AMERICAN UNIVERSITY OF BEIRUT

Maroun Semaan Faculty of Engineering and Architecture
Department of Electrical and Computer Engineering

CR10 Robot Analysis and Control

EECE 661 Project
Final Report
Fall 2024-2025

by

Mostafa Kassem CCE
Ralph Al-Fata CCE

Professor Hussein Hussein

December 14, 2024

Contents

0.1 Introduction

The **CR10** is a collaborative robot (cobot) developed by **Fanuc**, a leading company in industrial automation and robotics. The CR10 is designed to work alongside humans safely and efficiently. Below are some of its key characteristics:

- **Payload Capacity:** 10 kg, making it suitable for medium load tasks.
- **Reach:** Approximately 1,420 mm, providing flexibility in workspace coverage.
- **Safety Features:** Equipped with advanced force sensors, allowing safe operation close to humans without the need for protective barriers.
- **Applications:** Ideal for assembly, material handling, packaging, and inspection tasks.
- **Ease of Use:** Simplified programming and deployment reduce the need for extensive training.

The CR10 combines *precision*, *flexibility*, and *safety*, making it an excellent choice for industries seeking automation solutions that maintain workplace safety.

The CR10 collaborative robot is used in sectors such as manufacturing, automotive, electronics, logistics and warehousing, food and beverage, pharmaceuticals, and metal fabrication.

In this report, we will present the different parts of the project, including Analytical Solving, Practicing URDF, and Robot Solver along with explaining the work done in each of them supported with photos and videos for better explanation.

The code is found in the following GitHub repository `https://github.com/mostafa-IK03/CR10-Robot-Analysis-and-Control-.git`.

0.2 Analytical Solving

This section involves using existing Python Libraries to perform forward kinematics, compute the Jacobian matrix, and develop the inverse kinematics model of the CR10 Robot.

0.2.1 Python Libraries

After conducting research on various python libraries to calculate the forward and inverse kinematics along with the Jacobian matrix, we found couple of libraries including Visual Kinematics, TriP, Robotics Toolbox for Python, PyKDL, robo_analyzer, SymPy Robotics, forwardkinematics, and pytorch-kinematics and Kinpy. However, some of the mentioned libraries can compute part of the requirements. Thus, we chose to move on with the Kinpy library [2] for its simplicity, supporting the calculation of both kinematics types and the Jacobian matrix, also for it supporting the use of URDF files.

0.2.2 Forward Kinematics

Forward kinematics calculates the position and orientation of the end-effector based on joint parameters. The Denavit-Hartenberg (DH) parameters provide a standardized way to describe the kinematic chain of a robot.

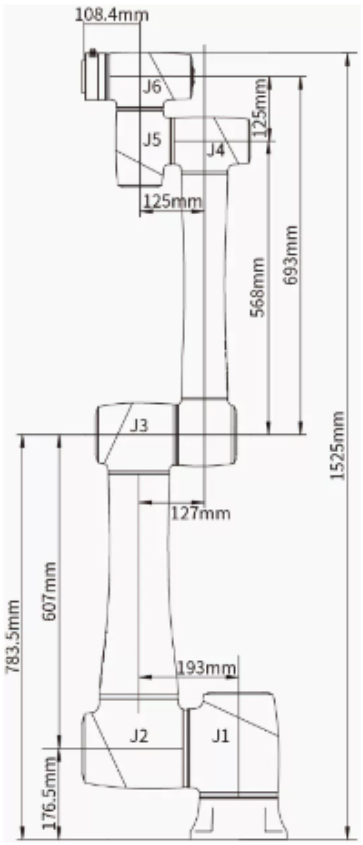

Figure 1: CR10 Robot Measurements [1]

We used Fig.[1] to derive the DH parameters shown in Fig.[2], to be able to find the forward kinematics.

	θ	**d**	**a**	α
1	θ_1	d_1	0	$+90°$
2	$\theta_2 + 90$	0	d_2	0
3	θ_3	0	d_3	0
4	$\theta_4 - 90$	d_4	0	$-90°$
5	θ_5	d_5	0	$+90°$
6	θ_6	d_6	0	0

