



FACULTY OF ENGINEERING AND SCIENCE

Robotics MAS248

ROS UR5 Drawing

Johanna H. Wörz, Sebastian Gehrler, Cătălin M. Orzan

supervized by

Ilya Tyapin

November 24, 2023

TABLE OF CONTENTS

1. Abstract.....	2
2. Introduction.....	2
3. Method.....	2
3.1 Setting up the environment	2
3.1.1 Setting up Virtual Box	2
3.2 Setting up the simulation.....	5
3.3 Implementing Python scripts.....	5
3.3.1 Collision boxes.....	5
3.3.2 Moving robot joints.....	6
3.3.3 Moving the robot above the paper	6
3.3.4 Processing the image	6
3.3.5 Planning and executing the path	7
3.4 Program running.....	8
4. Results.....	10
5. Discussions	10
6. Conclusions.....	10
7. Bibliography	11
8. Appendix.....	12
8.1 Python script.....	12
8.2 Catkin photo	28

1. Abstract

The project consisted of programming an UR5 robotic arm with ROS on Ubuntu 20.04. Setup was performed using the catkin environment, modifying the URDF parameters of the robot, and integrating Python scripts for image processing, path generation, and collision box setup. After achieving each step, we followed it with visualization in MoveIT, coupled with rigorous testing through Gazebo. An Ethernet connection to the robot was established and the program run through ROS, which led to a successful image drawing on the physical UR5. Our study highlights the efficient integration of ROS, MoveIT, Gazebo, and Python scripts in the Ubuntu environment, demonstrating the practical application.

2. Introduction

“Robotics are beginning to cross that line from absolutely primitive motion to motion that resembles animal or human behavior.” as stated by J.J. Abrams.

As the use of robots becomes more human-like and therefore increasingly important, this project provides an excellent introduction to the world of robotics. The primary objective of this project is to draw a picture using the UR5 robot - to achieve this, a virtual machine running Ubuntu 20.04 is used. To program ROS using Gazebo, MoveIt and Python scripting are utilized. Three students from the MAS248 course, Sebastian Gehrler, Cătălin Orzan and Johanna Wörz, are currently working on this project. They have applied the knowledge gained from MAS248 lectures and online research to accomplish the project goals.

3. Method

3.1 Setting up the environment

3.1.1 Setting up Virtual Box

In VirtualBox, a new virtual machine was created by clicking new and following the steps. The following settings were used for optimal performance:

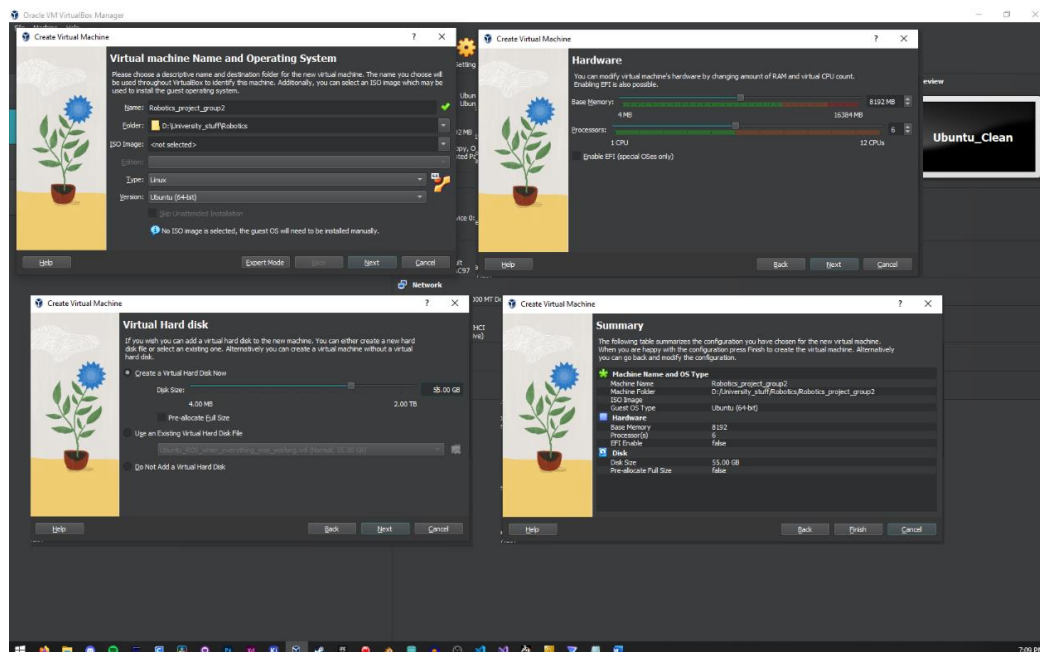


Figure 1: VirtualBox Settings

After creating it with the Linux ISO, provided in the canvas, more video memory was allocated and the bidirectional clipboard was activated in the settings. Then the virtual machine was ready for use. Following applications were installed:

- Terminator;
- Visual Studio;
- ROS;
- MOVEit;
- Catkin;
- RQT and Plugins;
- Python3 and pip.

Then the provided launch file for the UR5 was manually copied into the launch folder for the UR Robot Driver folder (from inside the src in catkin), which has the script command port removed (left: old launch file, right: new one)



