İstanbul Teknik Üniversitesi

Elektrik-Elektronik Fakültesi

KON421E – Robot Control

Homework 2

Prof. Dr. Ovsanna Seta Estrada

TA: Aykut Özdemir

Student:

Oğuzhan KÖSE

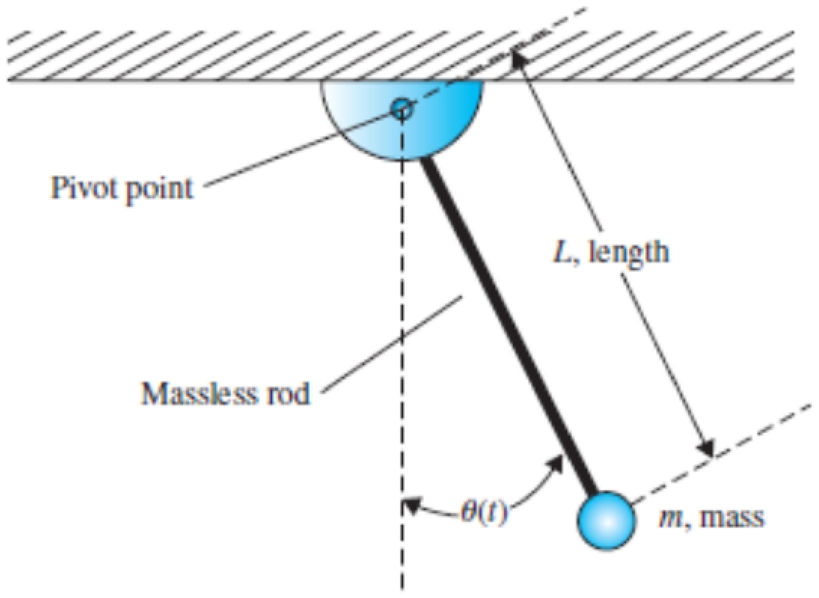
040160457

11.06.2021

Contents

1 System Overview

In this project, we are expected to create a single-link arm simulation in ROS/Gazebo environment and control the system. Example system model can be seen figure below.



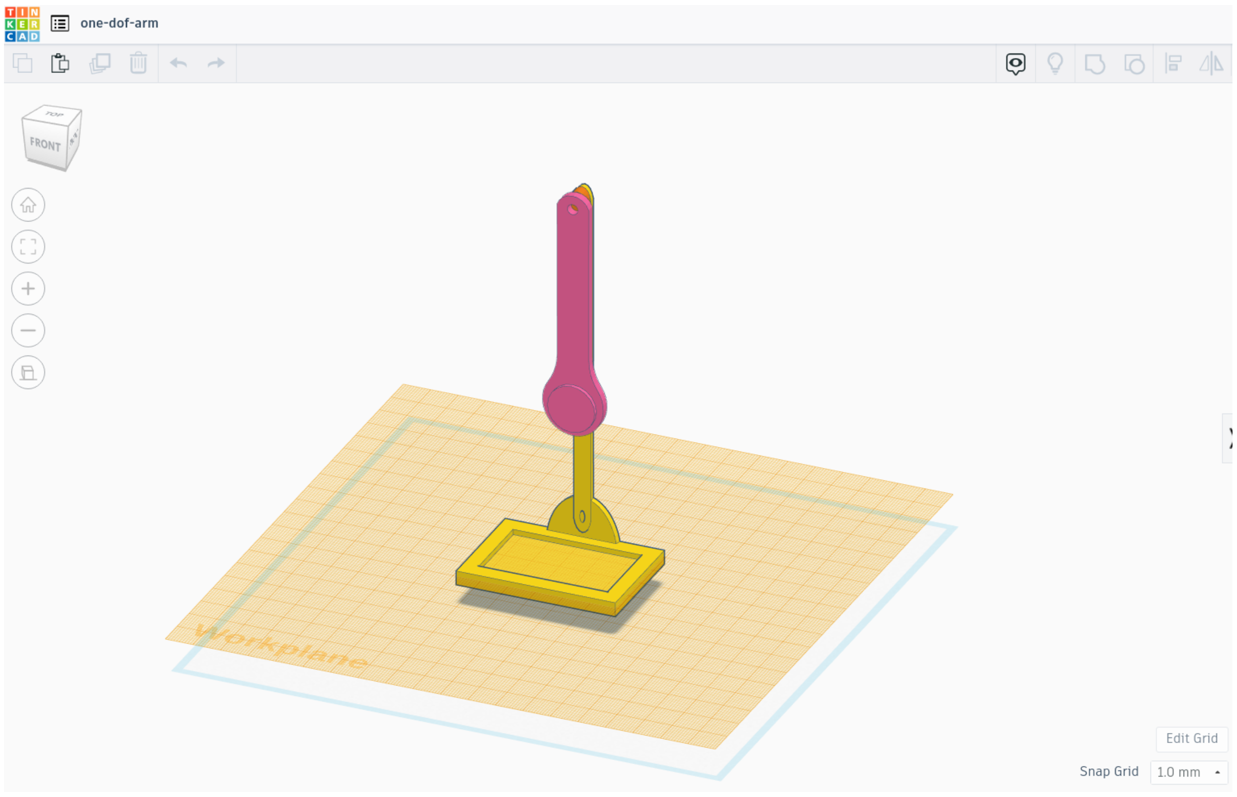
The arm is considered to be driven by a hysteresis current controlled H-Bridge and PMDC. Since Gazebo does not supports motor elements directly. Instead of adding a plugin for motor part, motor is simulated by a ROS node.

