# ROBT 611 - ROS2 BASIC PRACTICE

Kanagat Kantoreyeva

***Abstract*—This assignment focuses on the completion of ROS 2 tutorial Session 3, which addresses motion control of manipulators through a series of structured exercises. The first two tasks involve enhancing the simulation environment by incorporating an additional object, such as a box, using URDF models sourced from a designated repository. In Exercise 3.2, a secondary service is developed to facilitate the transformation retrieval from the camera to the table, enhancing spatial awareness within the robotic framework. Exercise 3.4 necessitates the consideration of varied robot start and goal poses, necessitating the planning of a robot trajectory that incorporates collision avoidance strategies to navigate around the newly added object. Finally, Exercise 4.0 requires the execution of the C++ code to demonstrate the robot's motion, integrating obstacle avoidance techniques established in previous tasks. This assignment will reinforce the understanding of robotic motion planning and control within the ROS 2 ecosystem, emphasizing practical implementation and problem-solving skills in real-time environments.**

## I. TASK 1. FOLLOW THE INSTRUCTIONS AND COMPLETE ROS 2 TUTORIAL SESSION 3 - MOTION CONTROL OF MANIPULATORS FOLLOWING ALL STANDARD TASKS.

*1) Intro to URDF:* In this part, we explore the practical applications of the Unified Robot Description Format (URDF) in ROS 2, focusing on enhancing a robot's workspace by defining its geometry and physical context. By incorporating key elements like a "world" origin frame, a "table" frame with rectangular prism geometry, and a "$camera\_frame$" with a flipped Z axis, we make the necessary framework for performing collision checking, understanding robot kinematics, and handling transformations. This setup forms the foundation for dynamic path planning and obstacle avoidance, critical capabilities in ROS. By expanding upon the basics, we aim to build a more sophisticated robotic workcell that can be adapted for complex, real-world tasks. The screenshot of the completed part can be found from Fig **??**.

*2) Workcell Xacro:* In this part of the assignment, we transition from using simple URDFs to utilizing XACRO (XML Macros), a more efficient way to manage complex robotic descriptions. Writing URDFs with numerous elements can become tedious and error-prone due to the repetitive nature of defining links and joints. By using XACRO, we can modularize and reuse components, much like functions and classes in programming languages, reducing duplication and minimizing errors. XACRO also offers additional features such as file inclusion, constant variables, and mathematical expressions, allowing for more flexibility and readability in defining robot assemblies.

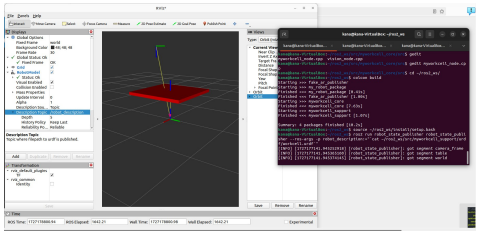

Figure 1. Rviz part 1

In this exercise, we convert the previously created URDF workcell into a XACRO file, leveraging its advantages for a more scalable approach. We then include a pre-defined XACRO macro for a UR5 robot, integrating it into the workspace by linking it to the table. This enables dynamic interaction between the robot and the workcell geometry, setting the stage for further manipulation tasks and path planning exercises. The screenshot of the completed task can be seen from Fig **??**.

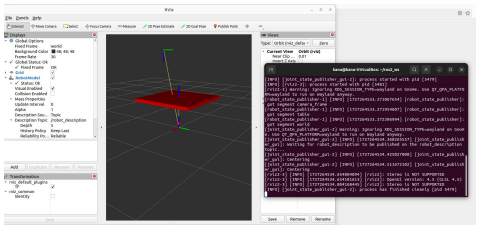

Figure 2. Rviz part 2

*3) Transforms using TF:* In this part of the assignment, we focus on leveraging ROS's TF (Transform) library, which plays a crucial role in monitoring the position and orientation of various parts, robot links, or tools within a workspace. TF allows the system to track and compute transformations between connected frames over time, providing essential context for dynamic environments. This capability is fundamental when working with robotic systems that need to interpret positional data from sensors, such as cameras, and relate it to the robot's own coordinate frames.

In this exercise, the vision system returns the pose of parts relative to the camera's optical frame. How-

ever, for the robot to use this data effectively, the pose must be transformed from the camera's reference frame to the robot's base frame. To achieve this, we modify the service callback inside the $vision\_node$ to apply the necessary transformation. The transformed part pose is then made available to the robot, allowing it to make decisions or take actions based on the correct spatial relationship between the camera and the robot. This step is critical for enabling the robot to interact with objects in its environment with accuracy and precision. The screenshot of the completed task can be seen from Fig 3 and also the one when the base frame is changed is illustrated in Fig 4.