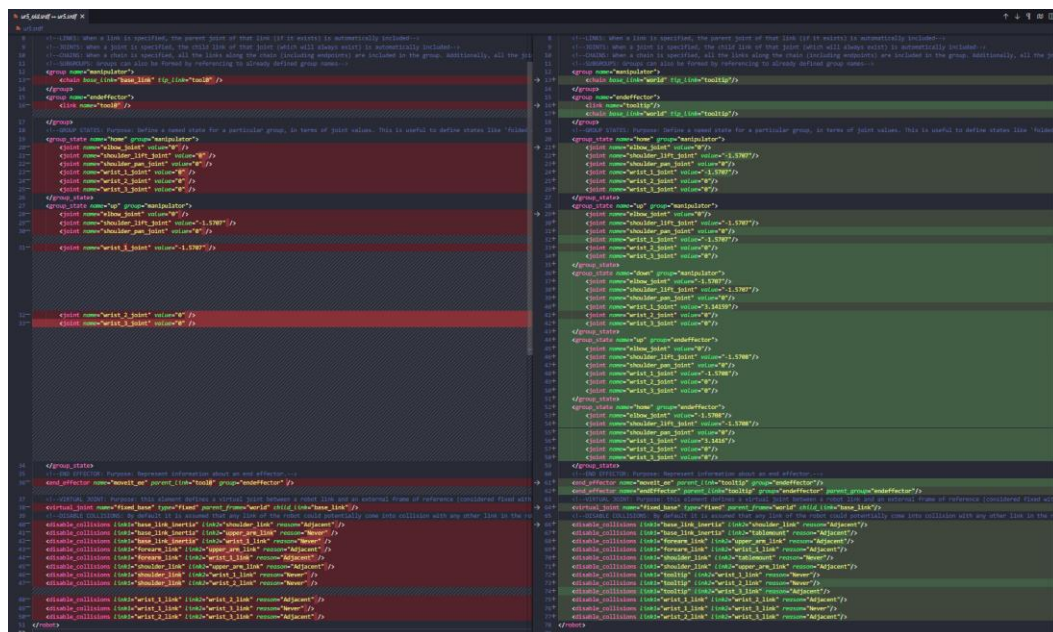
```

1 <?xml version="1.0"?>
2 <launch>
3   <arg name="debug" default="false" doc="Debug flag that will get passed on to ur_common.launch" />
4   <arg name="robot_ip" doc="IP address by which the robot can be reached." />
5   <arg name="reverse_ip" default="" doc="IP of the driver, if set to empty it will detect it aut" />
6   <arg name="reverse_port" default="50001" doc="Port that will be opened by the driver to allow" />
7   <arg name="script_sender_port" default="50002" doc="The driver will offer an interface to rece" />
8   <arg name="trajectory_port" default="50003" doc="Port that will be opened by the driver to all" />
9   <arg name="script_command_port" default="50004" doc="Port that will be opened by the driver to" />
10  <arg name="tf_prefix" default="" doc="tf_prefix used for the robot." />
11  <arg name="controllers" default="joint_state_controller scaled_pos_joint_traj_controller speed" />
12  <arg name="stopped_controllers" default="pos_joint_traj_controller joint_group_vel_controller" />
13  <arg name="controller_config_file" default="$(find ur_robot_driver)/config/ur5_controllers.yaml" />
14  <arg name="robot_description_file" default="$(find ur_description)/launch/load_ur5.launch" />
15  <arg name="kinematics_config" default="$(find ur_description)/config/ur5/default_kinematics.yaml" />
16  <arg name="headless_mode" default="false" doc="Automatically send URScript to robot to execut" />
17  <arg name="ur_hardware_interface_node_required" default="true" doc="Shut down ros environment" />
18  <include file="$(find ur_robot_driver)/launch/ur_common.launch" pass_all_args="true" />
19  <arg name="use_tool_communication" value="false" />
20 /<include>
21 /<launch>
22
1 <?xml version="1.0"?>
2 <launch>
3   <arg name="debug" default="false" doc="Debug flag that will get passed on to ur_common.launch" />
4   <arg name="robot_ip" doc="IP address by which the robot can be reached." />
5   <arg name="reverse_ip" default="" doc="IP of the driver, if set to empty it will detect it aut" />
6   <arg name="reverse_port" default="50001" doc="Port that will be opened by the driver to allow" />
7   <arg name="script_sender_port" default="50002" doc="The driver will offer an interface to rece" />
8   <arg name="trajectory_port" default="50003" doc="Port that will be opened by the driver to all" />
9   <arg name="tf_prefix" default="" doc="tf_prefix used for the robot." />
10  <arg name="controllers" default="joint_state_controller scaled_pos_joint_traj_controller speed" />
11  <arg name="stopped_controllers" default="pos_joint_traj_controller joint_group_vel_controller" />
12  <arg name="controller_config_file" default="$(find ur_robot_driver)/config/ur5_controllers.yaml" />
13  <arg name="robot_description_file" default="$(find ur_description)/launch/load_ur5.launch" />
14  <arg name="kinematics_config" default="$(find ur_description)/config/ur5/default_kinematics.yaml" />
15  <arg name="headless_mode" default="false" doc="Automatically send URScript to robot to execut" />
16  <arg name="ur_hardware_interface_node_required" default="true" doc="Shut down ros environment" />
17  <include file="$(find ur_robot_driver)/launch/ur_common.launch" pass_all_args="true" />
18  <arg name="use_tool_communication" value="false" />
19 /<include>
20 /<launch>
21

```

Figure 2 Launch file of UR5

In order to modify the robot as needed, files needed to be modified. Starting with the srdf file from the fmauch_universal_robot/ur5_moveit_config/config/ur5.srdf accordingly: (left: old launch file, right: new one)



```

1 <?xml version="1.0"?>
2 <robot name="ur5">
3   <link name="base_link">
4     <visual>
5       <model name="ur5_base_link" />
6     </visual>
7     <collision>
8       <model name="ur5_base_link" />
9     </collision>
10    <inertial>
11      <mass value="10.0" />
12      <center_of_mass x="0.0" y="0.0" z="0.0" />
13      <inertia ixx="0.0001" iyy="0.0001" izz="0.0001" />
14      <ixy="0.0" iyz="0.0" ixy="0.0" />
15    </inertial>
16  </link>
17  <link name="wrist_1_link">
18    <visual>
19      <model name="ur5_wrist_1_link" />
20    </visual>
21    <collision>
22      <model name="ur5_wrist_1_link" />
23    </collision>
24    <inertial>
25      <mass value="1.0" />
26      <center_of_mass x="0.0" y="0.0" z="0.0" />
27      <inertia ixx="0.0001" iyy="0.0001" izz="0.0001" />
28      <ixy="0.0" iyz="0.0" ixy="0.0" />
29    </inertial>
30  </link>
31  <link name="wrist_2_link">
32    <visual>
33      <model name="ur5_wrist_2_link" />
34    </visual>
35    <collision>
36      <model name="ur5_wrist_2_link" />
37    </collision>
38    <inertial>
39      <mass value="1.0" />
40      <center_of_mass x="0.0" y="0.0" z="0.0" />
41      <inertia ixx="0.0001" iyy="0.0001" izz="0.0001" />
42      <ixy="0.0" iyz="0.0" ixy="0.0" />
43    </inertial>
44  </link>
45  <link name="wrist_3_link">
46    <visual>
47      <model name="ur5_wrist_3_link" />
48    </visual>
49    <collision>
50      <model name="ur5_wrist_3_link" />
51    </collision>
52    <inertial>
53      <mass value="1.0" />
54      <center_of_mass x="0.0" y="0.0" z="0.0" />
55      <inertia ixx="0.0001" iyy="0.0001" izz="0.0001" />
56      <ixy="0.0" iyz="0.0" ixy="0.0" />
57    </inertial>
58  </link>
59  <link name="wrist_4_link">
60    <visual>
61      <model name="ur5_wrist_4_link" />
62    </visual>
63    <collision>
64      <model name="ur5_wrist_4_link" />
65    </collision>
66    <inertial>
67      <mass value="1.0" />
68      <center_of_mass x="0.0" y="0.0" z="0.0" />
69      <inertia ixx="0.0001" iyy="0.0001" izz="0.0001" />
70      <ixy="0.0" iyz="0.0" ixy="0.0" />
71    </inertial>
72  </link>
73  <link name="wrist_5_link">
74    <visual>
75      <model name="ur5_wrist_5_link" />
76    </visual>
77    <collision>
78      <model name="ur5_wrist_5_link" />
79    </collision>
80    <inertial>
81      <mass value="1.0" />
82      <center_of_mass x="0.0" y="0.0" z="0.0" />
83      <inertia ixx="0.0001" iyy="0.0001" izz="0.0001" />
84      <ixy="0.0" iyz="0.0" ixy="0.0" />
85    </inertial>
86  </link>
87  <joint name="base_joint" type="revolute" parent="base_link" child="wrist_1_link">
88    <axis x="0.0" y="1.0" z="0.0" />
89    <limit lower="0.0" upper="3.14159" />
90    <dynamic_friction coefficient="0.1" />
91    <static_friction coefficient="0.1" />
92  </joint>
93  <joint name="wrist_1_joint" type="revolute" parent="wrist_1_link" child="wrist_2_link">
94    <axis x="0.0" y="0.0" z="1.0" />
95    <limit lower="0.0" upper="3.14159" />
96    <dynamic_friction coefficient="0.1" />
97    <static_friction coefficient="0.1" />
98  </joint>
99  <joint name="wrist_2_joint" type="revolute" parent="wrist_2_link" child="wrist_3_link">
100   <axis x="0.0" y="1.0" z="0.0" />
101   <limit lower="0.0" upper="3.14159" />
102   <dynamic_friction coefficient="0.1" />
103   <static_friction coefficient="0.1" />
104 </joint>
105 <joint name="wrist_3_joint" type="revolute" parent="wrist_3_link" child="wrist_4_link">
106   <axis x="0.0" y="0.0" z="1.0" />
107   <limit lower="0.0" upper="3.14159" />
108   <dynamic_friction coefficient="0.1" />
109   <static_friction coefficient="0.1" />
110 </joint>
111 <joint name="wrist_4_joint" type="revolute" parent="wrist_4_link" child="wrist_5_link">
112   <axis x="0.0" y="1.0" z="0.0" />
113   <limit lower="0.0" upper="3.14159" />
114   <dynamic_friction coefficient="0.1" />
115   <static_friction coefficient="0.1" />
116 </joint>
117 <end effector name="wrist_5" parent="wrist_5_link" group="wrist_5" />
118 </robot>