Figure 2: DH Parameters

Where d1=176.5, d2= 607, d3=568, d4=191, d5=125, d6=111.4

Using the Kinpy library, we wrote a code that computes the forward kinematics given the DH parameters found before.[3][4]

```python
import numpy as np
import kinpy as kp

# Open the URDF file and read its content
with open("CR10URDF.urdf", "rb") as f:
    urdf_text = f.read()

# Build the serial chain using the URDF file content
chain = kp.build_serial_chain_from_urdf(urdf_text, "Link6")
print(chain)
print("Joint Parameter Names:", chain.get_joint_parameter_names())

# Define joint angles (example configuration)
th = [0.0, np.pi / 2.0, 0.0, -np.pi / 2.0, 0.0, 0.0]

# Compute forward kinematics
fk_result = chain.forward_kinematics(th, end_only=False)
print("\nForward Kinematics Result:")
print(fk_result)

# Extract the end-effector pose (position and orientation)
end_effector_name = "Link6"
end_effector_pose = fk_result[end_effector_name]

print("\nEnd-Effector Position:", end_effector_pose.pos)
print("End-Effector Orientation (Quaternion):", end_effector_pose.rot)
```

Figure 3: Forward Kinematics Code

```
└── Link1_frame
    └── Link2_frame
        └── Link3_frame
            └── Link4_frame
                └── Link5_frame
                    └── Link6_frame

Joint Parameter Names: ['joint1', 'joint2', 'joint3', 'joint4', 'joint5', 'joint6']

Forward Kinematics Result:
{'Link1': Transform(rot=[1. 0. 0. 0.], pos=[0.     0.     0.1765]), 'Link2': Transform(rot=[ 7.07106781e-01  7.07106781e-01  5.5
5111512e-17 -2.59734824e-06], pos=[0.     0.     0.1765]), 'Link3': Transform(rot=[ 7.07106781e-01  7.07106781e-01  5.55111512e-
17 -2.59734824e-06], pos=[-6.0700000e-01  2.2296355e-06  1.7650223e-01]), 'Link4': Transform(rot=[-3.89602236e-06 -1.29867412e-0
6  7.07106781e-01 -7.07106781e-01], pos=[-1.1750007  -0.19099568  0.17650432]), 'Link5': Transform(rot=[-1.83659581e-06 -3.67320
848e-06 -1.83660255e-06 -1.00000000e+00], pos=[-1.17499978 -0.19099568  0.30150432]), 'Link6': Transform(rot=[-3.89602236e-06 -1
.29867412e-06  7.07106781e-01 -7.07106781e-01], pos=[-1.17500018 -0.29939568  0.30150471])}

End-Effector Position: [-1.17500018 -0.29939568  0.30150471]
End-Effector Orientation (Quaternion): [-3.89602236e-06 -1.29867412e-06  7.07106781e-01 -7.07106781e-01]
```

Figure 4: Forward Kinematics Code Output

0.2.3 Inverse Kinematics

Inverse kinematics determines the joint parameters required to achieve a desired position and orientation of the end-effector. It is fundamental for trajectory planning and motion control in robotics.

Figures [5] and [6] show the code used to find the inverse kinematics and its output respectively.

```python
# Define the desired end-effector pose (same as the one obtained from FK for testing)
desired_pose = kp.Transform(rot=end_effector_pose.rot, pos=end_effector_pose.pos)

# Compute inverse kinematics
ik_solution = chain.inverse_kinematics(desired_pose)

if ik_solution is not None:
    print("\nInverse Kinematics Solution (Joint Angles):")
    for name, angle in zip(chain.get_joint_parameter_names(), ik_solution):
        print(f"{name}: {np.degrees(angle):.2f} degrees")
else:
    print("\nInverse kinematics did not converge to a solution.")
```

Figure 5: Inverse Kinematics Code

```
Inverse Kinematics Solution (Joint Angles):
joint1: -0.00 degrees
joint2: 67.19 degrees
joint3: 43.68 degrees
joint4: -68.19 degrees
joint5: -0.00 degrees
joint6: -42.67 degrees
```

Figure 6: Inverse Kinematics Code Output

0.2.4 Jacobian Matrix

The Jacobian matrix maps the joint velocities to the end-effector's linear and angular velocities. It is essential for determining the robot's motion and analyzing singularities.

Figures [7] and [8] show the code used to compute the Jacobian matrix and its output respectively.

```python
# Compute the Jacobian matrix at the current joint configuration
jacobian_matrix = chain.jacobian(th)

print("\nJacobian Matrix:")
print(jacobian_matrix)
```