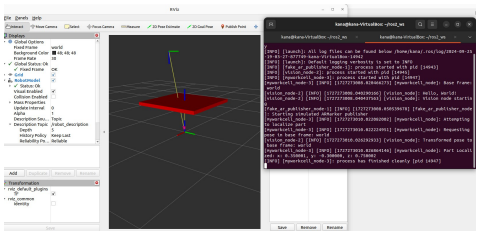

Figure 3. Rviz at tf part

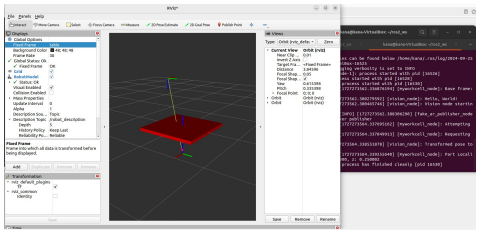

Figure 4. Rviz after changing base frame to table

After this part was done, I realized I haven't added some parts of the code to the urdf file, and because of this it was not showing the ur5 robot properly. This was fixed and the screenshot can be from Fig 5.

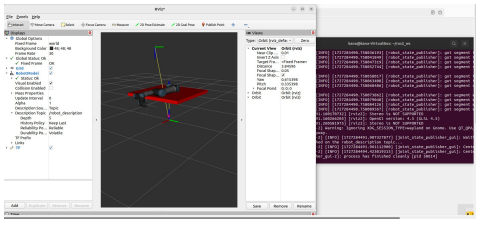

Figure 5. Fixed robot in rviz

*4) Build a MoveIt Package:* In this exercise, we focused on integrating MoveIt!, a powerful motion planning framework in ROS, to enable free-space motion planning for an industrial robot, specifically the UR5. MoveIt! simplifies complex tasks like planning motions between two points in space while avoiding collisions, making it an essential tool for robotic motion control. Although MoveIt! operates with intricate underlying mechanisms, its GUI Setup Assistant allows for an easy configuration process.

Our task is to generate a MoveIt! package for the UR5 workcell we previously built, ensuring it can be used with MoveIt!'s motion-control nodes. By using the MoveIt! Setup Assistant, we will create a package called '$myworkcell\_moveit\_config$' that includes the configuration files necessary for robot motion planning. The configuration will define a single group called "manipulator," which represents the kinematic chain between the UR5's '$base\_link$' and its end effector 'tool0'. This setup will enable us to control the robot's movements with precision while considering the physical constraints of the workspace. The screenshot of the completed task can be seen from Fig 6.

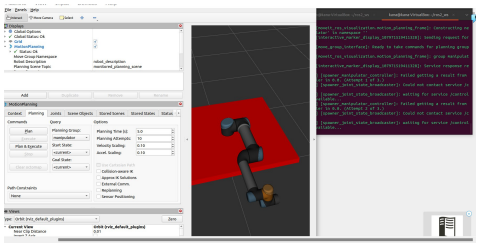

Figure 6. MoverIt Package built

*5) Motion Planning using RViz:* In this exercise, we will utilize the MoveIt! RViz plugin to plan and execute motions on a simulated UR5 robot, gaining hands-on experience with motion planning options and constraints. The first step involves configuring the Displays panel to enable crucial visualization features. We will activate "Show Robot Visual" and "Query Goal State" while leaving "Show Robot Collision" and "Query Start State" disabled for simplicity. Additionally, we will include a trajectory preview slider from the Panels dropdown, allowing us to review the last planned trajectory in detail.

Next, we will ensure that the RRTConnect planner is selected in the Context tab of the motion planning panel. To begin motion planning, we will navigate to the Planning tab and utilize the "Select Goal State" feature by choosing "¡random valid¿". This option attempts to find a collision-free position for the robot, though it may require multiple attempts due to its limitation of 100 random-sampling trials. We will observe the robot's proposed goal position and click "Plan" to visualize the generated motion. To enhance our understanding, we will enable looping animations and trail displays for the planned path. Finally, by clicking "Execute," we will observe the simulated robot update its position in response to the planned trajectory. This iterative process will be repeated, allowing us to experiment with both interactive markers for manual positioning and predefined named poses,

such as “straight up,” further solidifying our grasp of motion planning in ROS. All the steps mentioned above were accomplished and presented in a short video that will be submitted with this report.