```

Figure 3: srdf configuration of UR5

The next to change is the macro.xacro file from `fmauch_universal_robot/ur_description/urdf/inc/ur5_macro.xacro`, as seen in Figure 4.

```

1  ur5_macro.xacro X
2  1 <?xml version="1.0"?>
3  2 <robot xmlns:xacro="http://wiki.ros.org/xacro">
4  3 <!--
5  4 Convenience wrapper for the 'ur_robot' macro which provides default values
6  5 for the various "parameters files" parameters for a UR5.
7  6
8  7 This file can be used when composing a more complex scene with one or more
9  8 UR5 robots.
10 9
11 10 While the generic 'ur_robot' macro could also be used, it would require
12 11 the user to provide values for all parameters, as that macro does not set
13 12 any model-specific defaults (not even for the generic parameters, such as
14 13 the visual and physical parameters and joint limits).
15 14
16 15 Refer to the main 'ur_macro.xacro' in this package for information about
17 16 use, contributors and limitations.
18 17
19 18 NOTE: users will most likely want to override "at least" the
20 19 'kinematics.parameters.file' parameter. Otherwise, a default kinematic
21 20 model will be used, which will almost certainly not correspond to any
22 21 real robot.
23 22
24 23 <xacro:macro name="ur5_robot" params="
25 24 prefix
26 25 joint_limits_parameters_file:="$(find ur_description)/config/ur5/joint_limits.yaml"
27 26 kinematics_parameters_file:="$(find ur_description)/config/ur5/default_kinematics.yaml"
28 27 physical_parameters_file:="$(find ur_description)/config/ur5/physical_parameters.yaml"
29 28 visual_parameters_file:="$(find ur_description)/config/ur5/visual_parameters.yaml"
30 29 transmission_hw_interface: hardware_interface/PositionJointInterface
31 30 safety_limits:=false
32 31 safety_pos_margin:=0.15
33 32 safety_k_position:=20"
34 33 >
35 34
36 35 <xacro:include filename="$(find ur_description)/urdf/inc/ur_macro.xacro"/>
37 36
38 37 <link name="world"/>
39 38
40 39 <link name="tooltip"/>
41 40
42 41 <link name="toolinterface"/>
43 42 <origin xyz="0 0 0" rpy="0 0 0"/>
44 43 <visual>
45 44 <geometry>
46 45 <mesh filename="package://ur_description/meshes/ur5/collision/finaltool.STL" scale="0.001 0.001 0.001"/>
47 46 </geometry>
48 47 </visual>
49 48 <collision>
50 49 <geometry>
51 50 <mesh filename="package://ur_description/meshes/ur5/visual/finaltool.dae" scale="0.001 0.001 0.001"/>
52 51 </geometry>
53 52 </collision>
54 53 </link>
55 54
56 55 <link name="tablemount">
57 56 <origin xyz="0 0 0" rpy="0 0 0"/>
58 57 <visual>
59 58 <geometry>
60 59 <mesh filename="package://ur_description/meshes/ur5/collision/tablemount.STL" scale="0.001 0.001 0.001"/>
61 60 </geometry>
62 61 </visual>
63 62 <collision>
64 63 <geometry>
65 64 <mesh filename="package://ur_description/meshes/ur5/visual/tablemount.dae" scale="0.001 0.001 0.001"/>
66 65 </geometry>
67 66 </collision>
68 67 </link>
69 68
70 69 <joint name="world_joint" type="fixed">
71 70 <parent link="world"/>
72 71 <child link="tablemount"/>
73 72 <origin xyz="0 0 0.30" rpy="0 0 0"/>
74 73 </joint>
75 74
76 75 <joint name="tablemount_to_base_link_joint" type="fixed">
77 76 <parent link="tablemount"/>
78 77 <child link="$(prefix)base_link"/>
79 78 <origin xyz="0 0 0" rpy="0 0 $(pi)"/>
80 79 </joint>
81 80
82 81 <joint name="wrist_3_to_toolinterface_joint" type="fixed">
83 82 <parent link="$(prefix)wrist_3_link"/>
84 83 <child link="toolinterface"/>
85 84 <origin xyz="0 0 0" rpy="0 0 3.5"/>
86 85 </joint>
87 86
88 87 <joint name="toolinterface_to_tooltip_joint" type="fixed">
89 88 <parent link="toolinterface"/>
90 89 <child link="tooltip"/>
91 90 <origin xyz="0.000 -0.115 0.107" rpy="-1.57086327 0 0"/>
92 91 </joint>
93 92
94 93
95 94 <xacro:ur_robot
96 95 prefix="$(prefix)"
97 96 joint_limits_parameters_file="$(joint_limits_parameters_file)"
98 97 kinematics_parameters_file="$(kinematics_parameters_file)"
99 98 physical_parameters_file="$(physical_parameters_file)"
100 99 visual_parameters_file="$(visual_parameters_file)"
101 100 transmission_hw_interface="$(transmission_hw_interface)"
102 101 safety_limits="$(safety_limits)"
103 102 safety_pos_margin="$(safety_pos_margin)"
104 103 safety_k_position="$(safety_k_position)"
105 104 />
106 105
107 106 <gazebo>
108 107 <plugin name="ros_control" filename="libgazebo_ros_control.so">
109 108 <!-- robot_hardware_interface -->
110 109 <!-- robot_hardware_interface -->
111 110 </plugin>
112 111 </gazebo>
113 112 </xacro:macro>
114 113 </robot>

```

Figure 4 `ur5_macro.xacro`

With those adjustments, the simulated UR5 has the orientation and elements present on the real setup of the robots, as seen in Figure 5: Simulation (left side: MOVEit simulation, right side: robot setup).