Figure 7: Jacobian Matrix Code

```
Jacobian Matrix:
[[ 2.99395684e-01 -1.25004714e-01 -1.25002485e-01 -1.25000398e-01
   1.08400000e-01  0.00000000e+00]
 [-1.17500018e+00  4.59152101e-07  4.59152101e-07  4.59152100e-07
  -3.98175433e-07  0.00000000e+00]
 [-2.60901394e-16 -1.17499908e+00 -5.67999082e-01  9.18304201e-07
  -7.96350866e-07  0.00000000e+00]
 [ 2.54109884e-21 -3.67320510e-06 -3.67320510e-06 -3.67320510e-06
   7.34642370e-06 -3.67320510e-06]
 [-2.09374907e-16 -1.00000000e+00 -1.00000000e+00 -1.00000000e+00
   3.67319161e-06 -1.00000000e+00]
 [ 1.00000000e+00  1.34920963e-11  1.34920963e-11  1.34920963e-11
   1.00000000e+00  1.34920963e-11]]
```

Figure 8: Jacobian Matrix Code Output

0.3 Practicing URDF

This part of the report presents the part of the project which was constructing a CR10 description package in ROS2 using provided .stp files and an existing CR10 URDF package, enabling visualization in Rviz2, configuring joint control with the Joint States Publisher GUI, and developing a ROS2 node to calculate and publish the end effector's position in real-time using forward kinematics.

0.3.1 URDF Modeling

A **URDF (Unified Robot Description Format)** file is an XML-based format used to describe the structure, joints, and physical properties of a robot model. It defines the robot's links (rigid bodies), joints (connections between links), and their properties such as mass, inertia, and geometry. URDF files are commonly used in robotics software frameworks like **ROS** to simulate, visualize, and control robotic systems.

Rviz2 is a 3D visualization tool in ROS used to visualize robots, sensor data, and the environment. It can display information such as robot models, joint states, point clouds, and trajectories in real-time. Rviz is essential for debugging and verifying the performance of robotic systems by providing an interactive and graphical representation of the robot's state and environment.

For the first part of the project, we had to launch the CR10URDF.urdf file and visualize it on Rviz. For that, we found the official github repo for Dobot, the company that created the CR10 robot. We found the available launch file, in which there was the corresponding nodes that would allow us to visualize the robot. Here is a quick look into the code:

```python
def generate_launch_description():
    # mane = str(os.getenv("DOBOT_TYPE"))
    urdf_path = get_package_share_path('cr10_description')
    default_model_path = urdf_path / f'urdf/CR10URDF.urdf'
    default_rviz_config_path = urdf_path / 'rviz/urdf.rviz'

    gui_arg = DeclareLaunchArgument(name='gui', default_value='true', choices=['true', 'false'],
                                    description='Flag to enable joint_state_publisher_gui')
    model_arg = DeclareLaunchArgument(name='model', default_value=str(default_model_path),
                                      description='Absolute path to robot urdf file')
    rviz_arg = DeclareLaunchArgument(name='rvizconfig', default_value=str(default_rviz_config_path),
                                     description='Absolute path to rviz config file')

    robot_description = ParameterValue(Command(['xacro ', LaunchConfiguration('model')]),
                                       value_type=str)
```

Figure 9: First part of display.launch.py file

In Fig. [9], we start by declaring the arguments, and importing all the necessary paths to the rviz, and urdf files.

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

    # Depending on gui parameter, either launch joint_state_publisher or joint_state_publisher_gui
    joint_state_publisher_node = Node(
        package='joint_state_publisher',
        executable='joint_state_publisher',
        condition=UnlessCondition(LaunchConfiguration('gui'))
    )

    joint_state_publisher_gui_node = Node(
        package='joint_state_publisher_gui',
        executable='joint_state_publisher_gui',
        condition=IfCondition(LaunchConfiguration('gui'))
    )

    rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        name='rviz2',
        output='screen',
        arguments=['-d', LaunchConfiguration('rvizconfig')],
    )