## II. Task 2. For exercises 3.0 and 3.1 - modify the setup by adding an additional object to the scene, e.g. a box.

For this task, a simple cube was added to the urdf file. The modified urdf file includes the new link and new joint.

```
<!-- Cube link -->
  <link name="cube">
    <inertial>
      <origin xyz="0 0 0" />
      <mass value="1.0" />
      <inertia  ixx="1.0" ixy="0.0"
      ixz="0.0"  iyy="1.0"
      iyz="0.0" izz="1.0" />
    </inertial>
    <visual>
      <origin xyz="0 0 0"/>
      <geometry>
        <box size="1 1 1" />
      </geometry>
    </visual>
    <collision>
      <origin xyz="0 0 0"/>
      <geometry>
        <box size="1 1 1" />
      </geometry>
    </collision>
  </link>

  <!-- Joint for the cube -->
  <joint name="table_to_cube"
  type="fixed">
    <parent link="table"/>
    <child link="cube"/>
    <origin xyz="2 2 0.5"
    rpy="0 0 0"/>
  </joint>
```

After that, the workspace was built without errors and a cube appeared in the rviz. It can be seen from Fig 7

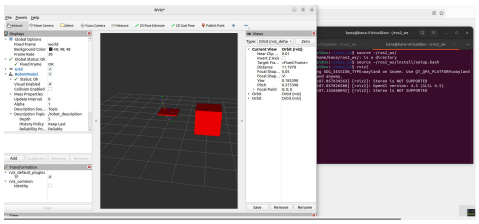

Figure 7. A cube

## III. Task 3. For exercise 3.2 create a second service to return the transformation from the camera to the table.

For this task an additional transformation from the camera to the table was added. In order to do this, additional lines of code were written into *vision_node.cpp*. Mainly, the changes made were the following: 1)Lookup the transformation from the camera to the table frame.This retrieves the transformation between the camera frame and table. The TimePointZero argument means it will use the latest available transformation. 2)Transform the pose using tf2::doTransform(). This applies the transformation from the camera frame to the table frame. $target\_pose\_from\_cam$ is transformed into $target\_pose\_from\_table$. The modified code will be attached to this report. The screenshot of the completed task can be seen from Fig 8.

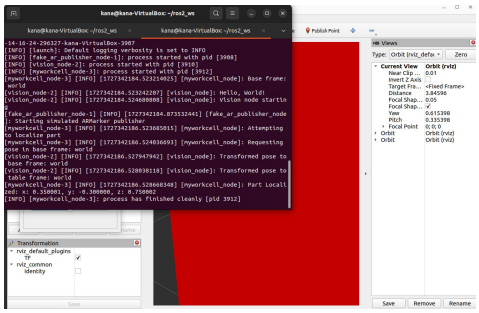

Figure 8. Additional transformation

## IV. Task 4. For exercise 3.4 please consider different robot start and goal poses and plan the robot trajectory with collision avoidance of an additional object (from task 2) on the scene.

An additional box was added for this task since the previous box was a little too big. In the video demonstration the planning was shown from the start position: home to the random value. The task was successfully completed and was shown in the video demonstration attached to this report.

## V. Task 5. Complete Exercise 4.0 following the instructions. Demonstrate the robot motion with the same obstacle avoidance from previous task by running the C++ code.

In this exercise, we transition from using the MoveIt! RViz interface to controlling the robot programmatically via the MoveIt! C++ API. This allows for more flexible and automated control of the robot’s motion, an essential skill for building more sophisticated robotic applications. After setting up a

functional MoveIt! configuration and experimenting with the RViz planning tools, we now focus on integrating motion planning directly into code.

Our task is to modify the '$myworkcell\_core$' node to utilize the MoveItCpp API. This will enable us to plan and execute motions from within a C++ program, rather than relying on the RViz interface. The specific objective is to move the robot's tool frame to the center of a part's location, as determined by the vision node service call. This exercise highlights the simplicity and effectiveness of MoveIt!'s C++ interface for motion control, providing a foundational understanding of how to incorporate real-time planning and execution into custom ROS applications. By automating this process in code, we can enable the robot to autonomously interact with objects in its workspace, an essential step in developing a fully functional robotic system. The task was completed guided by the provided instructions.

## VI. Conclusion

In conclusion, this assignment provided a comprehensive exploration of ROS 2's powerful tools for motion control, from defining a robot's workcell using URDF and XACRO to leveraging MoveIt! for motion planning both visually through RViz and programmatically using C++. We first built a solid foundation by constructing and configuring a workcell, adding dynamic components like the UR5 robot, and ensuring proper transformations between frames using the TF library. By configuring a MoveIt! package, we were able to experiment with path planning and obstacle avoidance in a simulated environment. Finally, transitioning from RViz to MoveItCpp in C++ enabled us to automate robot motion planning, adding a layer of programmability to our setup. Through these exercises, we've gained a deep understanding of robot motion planning and control, setting the stage for more advanced applications in robotics.