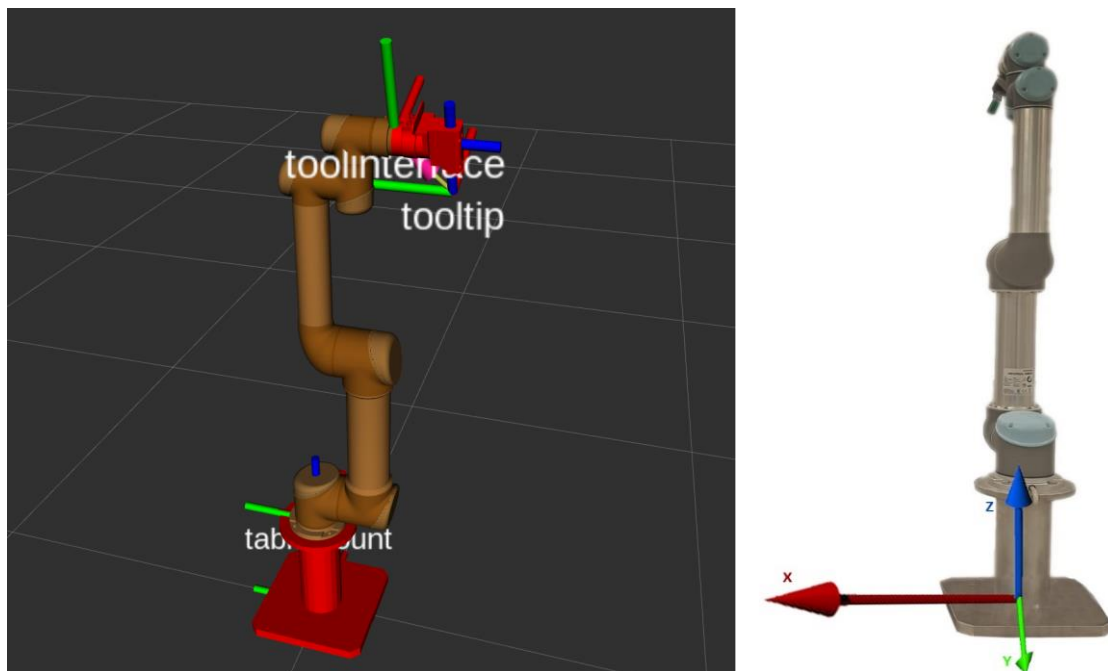


Figure 5: Simulation and real setup

3.2 Setting up the simulation

To set up the simulation, following command lines were utilized:

```
source /opt/ros/noetic/setup.bash
cd ~/catkin_ws
source devel/setup.bash
catkin build
cd src
roscore
roslaunch ur_gazebo ur5_bringup.launch
roslaunch ur5_moveit_config ur5_moveit_planning_execution.launch sim:=true
roslaunch ur5_moveit_config moveit_rviz.launch config:=true
```

3.3 Implementing Python scripts

The goal of the project is to draw the contours of an image by processing it to find waypoints that can then be executed by the robot. This will be split into several tasks:

1. Attaching a collision box to ensure that the robot cannot move outside of the designated area (in our case, the desk);
2. Moving the robot joints in order to avoid singularities in the home position;
3. Moving the robot above the paper on which the image will be drawn on;
4. Processing the image to find its contours and generate waypoints from them;
5. Planning the path by adding these waypoints and then executing it.

Several python scripts were provided. One to process the image, another to generate the path and one to setup collision boxes. We started by running them on their own.

3.3.1 Collision boxes

The first script is the *collision_scene_example.py*. It is used to generate boundaries that the robot cannot leave. This is done for safety reasons as well as preventing robot joints from crashing into the ground. We chose the area of the table as a floor and added four walls around it with a height of two meters, so the robot cannot leave the box. To do this the values in the python script are modified to fit our setup. The coordinates are all related to the robot origin, which is mounted at a distance from the edge of the table ($x = 16.4$ cm, $y = -16.4$ cm).



Figure 6: Robot origin measurement

The coordinates of the box relate to the center of the box. Therefore, to position the center of the table at the actual place in the simulation, we need to subtract the offset distance from the center of the table. For example, if the table has the following dimensions: 1.8 m x 0.8 m x 0.05 m, the pose coordinates are half of the size minus the offset.

```

163
164     def add_floor(self):
165         box1_pose = [(1.8/2-0.164), -(0.8/2-0.164), -0.026, 0, 0, 0, 1] # Table
166         positioning, where 0.9 is half of the table with
167         box1_dimensions = [1.8, 0.8, 0.05]
168         self.add_box_object("floor", box1_dimensions, box1_pose)
169

```

Figure 7: Collision box and calculated coordinates

With the floor (represented by the desk) added, as well as the four walls, this script can be implemented into the final project script. This is done in the class *CollisionSceneExample()*.

3.3.2 Moving robot joints

The next step is moving the robot to a pose using joint angles. This is done to avoid singularities, as the robot starts in a position where they can occur. The angles are changed only slightly from the initial position. The joint angles used are visible in Figure 8.

```

joint_goal[0] = 0.001
joint_goal[1] = -1.0-(pi/2)
joint_goal[2] = -1.0
joint_goal[3] = 0.001-(pi/2)
joint_goal[4] = 0.001
joint_goal[5] = 0.9

```

Figure 8: New joint angle

3.3.3 Moving the robot above the paper

```

pose_goal.orientation.w = 1.0
pose_goal.position.x = 0.5
pose_goal.position.y = -0.4
pose_goal.position.z = 0.20

```

Figure 9: Position above paper

From the new singularity free position, a new pose was planned using coordinates to guide the robot to a place above the paper. This is done with cartesian coordinates in the world frame. Here the robot is moved to a pose 20 cm above the paper ($x = 50$ cm, $y = -40$ cm) with the pen orientation (w) downwards. The chosen coordinates are shown in Figure 9.

3.3.4 Processing the image

For the script to process the image (*process_image.py*), first step needed was setting up the path for the image (Figure 10: Image path). Running the script could now generate points based on the countour of the image.

```

1 #!/usr/bin/env python
2
3 # https://github.com/bigdayangyu/UR5-project/blob/master/ur5_project.m
4
5 import numpy as np
6 import cv2
7 from matplotlib import pyplot as plt
8 img = cv2.imread("goose.jpg")
9

```

Figure 10: Image path

The entire script can be added as a function to the *project.py* script, which returns the x, y and z coordinates of the contour points. The chosen image is a goose, based on the famous samperson's game, The Desktop Goose. Both the image and the generated points can be visualized in Figure 11.