    return LaunchDescription([
        gui_arg,
        model_arg,
        rviz_arg,
        joint_state_publisher_node,
        joint_state_publisher_gui_node,
        robot_state_publisher_node,
        rviz_node
    ])
```

Figure 10: Second part of display.launch.py file

In Fig. [10], we start the necessary nodes to have the visualization running. We have the node **joint_state_publisher** that allows us to control the joints of the robot, and we can see it live moving as shown in the video `https://drive.google.com/file/d/`

`1vDdTSZ3mXk-xHBQFgm_oDBxYjOtciAtD/view?usp=sharing`. Figure 11 shows the CR10 robot in rviz.

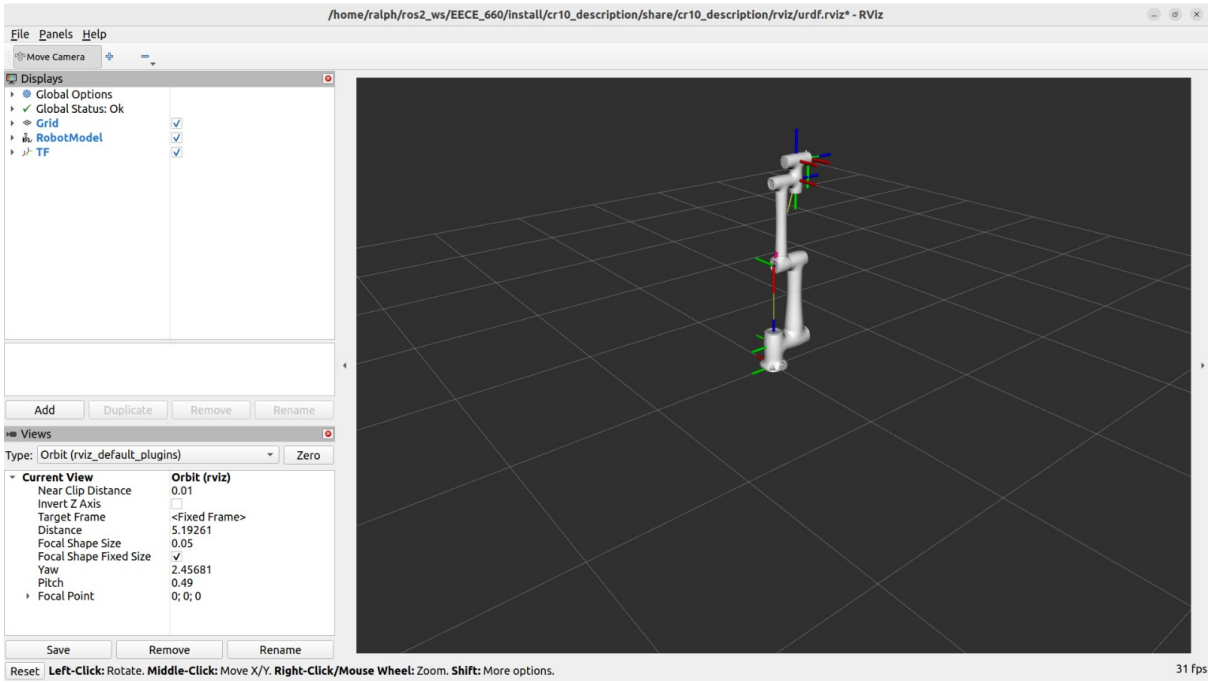

Figure 11: CR10 Robot shown in Rviz2

0.3.2 ROS2 Node

For the next part, the package **forward_kinematics** was implemented, in which there is a node named **ForwardKinematics**. Below is an explanation of the code.

```python
def __init__(self):
    super().__init__("ForwardKinematics")

    # Declare and retrieve the URDF_PATH parameter
    self.declare_parameter("urdf_path", "/home/ralph/ros2_ws/EECE_661/src/cr10_description/urdf/CR10URDF.urdf")
    urdf_path = self.get_parameter("urdf_path").get_parameter_value().string_value

    try:
        with open(urdf_path, "rb") as f:
            urdf_text = f.read()
        self.chain_ = kp.build_serial_chain_from_urdf(urdf_text, "Link6")
    except FileNotFoundError:
        self.get_logger().error(f"URDF file not found at path: {urdf_path}")
        raise

    # Subscriptions, publications, and timers
    self.subscription_ = self.create_subscription(JointState, '/joint_states', self.listener_callback, 10)
    self.publisher_ = self.create_publisher(JointInterface, '/end_effector_pos', 10)
    self.timer_ = self.create_timer(0.1, self.publish_end_effector_pos)  # Publish at 10 Hz

    # Message to store the pose of the end effector
    self.msg_pose_ = JointInterface()
```

Figure 12: Forward Kinematics code part 1

We start by defining the name of the node, we set the URDF file path, and pass it to the library **kinpy**, which we are using to calculate the forward kinematics. We then define our publishers and subscribers. We subscribe to the **/joint_states** topic, so we can get the current robot configuration, and we initiate the publisher **/end_effector_pos**, where we will publish the end effector position calculated from the forward kinematics.