In this project, in addition to Gazebo and ROS 3 main software is used mainly for modelling.

1. TinkerCAD : A online CAD drawing software. In this project it is used for drawing arm itself.
2. Blender : An open-source software which is mainly used for animation and rendering purposes. Because of it is easily script-able architecture it is also used in robotics applications.
3. Phobos : An add-on for the open-source 3D modeling software Blender that enables the creation of robot models for use in robot frameworks like ROS and ROCK or in real-time simulations such as MARS or Gazebo. Phobos is used for creating robot URDF file.

2 Modelling of the System

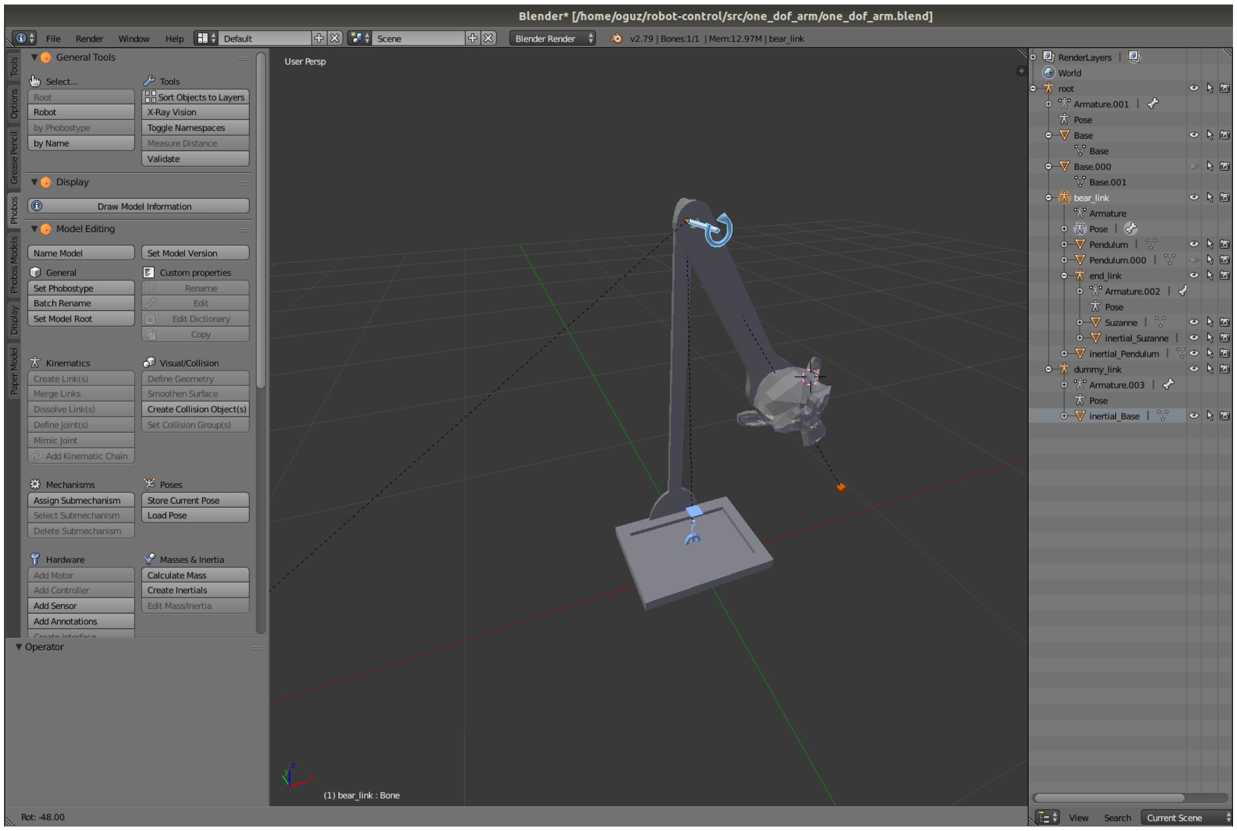
As the first step of modelling the one dof arm, arm is created by TinkerCAD in online environment. A basic design with only one joint is choosen. The design can be seen figure below.



The design is exported as STL file to the Blender and a as mascot monkey "Suzanne" is added to the tip point of the arm. After designing arm, as an add-on to Blender Phobos package is installed and configured. Using Phobos, necessary links and joints, joint types and joint constraints are defined. Up to this step design was used only for visual purposes. Then using geometric shapes of the parts, inertia of the parts is calculated from masses. And lastly collision objects of the parts are defined as boxes in order to decrease processing time of the simulation.

For exporting model Phobos has different options, such as urdf, sdf etc. For model file URDF and for the meshes STL file types are chosen. The GUI of Phobos and the design environment can be seen in figure below.

As PMDC motor model, with the help of Phobos, bear_joint parameters are edited to simulate a PMDC motor model. This way, a PMDC motor model is not created but joint dynamics simulates the behaviour of a PMDC motor and similar results to a PMDC motor can be obtained easily.