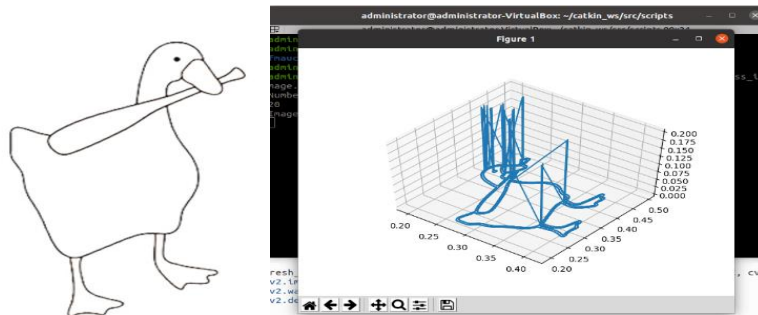


Figure 11: Goose.jpg and pathway

3.3.5 Planning and executing the path

As basis for the project script *project.py* the *move_group_python_interface_tutorial.py* script was used. All the necessary functions and classes of the other scripts were implemented and used in the *main()* function (Figure 12: main function of the project.py).

```

638 def main():
639     try:
640         print("")
641         print("-----")
642         print("Welcome to the MoveIt MoveGroup Python Interface")
643         print("-----")
644         print("Press (Ctrl-C) to exit at any time")
645         print("")
646         input(
647             "----- Press 'Enter' to begin the application by setting up the moveit_commander ----"
648         )
649         tutorial = MoveGroupPythonInterfaceTutorial()
650         x_coord, y_coord, z_coord = processing_image()
651         print("X: ", x_coord[0])
652         print("Y: ", y_coord[0])
653         print("Z: ", z_coord[0])
654
655         input("----- Press 'Enter' to attach the floor and walls to the robot ----")
656         tutorial.go_to_joint_end()
657         tutorial.add_box()
658         tutorial.attach_box()
659         collision = CollisionSceneExample()
660         collision.add_all()
661
662         input("----- Press 'Enter' to execute a movement using a joint state goal ----")
663         tutorial.go_to_joint_state()
664
665         input("----- Press 'Enter' to execute a movement using a pose goal ----")
666         tutorial.go_to_pose_goal()
667
668         input("----- Press 'Enter' to plan and execute a path with an attached collision object ----")
669         cartesian_plan, fraction = tutorial.plan_cartesian_path()
670         tutorial.display_trajectory(cartesian_plan)
671         tutorial.execute_plan(cartesian_plan)
672
673         input("----- Press 'Enter' to execute a movement using a joint end position ----")
674         tutorial.detach_box()
675         tutorial.remove_box()
676         tutorial.go_to_joint_end()
677
678         print("----- Process completed!")
679     except rospy.ROSInterruptException:
680         return
681     except KeyboardInterrupt:
682         return

```

Figure 12: main function of the project.py

One of the initial function of the script is processing the image and storing the found coordinates of the waypoints (line 650 in Figure 12: main function of the project.py). After that the robot movement is set to its initial position, the `go_to_joint_end()` function makes sure the correct place is reached. Then the collision scene is set up calling all the functions that add the floor and wall boxes. Next, the robot moves out of the singularity with `go_to_joint_state()` and then moves to the position above the paper with `go_to_pose_goal()` function. After planning the paths, Rviz takes care of the visualization and execution (Lines 668 to 670 in **Error! Reference source not found.2**).

The last step is removing the collision objects and return the robot to the home position.

3.4 Program running

3.4.1 Connection to the robot

A laptop running Windows 10 was utilized to connect ROS with VirtualBox to the physical UR5 Robot. First step of this process consisted in connecting the physical ethernet cable. Adapter settings from windows were changed accordingly: by right clicking the ethernet adapter and selecting properties, the VirtualBox service was installed under IPV4, and the the following data was written: 192.168.56.100 for the computer's IP, 192.168.56.101 for the robot (gateway) and 255.255.255.0 for the submask.

The settings of the utilized virtual machine in Oracle VM VirtualBox were changed as following: from the Network panel of the settings screen, the first two adapters were enabled, first one being set to bridge adapter to the right ethernet adapter in windows and the second one to host-only adapter with virtualbox.

Inside the Ubuntu running machine, the Network settings require changes as well. IPV4 was set to manual mode, with the same IP addresses as the ones in windows control panel.

The UR5 robot had to be configured using the teach pendant.

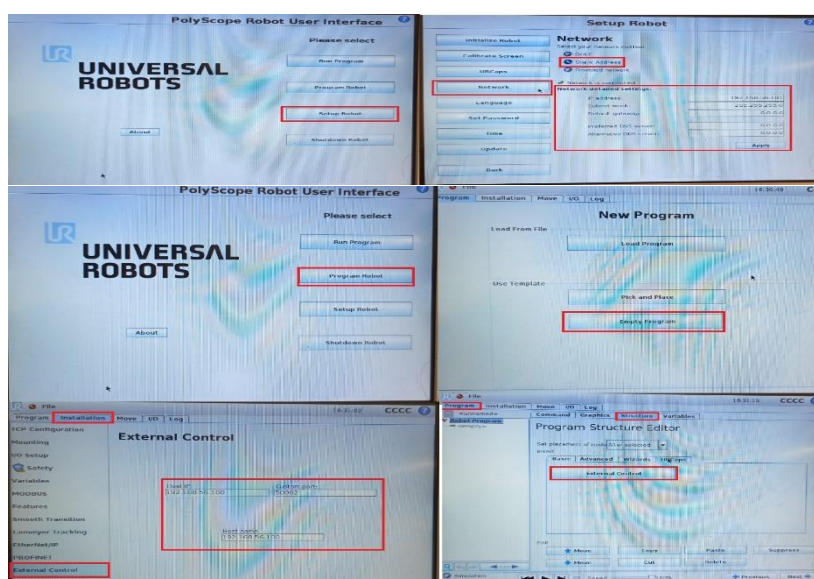


Figure 13: Teach Pendant Settings

As visible in Figure 13: Teach Pendant Settings, Setup Robot was selected and the IP was set to 192.168.56.101. The robot was powered on and brakes were released from the "Initialize Robot" menu. Back to the main menu, an empty program was opened. From the installation tab, the IP of the host was checked, in order to assure it's set to 192.168.56.100. With this steps accomplished, the program tab was selected, and external control was inserted into the program from the structure subtab. At this point, the robot was able to be seen by both the windows and the virtual machine running Ubuntu, fact demonstrated by the ability to successfully run the ping command to the robot's IP. A series of commands can now be used to run the program:

```
mkdir -p
/home/administrator/catkin_ws/src/Universal_Robots_ROS_Driver/ur_calibration/etc
source /opt/ros/noetic/setup.bash
cd ~/catkin_ws
source devel/setup.bash
catkin clean
catkin build
cd src
roscore
roslaunch ur_calibration calibration_correction.launch robot_ip:=192.168.56.101
target_filename:="$(rospack find ur_calibration)/etc/my_ur5_calibration.yaml"
```

3.4.2 Running the robot

With this steps accomplished, the robot can be started from the play button at the bottom of the teach pendant, at first with low speed set (under 20%). Then the following commands:

```
roslaunch ur_robot_driver ur5_bringup.launch robot_ip:=192.168.56.101
[reverse_prot:=REVERSE_PORT] kinematics_config:="$(rospack find
ur_calibration)/etc/my_ur5_calibration.yaml"
roslaunch ur5_moveit_config ur5_moveit_planning_execution.launch limited:=true
roslaunch ur5_moveit_config moveit_rviz.launch config:=true
```

The last command is running the script. For us, this could be accomplished using this command line:

```
roslaunch project project.py
```

First, the robot positions itself above the drawing area and adjusts its orientation to ensure accurate pen drawing. Once it is properly aligned, it proceeds to draw the processed image.

