use_sim_time},
        ],
    )

    joint_broad_spawner = Node(
        package='controller_manager',
        executable='spawner',
        parameters=[
            {'use_sim_time': use_sim_time},
        ],
        arguments=['joint_broad', '--controller-manager',
'/controller_manager'],
    )

    urdf_spawn_node = Node(
        package='gazebo_ros',
        executable='spawn_entity.py',
        arguments=[
            '-entity', 'diff_bot',
            '-topic', 'robot_description'
        ],
        output='screen'
    )
```

```
rviz_node = Node(
    package='rviz2',
    executable='rviz2',
    name='rviz2',
    output='screen',
    arguments=['-d', rviz_config_dir],
    parameters=[
        {'use_sim_time': use_sim_time},

    ],
)

odom_node = Node(
    package='diff_bot_description',
    executable='odom_transform',
    name = 'odom_transformer'
)

robot_localization = Node(
package='robot_localization',
executable='ekf_node',
name='ekf_filter_node',
output='screen',
parameters=[robot_localization_file_path,
{'use_sim_time': use_sim_time}])
```

In the generate launch description function, you basically add nodes and pass arguments about the said nodes to make a description of the nodes.

```
share_dir = get_package_share_directory('diff_bot_description')
   use_sim_time = LaunchConfiguration('use_sim_time', default='True')
   xacro_file = os.path.join(share_dir, 'urdf', 'diff_bot.xacro')
   robot_localization_file_path = os.path.join(share_dir, 'config',
'ekf.yaml')
   robot_description_config = xacro.process_file(xacro_file)
   robot_urdf = robot_description_config.toxml()
   rviz_config_dir = os.path.join(share_dir, 'config', 'display.rviz')
   world = os.path.join(
```

```
        get_package_share_directory('turtlebot3_gazebo'),
        'worlds',
        'turtlebot3_house.world')
```

These lines are basically variables that store the location of the config files, rviz files, and world files.

```
use_sim_time = LaunchConfiguration('use_sim_time', default='True')
```

This line stores the argument, which can be used throughout. use_sim_time basically means to keep the time of all the nodes, wherever you specify it as the gazebo time. Since we are running a simulation, we set it as true.

```
robot_state_publisher_node = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        parameters=[
            {'robot_description': robot_urdf},
            {'use_sim_time': use_sim_time},
        ]
   )

   joint_state_publisher_node = Node(
        package='joint_state_publisher',
        executable='joint_state_publisher',
        name='joint_state_publisher'
   )
```

These both nodes are required to publish the robot states and joint states, that is the transformation between each link specified in the URDF. However, we are not using the joint_state publisher because we have a node publishing the joint states by the ros2_controllers.

```
 gazebo_server = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([
            PathJoinSubstitution([
                FindPackageShare('gazebo_ros'),
                'launch',
```

```
                'gzserver.launch.py'
            ])
        ]),
        launch_arguments={
            'pause': 'true',
            'world': world,
        }.items()
    )

    gazebo_client = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([
            PathJoinSubstitution([
                FindPackageShare('gazebo_ros'),
                'launch',
                'gzclient.launch.py'
            ])
        ])
    )
```

The gazebo_server and gazebo_client launches gazebo, with the launch arguments of the world file, and whether to pause the simulation while starting.

```
diff_drive_spawner = Node(
        package='controller_manager',
        executable='spawner',
        arguments=['diff_controller', '--controller-manager',
'/controller_manager'],
        parameters=[
            {'use_sim_time': use_sim_time},
        ],
    )

    joint_broad_spawner = Node(
        package='controller_manager',
        executable='spawner',
        parameters=[
            {'use_sim_time': use_sim_time},
        ],
        arguments=['joint_broad', '--controller-manager',
'/controller_manager'],
    )
```

The diff_drive spawner and the joint_broad spawner are both pointing out to the controller manager nodes, the diff_drive spawner starting the node for contorlloing the robot joints, and the joint_broad spawner, publishing the angles of joints. This is the node that is now handling all the joint publishing, hence why the previous joint node is not used.

```
rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        name='rviz2',
        output='screen',
        arguments=['-d', rviz_config_dir],
        parameters=[
            {'use_sim_time': use_sim_time},

        ],
   )

   odom_node = Node(
        package='diff_bot_description',
        executable='odom_transform',
        name = 'odom_transformer'
   )

   robot_localization = Node(
   package='robot_localization',
   executable='ekf_node',
   name='ekf_filter_node',
   output='screen',
   parameters=[robot_localization_file_path,
   {'use_sim_time': use_sim_time}])
```

The rviz_node, odom_node, and robot_localization node start rviz, the custom Odom publisher, and the localization package(which is again not used, but shown for demonstration purposes).

The node names can be anything, based on your preferences, just that the package name (the package containing the source code) and the executable (the node that we initialize the package to run), are non-negotiable. Those must have constant names based on the ros2_packages or your custom packages.

```
return LaunchDescription([
       joint_broad_spawner, #The node managed by controller_manager
       diff_drive_spawner, # Node for controlling the robot
       # joint_state_publisher_node, # Commented out because we already
have joint data by the joint_broad_spawner
       gazebo_server, #gazebo node
       gazebo_client, #gazebo node
       robot_state_publisher_node, #Transformation of robot links node
       urdf_spawn_node,
       odom_node, #Custom odom TF node
       rviz_node, #RVIZ
       # robot_localization # commented cause we have custom odom running
   ])
```

Finally, this is where we tell the launch file which nodes to launch when it is called.

Final Thoughts

If you've read this far, thanks a lot and I hope you liked it, even if just a little bit.

Some general points to keep in mind:

- Source your workspace!
- Make sure gazebo is sourced by running:
  . /usr/share/gazebo/setup.sh
  Or just add it in your .bashrc
- If you did all the steps correctly and still you have no results, check if you have the following ros2 packages installed:
  - ros2_control
  - ros2_controllers
  - gazebo_ros2_control
  - controller_manager
- No need to follow the naming conventions I did, you can do your own thing! I've figured this is the best way to invoke errors, and thus learn.