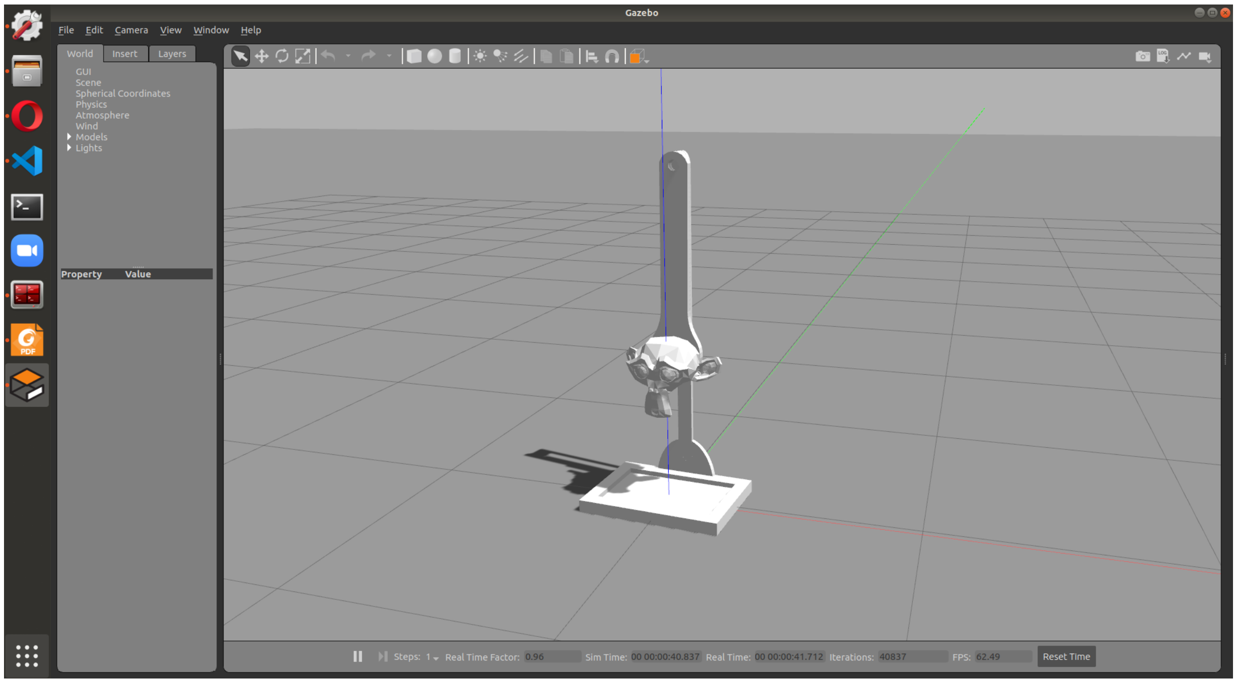
The exported URDF file can be seen below.

```xml
<?xml version="1.0"?>
<!-- created with Phobos 1.0.1 "Capricious Choutengan" -->
  <robot name="ppend">

    <link name="bear_link">
      <inertial>
        <origin xyz="0.03534 -1.38324 0.19726" rpy="0 0 0"/>
        <mass value="2.0"/>
        <inertia ixx="1.22159" ixy="-0.03965" ixz="-0.00672"
                          iyy="1.19396" iyz="-0.18344" izz="0.03565"/>
      </inertial>
      <visual name="Pendulum">
        <origin xyz="0.03534 -1.38324 0.19726" rpy="-1.5708 0 0"/>
        <geometry>
          <mesh filename="package://one_dof_arm/meshes/dae/Pendulum.dae"
                            scale="1.0 1.0 1.0"/>
        </geometry>
      </visual>
      <collision name="Pendulum.000">
        <origin xyz="0.04111 -1.37972 0.22758" rpy="-1.5708 0 0"/>
        <geometry>
        <mesh filename="package://one_dof_arm/meshes/dae/Pendulum.001.dae"
                            scale="1.0 1.0 1.0"/>
        </geometry>
      </collision>
    </link>

    <link name="dummy_link">
      <inertial>
        <origin xyz="0.01676 -0.05476 -0.089" rpy="0 0 0"/>
        <mass value="50.0"/>
        <inertia ixx="12.5678" ixy="0.40987" ixz="0.12657"
                            iyy="12.21951" iyz="-2.43533" izz="1.71693"/>
      </inertial>
    </link>

    <link name="end_link">
      <inertial>
        <origin xyz="0.00323 -0.04129 0.10447" rpy="0 0 0"/>
        <mass value="0.001"/>
        <inertia ixx="2e-05" ixy="0" ixz="0" iyy="2e-05"
                            iyz="0" izz="2e-05"/>
      </inertial>
      <visual name="Suzanne">
        <origin xyz="0.00323 -0.04129 0.10447" rpy="-1.5708 0 0"/>
        <geometry>
          <mesh filename="package://one_dof_arm/meshes/dae/Suzanne.dae"
                            scale="1.0 1.0 1.0"/>
        </geometry>
      </visual>
    </link>

    <link name="root">
      <visual name="Base">
        <origin xyz="0.01464 -0.0536 -0.0803" rpy="0 0 0"/>
        <geometry>
          <mesh filename="package://one_dof_arm/meshes/dae/Base.dae"
                            scale="1.0 1.0 1.0"/>
        </geometry>
```

```xml
      </visual>
      <collision name="Base.000">
        <origin xyz="-0.01032 -0.01632 -0.08093" rpy="0 0 0"/>
        <geometry>
          <mesh filename="package://one_dof_arm/meshes/dae/Base.001.dae"
                            scale="1.0 1.0 1.0"/>
        </geometry>
      </collision>
    </link>

    <joint name="bear_link" type="revolute">
      <origin xyz="-0.0207 0.14365 1.30294" rpy="1.5708 0 0"/>
      <parent link="root"/>
      <child link="bear_link"/>
      <axis xyz="0 0 1.0"/>
      <limit lower="-1.5708" upper="1.5708" effort="1000.0" velocity="0"/>
    </joint>

    <joint name="dummy_link" type="fixed">
      <origin xyz="-0.00212 0.00116 0.0087" rpy="0 0 0"/>
      <parent link="root"/>
      <child link="dummy_link"/>
    </joint>

    <joint name="end_link" type="revolute">
      <origin xyz="0.00225 -0.74383 0.03795" rpy="0 0 0"/>
      <parent link="bear_link"/>
      <child link="end_link"/>
      <axis xyz="0 0 1.0"/>
      <limit lower="0" upper="2e-05" effort="0" velocity="0"/>
    </joint>
</robot>
```