4. Results



Figure 14: Goose drawing

After successfully implementing and executing the project, the UR5 robot was able to draw the desired image, as shown in Figure 14: Goose drawing.

5. Discussions

Improvements could be made by reconsidering the chosen joint positions and conducting a trajectory analysis.. To enhance the operation, the speed of the robot could be increased, and the python code for processing the image could be optimized as it currently draws the outlines multiple times.

6. Conclusions

In summary, the project successfully integrated ROS, Gazebo and Python scripting to enable the UR5 robot to create an image using a pen. The teamwork of MAS248 students Sebastian Gehrler and Cătălin M. Orzan and Johanna H. Wörz demonstrated the practical application of theoretical knowledge in robotics.

7. Bibliography

Tyapin, I. (2023). "MAS248 Robotics" class.

ROS Wiki. Installation. Retrieved from <http://wiki.ros.org/noetic/Installation>

ROS Industrial. Getting Started with a Universal Robot and ROS-Industrial. Retrieved from http://wiki.ros.org/universal_robot/Tutorials/Getting%20Started%20with%20a%20Universal%20Robot%20and%20ROS-Industrial

ROS-Industrial. Universal Robot. Retrieved from https://github.com/ros-industrial/universal_robot

YouTube. Dec 29, 2020. Configuring UR5 with ROS. Retrieved from https://www.youtube.com/watch?v=BS6pFmr7_lA

GitHub. Desktop Goose. Retrieved from <https://github.com/topics/desktop-goose>

OpenAI. (2023). UiO GPT (Version GPT-3.5 Turbo) [Large language model]. University of Agder. (Used mostly for debugging purposes)

8. Appendix

8.1 Python script

```
#!/usr/bin/env python3

# Software License Agreement (BSD License)
#
# Copyright (c) 2013, SRI International
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#
# * Redistributions of source code must retain the above copyright
#   notice, this list of conditions and the following disclaimer.
# * Redistributions in binary form must reproduce the above
#   copyright notice, this list of conditions and the following
#   disclaimer in the documentation and/or other materials provided
#   with the distribution.
# * Neither the name of SRI International nor the names of its
#   contributors may be used to endorse or promote products derived
#   from this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
# COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
# BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
# CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
# LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
# ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.
#
# Author: Acorn Pooley, Mike Lautman

## BEGIN_SUB_TUTORIAL imports
##
## To use the Python MoveIt interfaces, we will import the `moveit_commander`_
namespace.
## This namespace provides us with a `MoveGroupCommander`_ class, a
`PlanningSceneInterface`_ class,
## and a `RobotCommander`_ class. More on these below. We also import `rospy`_
and some messages that we will use:
```

```
##

# Python 2/3 compatibility imports

#move group python interface
from __future__ import print_function
from six.moves import input
try:
    from math import pi, tau, dist, fabs, cos
except: # For Python 2 compatibility
    from math import pi, fabs, cos, sqrt

tau = 2.0 * pi

def dist(p, q):
    return sqrt(sum((p_i - q_i) ** 2.0 for p_i, q_i in zip(p, q)))

from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list
## END_SUB_TUTORIAL
from moveit_commander import RobotCommander, PlanningSceneInterface
import geometry_msgs.msg
import time

import sys
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
#processing the image
import numpy as np
import matplotlib.pyplot as plt
import cv2
from mpl_toolkits import mplot3d

def all_close(goal, actual, tolerance):
    """
    Convenience method for testing if the values in two lists are within a
    tolerance of each other.
    For Pose and PoseStamped inputs, the angle between the two quaternions is
    compared (the angle
    between the identical orientations q and -q is calculated correctly).
    @param: goal      A list of floats, a Pose or a PoseStamped
    @param: actual    A list of floats, a Pose or a PoseStamped
    @param: tolerance A float
    @returns: bool
    """
```



```

if type(goal) is list:
    for index in range(len(goal)):
        if abs(actual[index] - goal[index]) > tolerance:
            return False

elif type(goal) is geometry_msgs.msg.PoseStamped:
    return all_close(goal.pose, actual.pose, tolerance)

elif type(goal) is geometry_msgs.msg.Pose:
    x0, y0, z0, qx0, qy0, qz0, qw0 = pose_to_list(actual)
    x1, y1, z1, qx1, qy1, qz1, qw1 = pose_to_list(goal)
    # Euclidean distance
    d = dist((x1, y1, z1), (x0, y0, z0))
    # phi = angle between orientations
    cos_phi_half = fabs(qx0 * qx1 + qy0 * qy1 + qz0 * qz1 + qw0 * qw1)
    return d <= tolerance and cos_phi_half >= cos(tolerance / 2.0)

return True

# processing the image
def processing_image():
    img = cv2.imread('/home/johanna/catkin_ws/src/project/src/Goose.jpg')
    img = cv2.rotate(img, cv2.ROTATE_90_CLOCKWISE)
    blur = cv2.GaussianBlur(img, (5,5), cv2.BORDER_DEFAULT)
    imggray = cv2.cvtColor(blur, cv2.COLOR_BGR2GRAY)
    thresh_image =
cv2.adaptiveThreshold(imggray, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
cv2.THRESH_BINARY, 11, 5)
    # Find Canny edges
    edged = cv2.Canny(thresh_image, 30, 200)
    contours, hierarchy =
cv2.findContours(edged, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
    # drawing contours over blank image
    contours_image = np.zeros(img.shape, dtype=np.uint8);
    cv2.drawContours(contours_image, contours, -1, (0,255,0), 3)
    print("Number of Contours found = " + str(len(contours)))
    x_coord = []
    y_coord = []
    z_coord = []
    for cont in contours:
        pixel_pose = cont.reshape(-1,2)
        x_coord.extend(pixel_pose[:,1])
        y_coord.extend(pixel_pose[:,0])

        z_coord_zero = [0.0 for i in pixel_pose]
        z_coord.extend(z_coord_zero)