3 Creating Simulation Environment

With the help of the URDF file of the robot that is generated by Phobos, an empty gazebo world is created to keep computational costs low. For that purpose a launch file is written. The launch file is responsible for publishing link and joint states, opening gazebo environment and spawning the robot in the world. The robot in Gazebo environment and the launch file can be seen below.

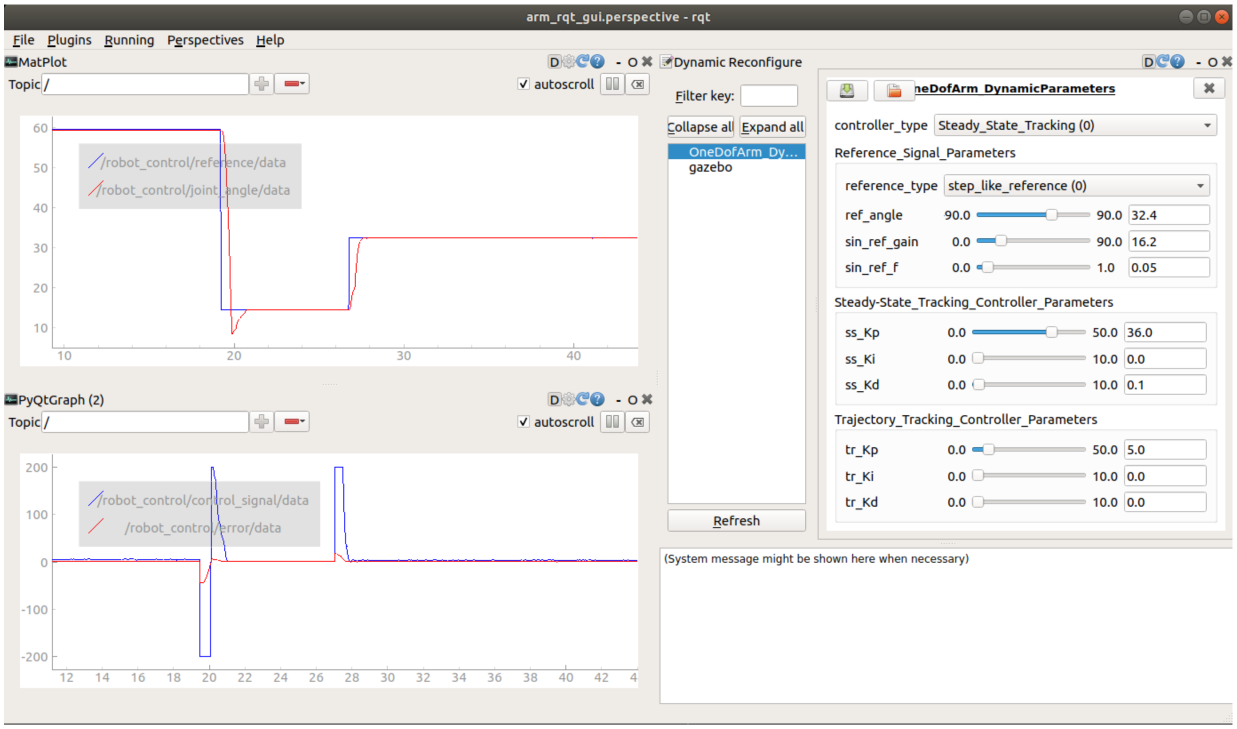


Steps: 1 Real Time Factor: 0.96 Sim Time: 00 00:00:40.837 Real Time: 00 00:00:41.712 Iterations: 40837 FPS: 62.49

```xml
<launch>
<!-- When calling the launch file you can provide the .urdf file
that you want to look at as a parameter.
Example: roslaunch your_pkg_name inspect_urdf.launch
model:='£(find your_pkg_name)/path/to/your_urdf.urdf' -->

<arg name="model_config" default="$(find one_dof_arm)/urdf/ppend.urdf"/>
<arg name="rvizconfig" default="$(find one_dof_arm)/rviz/urdf.rviz" />

<param name="robot_description" command="$(find xacro)/xacro
                                --inorder $(arg model_config)" />
  <!-- Start Gazebo. -->
<param name="/use_sim_time" value="false"/>

```

```
<node name="joint_state_publisher" pkg="joint_state_publisher"
                                type="joint_state_publisher" />

<node name="robot_state_publisher" pkg="robot_state_publisher"
                                type="robot_state_publisher"/>

<!-- Spawn a robot into Gazebo -->
<node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
  args="-file $(arg model_config) -urdf -z 0.05 -model one_dof"/>

<include file="$(find gazebo_ros)/launch/empty_world.launch" >
        <arg name="world_name" value="worlds/empty_world.world"/>
</include>

<node name="OneDofArm_DynamicParameters" pkg="dynamic_parameters"
                                                type="server.py"/>
</launch>
```