```

```

x_coord.extend([pixel_pose[-1,1]])
y_coord.extend([pixel_pose[-1,0]])
z_coord.extend([0.1])

#scale the image to reasonable size
print("Image size = ",img.shape)
x_coord = (-min(x_coord) + x_coord)/(max(x_coord)-min(x_coord))*0.105#+
0.20;
y_coord = (-min(y_coord) + y_coord)/(max(y_coord)-min(y_coord))*0.145 #+
0.20;

return x_coord, y_coord, z_coord

# Boundaries
class CollisionSceneExample(object):
    def __init__(self):
        self._scene = PlanningSceneInterface()

        # clear the scene
        self._scene.remove_world_object()

        self.robot = RobotCommander()

        # pause to wait for rviz to load
        # print("===== Waiting while RVIZ displays the scene with
obstacles...")

        # TODO: need to replace this sleep by explicitly waiting for the scene
to be updated.
        rospy.sleep(2)

    def add_floor(self):
        box1_pose = [(1.8/2-0.164), -(0.8/2-0.164), -0.025, 0, 0, 0, 1]
        box1_dimensions = [1.8, 0.8, 0.05] #table length,

        self.add_box_object("floor", box1_dimensions, box1_pose)

        self.add_box_object("floor", box1_dimensions, box1_pose)

        # print("===== Added one obstacle to RViz!!")

    def add_four_walls(self):
        box1_pose = [-0.164, -(0.4-0.164), 1, 0, 0, 0, 1]
        box1_dimensions = [0.001, 0.8, 2]

        box2_pose = [(0.9-0.164), 0.164, 1, 0, 0, 0, 1]
        box2_dimensions = [1.8, 0.001, 2]

```

```

box3_pose = [1.8-0.164, -(0.4-0.164), 1, 0, 0, 0, 1]
box3_dimensions = [0.001, 0.8, 2]

box4_pose = [(0.9-0.164), -(0.8-0.164), 1, 0, 0, 0, 1]
box4_dimensions = [1.8, 0.001, 2]

self.add_box_object("wall1", box1_dimensions, box1_pose)
self.add_box_object("wall2", box2_dimensions, box2_pose)
self.add_box_object("wall3", box3_dimensions, box3_pose)
self.add_box_object("wall4", box4_dimensions, box4_pose)

# print("===== Added 4 obstacles to the scene!!")

def add_all(self):
    self.add_floor()
    self.add_four_walls()

def add_box_object(self, name, dimensions, pose):
    p = geometry_msgs.msg.PoseStamped()
    p.header.frame_id = self.robot.get_planning_frame()
    p.header.stamp = rospy.Time.now()
    p.pose.position.x = pose[0]
    p.pose.position.y = pose[1]
    p.pose.position.z = pose[2]
    p.pose.orientation.x = pose[3]
    p.pose.orientation.y = pose[4]
    p.pose.orientation.z = pose[5]
    p.pose.orientation.w = pose[6]

    self._scene.add_box(name, p, (dimensions[0], dimensions[1],
dimensions[2]))

class MoveGroupPythonInterfaceTutorial(object):
    """MoveGroupPythonInterfaceTutorial"""

    def __init__(self):
        super(MoveGroupPythonInterfaceTutorial, self).__init__()

        ## BEGIN_SUB_TUTORIAL setup
        ##
        ## First initialize `moveit_commander`_ and a `rospy`_ node:
        moveit_commander.roscpp_initialize(sys.argv)
        rospy.init_node("move_group_python_interface_tutorial",
anonymous=True)

        ## Instantiate a `RobotCommander`_ object. Provides information such
as the robot's
        ## kinematic model and the robot's current joint states

```



```

joint_goal[3] = 0.001-(pi/2)
joint_goal[4] = 0.001
joint_goal[5] = 0.9

# The go command can be called with joint values, poses, or without
any
# parameters if you have already set the pose or joint target for the
group
move_group.go(joint_goal, wait=True)

# Calling ``stop()`` ensures that there is no residual movement
move_group.stop()

## END_SUB_TUTORIAL

# For testing:
current_joints = move_group.get_current_joint_values()
return all_close(joint_goal, current_joints, 0.01)

def go_to_pose_goal(self):
    # Copy class variables to local variables to make the web tutorials
more clear.
    # In practice, you should use the class variables directly unless you
have a good
    # reason not to.
    move_group = self.move_group

    ## BEGIN_SUB_TUTORIAL plan_to_pose
    ##
    ## Planning to a Pose Goal
    ## ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    ## We can plan a motion for this group to a desired pose for the
    ## end-effector:
    pose_goal = geometry_msgs.msg.Pose()
    #pose_goal.orientation.x = 0.0
    #pose_goal.orientation.y = 0.0
    #pose_goal.orientation.z = 0.0
    pose_goal.orientation.w = 1.0
    pose_goal.position.x = 0.5
    pose_goal.position.y = -0.4
    pose_goal.position.z = 0.20

    move_group.set_pose_target(pose_goal)

    ## Now, we call the planner to compute the plan and execute it.
    plan = move_group.go(wait=True)

```



```

# Calling `stop()` ensures that there is no residual movement
move_group.stop()
# It is always good to clear your targets after planning with poses.
# Note: there is no equivalent function for
clear_joint_value_targets()
move_group.clear_pose_targets()

## END_SUB_TUTORIAL

# For testing:
# Note that since this section of code will not be included in the
tutorials
# we use the class variable rather than the copied state variable
current_pose = self.move_group.get_current_pose().pose
return all_close(pose_goal, current_pose, 0.01)

def plan_cartesian_path(self, scale=1):
    # Copy class variables to local variables to make the web tutorials
    more clear.
    # In practice, you should use the class variables directly unless you
    have a good
    # reason not to.
    move_group = self.move_group
    x_coord, y_coord, z_coord = processing_image()

    ## BEGIN_SUB_TUTORIAL plan_cartesian_path
    ##
    ## Cartesian Paths
    ## ^^^^^^^^^^^^^^^^^
    ## You can plan a Cartesian path directly by specifying a list of
    waypoints
    ## for the end-effector to go through. If executing interactively in
    a
    ## Python shell, set scale = 1.0.

    waypoints = []

    wpose = move_group.get_current_pose().pose

    wpose.orientation.x = 0.0
    wpose.orientation.y = 0.0
    wpose.orientation.z = 0.0
    wpose.orientation.w = 1.0

    for j in range(len(x_coord)):
        wpose.position.x = x_coord[j] + 0.3
        wpose.position.y = y_coord[j] - 0.55

```

```

wpose.position.z = z_coord[j]
waypoints.append(copy.deepcopy(wpose))

poses = geometry_msgs.msg.PoseArray()
poses.header.frame_id = self.move_group.get_planning_frame()
poses.poses = [point for point in waypoints]
self.poseArray_publisher.publish(poses)

# We want the Cartesian path to be interpolated at a resolution of 1
cm
# which is why we will specify 0.01 as the eef_step in Cartesian
# translation. We will disable the jump threshold by setting it to
0.0,
# ignoring the check for infeasible jumps in joint space, which is
sufficient
# for this tutorial.
(plan, fraction) = move_group.compute_cartesian_path(
    waypoints, 0.01, 0.0 # waypoints to follow # eef_step
) # jump_threshold

# Note: We are just planning, not asking move_group to actually move
the robot yet:
return plan, fraction

## END_SUB_TUTORIAL

def display_trajectory(self, plan):
    # Copy class variables to local variables to make the web tutorials
more clear.
    # In practice, you should use the class variables directly unless you
have a good
    # reason not to.
    robot = self.robot
    display_trajectory_publisher = self.display_trajectory_publisher

    ## BEGIN_SUB_TUTORIAL display_trajectory
    ##
    ## Displaying a Trajectory
    ## ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    ## You can ask RViz to visualize a plan (aka trajectory) for you. But
the
    ## group.plan() method does this automatically so this is not that
useful
    ## here (it just displays the same trajectory again):
    ##
    ## A `DisplayTrajectory`_ msg has two primary fields, trajectory_start
and trajectory.

```

```

    ## We populate the trajectory_start with our current robot state to
copy over
    ## any AttachedCollisionObjects and add our plan to the trajectory.
    display_trajectory = moveit_msgs.msg.DisplayTrajectory()
    display_trajectory.trajectory_start = robot.get_current_state()
    display_trajectory.trajectory.append(plan)
    # Publish
    display_trajectory_publisher.publish(display_trajectory)

    ## END_SUB_TUTORIAL

def execute_plan(self, plan):
    # Copy class variables to local variables to make the web tutorials
more clear.
    # In practice, you should use the class variables directly unless you
have a good
    # reason not to.
    move_group = self.move_group

    ## BEGIN_SUB_TUTORIAL execute_plan
    ##
    ## Executing a Plan
    ## ^^^^^^^^^^^^^^^^^^^^^
    ## Use execute if you would like the robot to follow
    ## the plan that has already been computed:
    move_group.execute(plan, wait=True)

    ## **Note:** The robot's current joint state must be within some
tolerance of the
    ## first waypoint in the `RobotTrajectory`_ or ``execute()`` will fail
    ## END_SUB_TUTORIAL

def wait_for_state_update(
    self, box_is_known=False, box_is_attached=False, timeout=4
):
    # Copy class variables to local variables to make the web tutorials
more clear.
    # In practice, you should use the class variables directly unless you
have a good
    # reason not to.
    box_name = self.box_name
    scene = self.scene

    ## BEGIN_SUB_TUTORIAL wait_for_scene_update
    ##
    ## Ensuring Collision Updates Are Received
    ## ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```

```

        ## If the Python node dies before publishing a collision object update
        message, the message
        ## could get lost and the box will not appear. To ensure that the
        updates are
        ## made, we wait until we see the changes reflected in the
        ## ``get_attached_objects()`` and ``get_known_object_names()`` lists.
        ## For the purpose of this tutorial, we call this function after
        adding,
        ## removing, attaching or detaching an object in the planning scene.
        We then wait
        ## until the updates have been made or ``timeout`` seconds have passed
        start = rospy.get_time()
        seconds = rospy.get_time()
        while (seconds - start < timeout) and not rospy.is_shutdown():
            # Test if the box is in attached objects
            attached_objects = scene.get_attached_objects([box_name])
            is_attached = len(attached_objects.keys()) > 0

            # Test if the box is in the scene.
            # Note that attaching the box will remove it from known_objects
            is_known = box_name in scene.get_known_object_names()

            # Test if we are in the expected state
            if (box_is_attached == is_attached) and (box_is_known ==
is_known):
                return True

            # Sleep so that we give other threads time on the processor
            rospy.sleep(0.1)
            seconds = rospy.get_time()

        # If we exited the while loop without returning then we timed out
        return False
    ## END_SUB_TUTORIAL

    def add_box(self, timeout=4):
        # Copy class variables to local variables to make the web tutorials
        more clear.
        # In practice, you should use the class variables directly unless you
        have a good
        # reason not to.
        box_name = self.box_name
        scene = self.scene

        ## BEGIN_SUB_TUTORIAL add_box
        ##
        ## Adding Objects to the Planning Scene
        ## ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```



```

    return all_close(joint_goal, current_joints, 0.01)

def main():
    try:
        print("")
        print("-----")
        print("Welcome to the MoveIt MoveGroup Python Interface")
        print("-----")
        print("Press Ctrl-D to exit at any time")
        print("")
        input(
            "===== Press `Enter` to begin the application by setting up
the moveit_commander ..."
        )
        tutorial = MoveGroupPythonInterfaceTutorial()
        x_coord, y_coord, z_coord = processing_image()
        print("X: ", x_coord[0])
        print("Y: ", y_coord[0])
        print("Z: ", z_coord[0])

        input("===== Press `Enter` to attach the floor and walls to the
robot ...")
        tutorial.go_to_joint_end()
        tutorial.add_box()
        tutorial.attach_box()
        collision = CollisionSceneExample()
        collision.add_all()

        input("===== Press `Enter` to execute a movement using a joint
state goal ...")
        tutorial.go_to_joint_state()

        input("===== Press `Enter` to execute a movement using a pose
goal ...")
        tutorial.go_to_pose_goal()

        input("===== Press `Enter` to plan and execute a path with an
attached collision object ...")
        cartesian_plan, fraction = tutorial.plan_cartesian_path()
        tutorial.display_trajectory(cartesian_plan)
        tutorial.execute_plan(cartesian_plan)

        input("===== Press `Enter` to execute a movement using a joint
end position ...")
        tutorial.detach_box()
        tutorial.remove_box()
        tutorial.go_to_joint_end()

```

```

        print("===== Process completed!")
    except rospy.ROSInterruptException:
        return
    except KeyboardInterrupt:
        return

if __name__ == "__main__":
    main()

```

8.2 Catkin photo

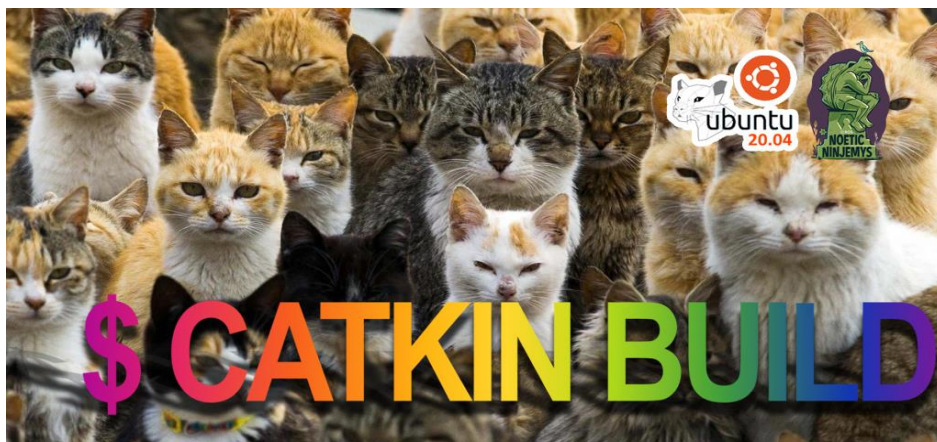


Figure 15: Cats assamble