In the project, a rqt_gui that will dynamically change controller parameters and set-points and monitor the situation of the robot is requested. For that purpose launch file also starts dynamic parameter server of the robot which will be used by rqt_gui later in the project.


arm_rqt_gui.perspective - rqt
File Plugins Running Perspectives Help
MatPlot
Topic /
autoscroll
/robot_control/reference/data
/robot_control/joint_angle/data
PyQtGraph (2)
Topic /
autoscroll
/robot_control/control_signal/data
/robot_control/error/data
Dynamic Reconfigure
Filter key:
Collapse all
Expand all
OneDofArm_Dy...
gazebo
Refresh
neDofArm_DynamicParameters

controller_type Steady_State_Tracking (0)

Reference_Signal_Parameters

reference_type	step_like_reference (0)		
ref_angle	90.0	90.0	32.4
sin_ref_gain	0.0	90.0	16.2
sin_ref_f	0.0	1.0	0.05

Steady-State_Tracking_Controller_Parameters

ss_Kp	0.0	50.0	36.0
ss_Ki	0.0	10.0	0.0
ss_Kd	0.0	10.0	0.1

Trajectory_Tracking_Controller_Parameters

tr_Kp	0.0	50.0	5.0
tr_Ki	0.0	10.0	0.0
tr_Kd	0.0	10.0	0.0

(System message might be shown here when necessary)

Also for creating reference signal for trajectory generation a ROS node is written. The node is responsible for watching parameter changes in the server and publishing the new reference signal with respect to the new parameters.

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import Float32
import time
import math
import numpy as np
import dynamic_reconfigure.client

class ReferenceSignalPublisher():
    def __init__(self, topic_name="/robot_control/reference"):

        self.ref_pub = rospy.Publisher("/robot_control/reference",
                                    Float32, queue_size=100)
        client = dynamic_reconfigure.client.Client("/OneDofArm_DynamicParameters",
                                timeout=30, config_callback=self.param_callback)

        # Initial values of parameters
        self.reference_type = "step"
        self.ref_angle = 0.0
        self.sin_ref_gain = 1.0
        self.sin_ref_f = 1.0
        self.start = start = time.time()

    def param_callback(self, config):
        if '0' == "{reference_type}".format(**config):
            self.reference_type = "step"
        else:
            self.reference_type = "trajectory"

        self.ref_angle = float("{ref_angle}".format(**config))
        self.sin_ref_gain = float("{sin_ref_gain}".format(**config))
        self.sin_ref_f = float("{sin_ref_f}".format(**config))

        #print(self.reference_type, self.ref_angle,
                                    self.sin_ref_gain, self.sin_ref_f)

    def pub_ref(self):
        if self.reference_type == "step":
            self.ref_pub.publish(self.ref_angle)
        elif self.reference_type == "trajectory":
            u_t = np.sin(2*np.pi*self.sin_ref_f*
                                (time.time()-self.start))*self.sin_ref_gain
            self.ref_pub.publish(u_t)
        else:
            print("ERROR! Unknown Reference Type!!")

if __name__ == '__main__':

    rospy.init_node("ref_publisher_node")
    publisher = ReferenceSignalPublisher()
    rate = rospy.Rate(50)

    while not rospy.is_shutdown():
        publisher.pub_ref()
        rate.sleep()
```

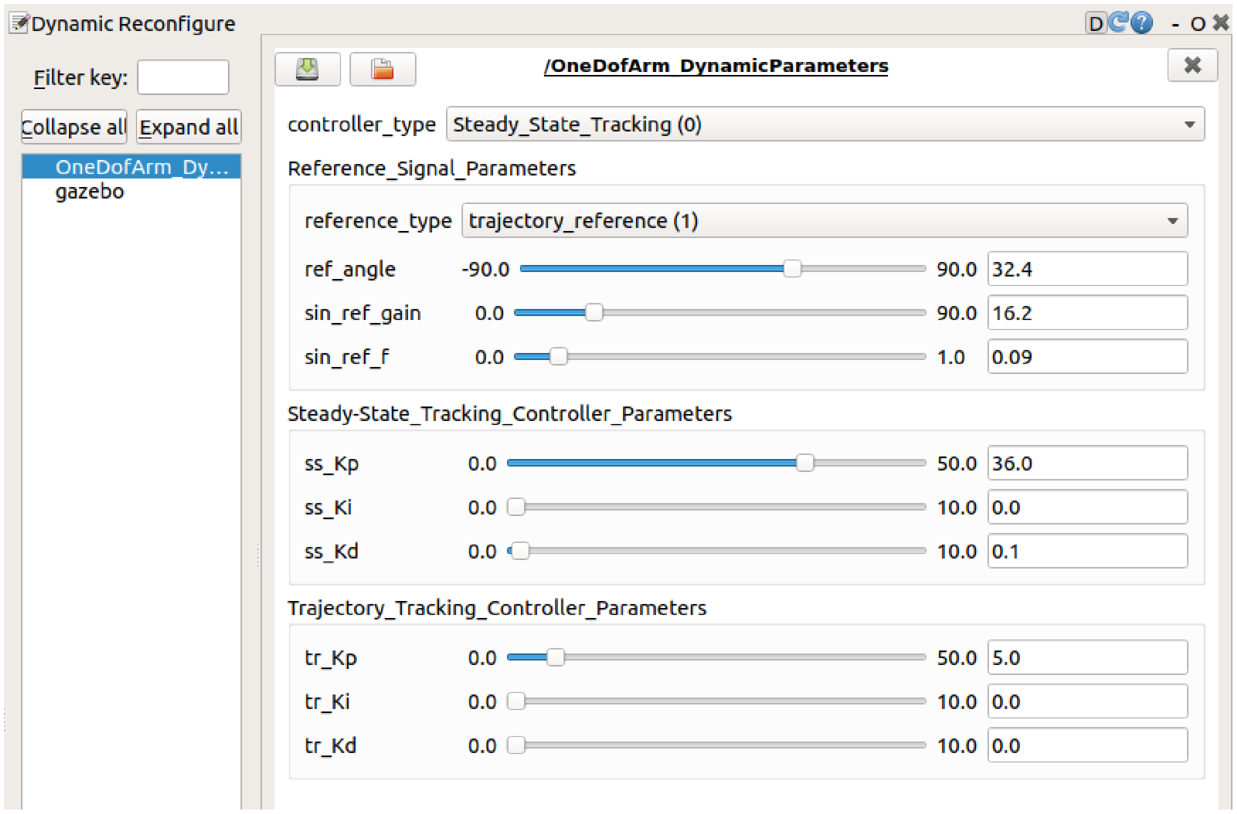
4 Control of The System

The one dof arm system is a nonlinear model. But within a feasible range of working angles it can be controlled by classical control structures like PID controller. Mainly within this project, two different controller are designed with respect to the use cases for two different use cases. These controllers and other necessary elements of the project is implemented in the main ROS node.

4.1 Main Node of One Dof Arm

Main node handles publishing necessary datas, getting controller parameters, creating control signals and applying control. As the controllers two PID type controller is implemented in the main node. From rqt_gui user can switch between these controllers easily. The controllers also have integral windup protection and control signal is limited by a saturation value. The difference between these two controllers is that one of them is tuned for set-point tracking control and the other one is tuned for trajectory tracking control. Also user can switch between two different reference signal types from rqt_gui. Amplitude and frequency of the reference signal will be changed accordingly.

The code of the main node can be seen below.


Dynamic Reconfigure

Filter key:

Collapse all | Expand all

OneDofArm_Dy...
gazebo

/OneDofArm_DynamicParameters

controller_type: Steady_State_Tracking (0)

Reference_Signal_Parameters

reference_type: trajectory_reference (1)

ref_angle	-90.0	90.0	32.4
sin_ref_gain	0.0	90.0	16.2
sin_ref_f	0.0	1.0	0.09

Steady-State_Tracking_Controller_Parameters

ss_Kp	0.0	50.0	36.0
ss_Ki	0.0	10.0	0.0
ss_Kd	0.0	10.0	0.1

Trajectory_Tracking_Controller_Parameters

tr_Kp	0.0	50.0	5.0
tr_Ki	0.0	10.0	0.0
tr_Kd	0.0	10.0	0.0

```python
#!/usr/bin/env python
import rospy
import gazebo_ros
from gazebo_msgs.srv import ApplyJointEffort, JointRequest, GetJointProperties
from gazebo_msgs.msg import LinkStates
from std_msgs.msg import Float32
import time
import math
import tf
import dynamic_reconfigure.client


class ArmControlNode():

    def __init__(self, joint_name = "bear_link"):

        self.joint_name = joint_name

        rospy.Subscriber("/robot_control/reference", Float32, self.ref_cb)

        self.control_pub = rospy.Publisher("/robot_control/control_signal",
                                            Float32, queue_size=100)
        self.ang_pub = rospy.Publisher("/robot_control/joint_angle",
                                            Float32, queue_size=100)
        self.error_pub = rospy.Publisher("/robot_control/error",
                                            Float32, queue_size=100)

        rospy.wait_for_service('/gazebo/apply_joint_effort')
        rospy.wait_for_service('/gazebo/clear_joint_forces')
        rospy.wait_for_service('/gazebo/get_joint_properties')

        self.effort_service = rospy.ServiceProxy('/gazebo/apply_joint_effort',
                                                            ApplyJointEffort)
        self.clear_effort = rospy.ServiceProxy('/gazebo/clear_joint_forces',
                                                            JointRequest)
        self.get_angle_srv = rospy.ServiceProxy('/gazebo/get_joint_properties',
                                                            GetJointProperties)

        client = dynamic_reconfigure.client.Client("/OneDofArm_DynamicParameters",
                            timeout=30, config_callback=self.param_callback)
        # initialization of default parameters
        self.ref_angle = 0
        self.Kp = 8.0
        self.Ki = 1.2
        self.Kd = 1.0
        self.windup_guard = 20
        self.lim_control = 200
        self.current_time = time.time()
        self.last_time = self.current_time
        self.PTerm = 0.0
        self.ITerm = 0.0
        self.DTerm = 0.0
        self.last_error = 0.0

    def ref_cb(self, msg):
        self.ref_angle = float(msg.data)

    def param_callback(self, config):
        c_type = float("{controller_type}".format(**config))
        if c_type == 0:
            self.Kp, self.Ki, self.Kd = float("{ss_Kp}".format(**config)),
                float("{ss_Ki}".format(**config)), float("{ss_Kd}".format(**config))
        elif c_type == 1:
            self.Kp, self.Ki, self.Kd = float("{tr_Kp}".format(**config)),
                float("{tr_Ki}".format(**config)), float("{tr_Kd}".format(**config))
        else:
            print("ERROR! Unknown Controller Type!!")

    def get_control_signal(self, feedback_value, setpoint, current_time=None):

        error = setpoint - feedback_value
        self.current_time = time.time()
        delta_time = self.current_time - self.last_time
        delta_error = error - self.last_error
        self.PTerm = self.Kp * error
        self.ITerm += error * delta_time

        if (self.ITerm < -self.windup_guard):
            self.ITerm = -self.windup_guard
        elif (self.ITerm > self.windup_guard):
            self.ITerm = self.windup_guard

        self.DTerm = 0.0
        if delta_time > 0:
            self.DTerm = delta_error / delta_time

        # Remember last time and last error for next calculation
        self.last_time = self.current_time
        self.last_error = error
        self.output = self.PTerm + (self.Ki * self.ITerm) +
                                        (self.Kd * self.DTerm)
        self.output = max(-self.lim_control,
                          min(self.lim_control, self.output))
        return self.output

    def make_effort(self):

        error = self.ref_angle - self.joint_angle
        effort = self.get_control_signal(self.joint_angle, self.ref_angle)
        self.error_pub.publish(error)
        self.control_pub.publish(effort)
        start_time = rospy.Duration.from_sec(0)
        duration = rospy.Duration.from_sec(0.1)
```

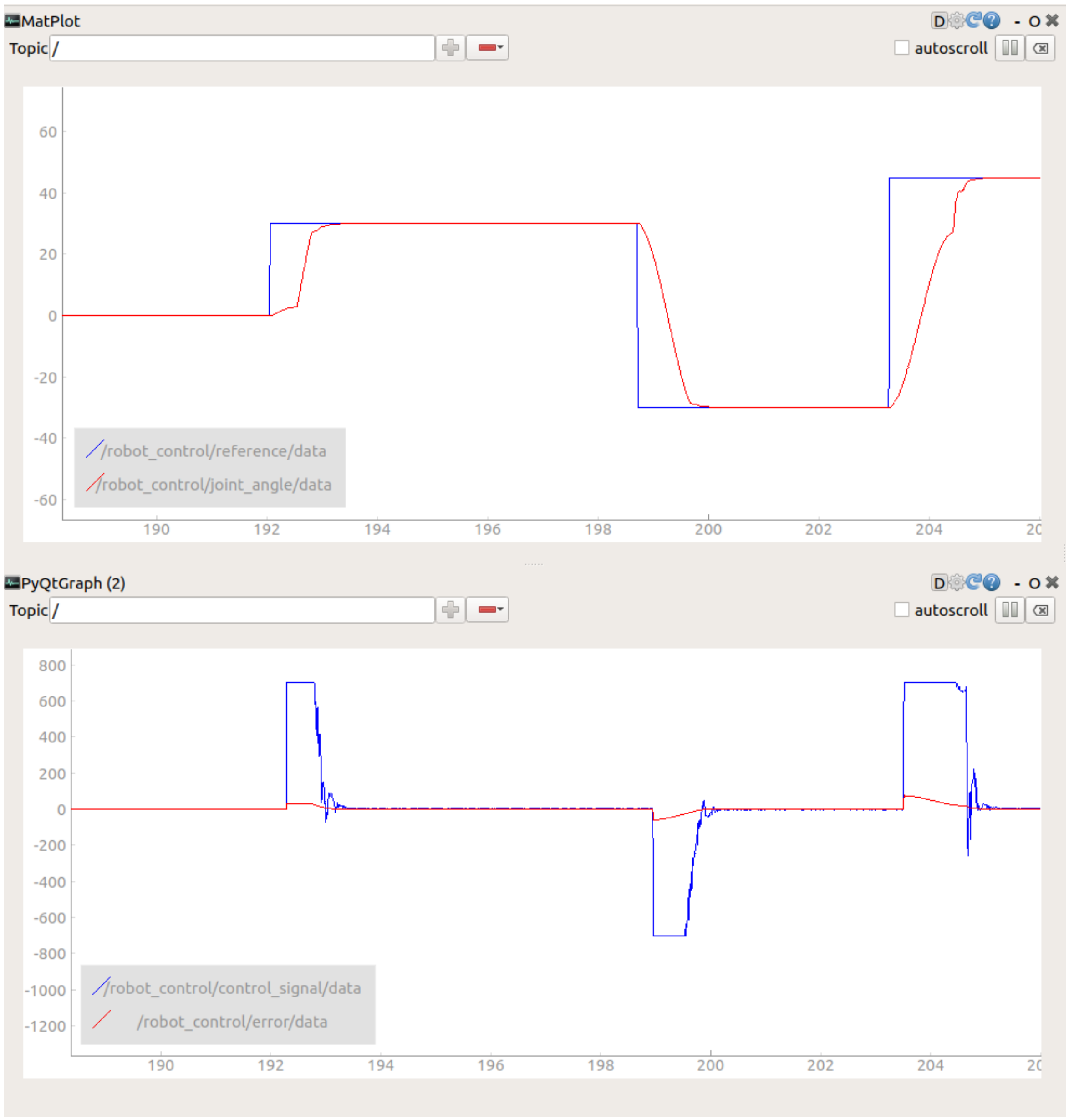
```python
        return self.effort_service(self.joint_name, effort,
                                   start_time, duration)

    def clear_effort(self):
        return self.clear_effort(self.joint_name)

    def get_joint_angle(self):
        self.joint_angle = math.degrees(
                    self.get_angle_srv(self.joint_name).position[0])
        self.joint_der = math.degrees(
                    self.get_angle_srv(self.joint_name).rate[0])
        self.ang_pub.publish(self.joint_angle)
        return self.joint_angle


if __name__ == '__main__':

    rospy.init_node("arm_controller_node")

    controller = ArmControlNode()

    while not rospy.is_shutdown():

        controller.get_joint_angle()
        controller.make_effort()
```

4.2 Set-point Tracking Control

The set-point controller of the system is designed to response fast and eliminate steady-state errors so, as set-point controller of the system a PD type controller is designed and implemented. **The controller coefficients are selected as follows:**

$$K_P = 37.5, K_D = 2.8 \tag{1}$$

And the set-point tracking performance of the controller can be seen below.

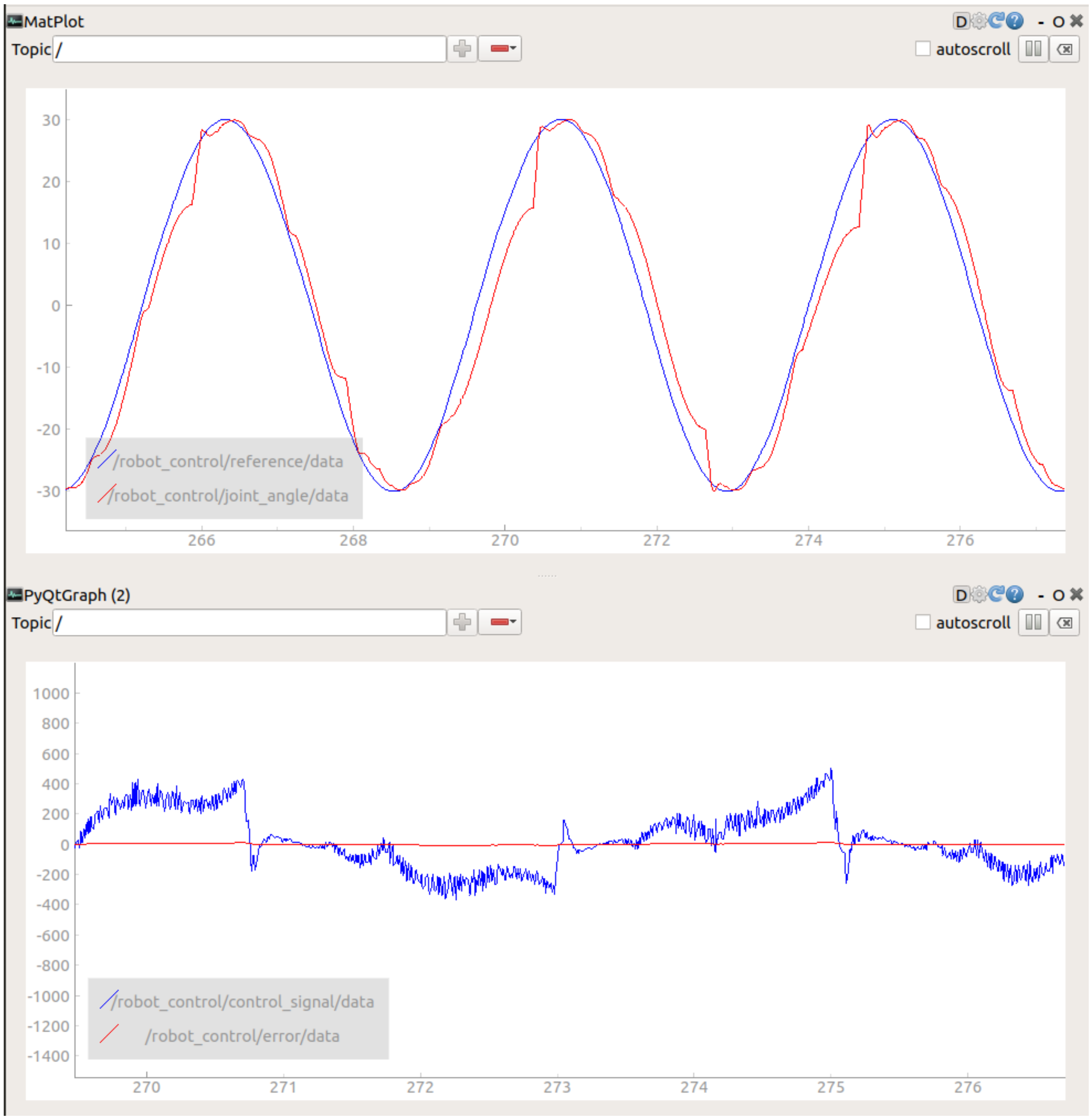


4.3 Trajectory Tracking Control

The trajectory tracking controller of the system is designed to response fast and decrease the overall error for a sinusoidal reference signal so, as trajectory tracking controller of the system a PID type controller is designed and implemented. The reference signal used for tuning the controller is sinusoidal signal with magnitude of 30 degrees and 0.2 Hz frequency. **The controller coefficients are selected as follows:**

$$K_P = 38.5, K_I = 0.3, K_D = 1.3 \tag{2}$$

And the set-point tracking performance of the controller can be seen below.



5 Testing and Results

For ease of usage two bash scripts are written in addition to these nodes.

- ./spawn_model.sh : This script runs the gazebo and prepares the environment.
- ./control.sh : This script runs the controller and rqt_gui.

The usage of the project package and the results can be seen in YouTube link below.

YouTube : `https://youtu.be/9dCtwyUWioM`

6 References

1. Prof. Dr. Ovsanna Seta Estrada, Robot Control Slides, 2021
2. Phobos: A tool for creating complex robot models, von Szadkowski, 2020 [`https://github.com/dfki-ric/phobos`]
3. `http://wiki.ros.org/Documentation`
4. `http://wiki.ros.org/gazebo`
5. `http://wiki.ros.org/dynamic_reconfigure`