```python
def listener_callback(self, msg):
    # Extract joint positions
    joint1 = msg.position[0]
    joint2 = msg.position[1]
    joint3 = msg.position[2]
    joint4 = msg.position[3]
    joint5 = msg.position[4]
    joint6 = msg.position[5]

    # Compute forward kinematics
    dh = [0.0 + joint1, np.pi / 2.0 + joint2, 0.0 + joint3, -np.pi / 2.0 + joint4, 0.0 + joint5, 0.0 + joint6]
    fk_result = self.chain_.forward_kinematics(dh, end_only=False)

    # Extract end effector position and orientation
    end_effector_name = "Link6"
    end_effector_pose = fk_result[end_effector_name]

    self.msg_pose_.x_position = end_effector_pose.pos[0]
    self.msg_pose_.y_position = end_effector_pose.pos[1]
    self.msg_pose_.z_position = end_effector_pose.pos[2]
    self.msg_pose_.x_rotation = end_effector_pose.rot[0]
    self.msg_pose_.y_rotation = end_effector_pose.rot[1]
    self.msg_pose_.z_rotation = end_effector_pose.rot[2]
    self.msg_pose_.w_rotation = end_effector_pose.rot[3]

def publish_end_effector_pos(self):
    # Publish the calculated end-effector position and orientation
    self.publisher_.publish(self.msg_pose_)
```

Figure 13: Forward Kinematics code part 2

In Fig. [13], we define the function callback, which constantly publishes the end effector position. The last function is defined to be able to periodically keep publishing the position.

The custom message **JointInterface** is dedicated to publish the position of the end effector. It can be seen in Fig. [14]. The rotation is expressed in quaternion.

```
float64 x_position
float64 y_position
float64 z_position
float64 x_rotation
float64 y_rotation
float64 z_rotation
float64 w_rotation
```

Figure 14: JointInterface custom message

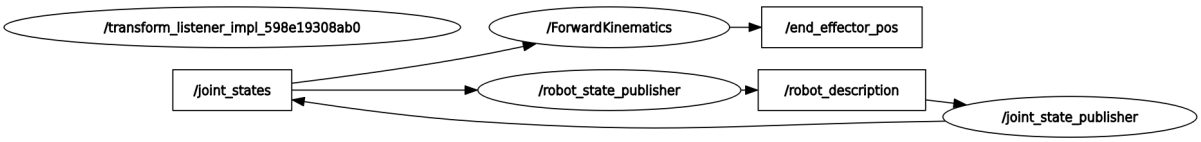

Figure 15: RQT Graph

A demo of this part is shown in the following video `https://drive.google.com/file/d/1vOp1uEBNMW6vmabAgAbqVawW2NCNHdQ9/view?usp=sharing`.

0.4 Robot Solver

This section of the report presents the project part of developing ROS2 [4] control model to command the CR10 robot in Rviz2 by moving the end effector between two points at a constant speed, integrating real-time visualization, analyzing the RQT graph, and demonstrating movement through parametric functions like circular paths.

0.4.1 Trajectory Planning

Trajectory planning is the process of determining a smooth, continuous path for the robot's end effector to move between specified points while maintaining constraints like speed, path shape, and kinematic limits.

Upon research done to find the best python library to perform trajectory planning we choose the Ruckig library [3] which is a real-time motion generation tool that calculates smooth, jerk-limited trajectories by providing velocity, acceleration, and jerk constraints on the joints of the robot.

By inputting the current and desired position, velocity and acceleration of the end-effector of the robot, Ruckig library find a smooth trajectory to move between those two points. Also, you can input optional intermediate waypoints and this can force the robot to follow some path shape of your choice.

Figure [16] shows the python code used importing the ruckig library to find the smooth path between two current and desired points, tested with one degree of freedom. The output of this code is plotted in Fig. [17], those plottings show how the trajectory is smooth without any jerky movements.

```python
import ruckig
import matplotlib.pyplot as plt

# Define the number of Degrees of Freedom (DoFs) and control cycle time
otg = ruckig.Ruckig(1, 0.01)  # 1 DoF, 10 ms control cycle
# Input parameters: current state, target state, and kinematic limits
input_param = ruckig.InputParameter(1)
input_param.current_position = [0.0]
input_param.current_velocity = [0.0]
input_param.current_acceleration = [0.0]
input_param.target_position = [1.0]
input_param.target_velocity = [0.0]
input_param.target_acceleration = [0.0]
input_param.max_velocity = [2.0]
input_param.max_acceleration = [2.0]
input_param.max_jerk = [10.0]
# Output parameters
output_param = ruckig.OutputParameter(1)
# Lists to store trajectory data for plotting
time_data = []
position_data = []
velocity_data = []
acceleration_data = []
# Generate the trajectory
print("Generating trajectory...")
while otg.update(input_param, output_param) == ruckig.Result.Working:
    time_data.append(output_param.time)
    position_data.append(output_param.new_position[0])
    velocity_data.append(output_param.new_velocity[0])
    acceleration_data.append(output_param.new_acceleration[0])
    output_param.pass_to_input(input_param)
print("Trajectory complete!")
```

Figure 16: Trajectory generation code using ruckig python library

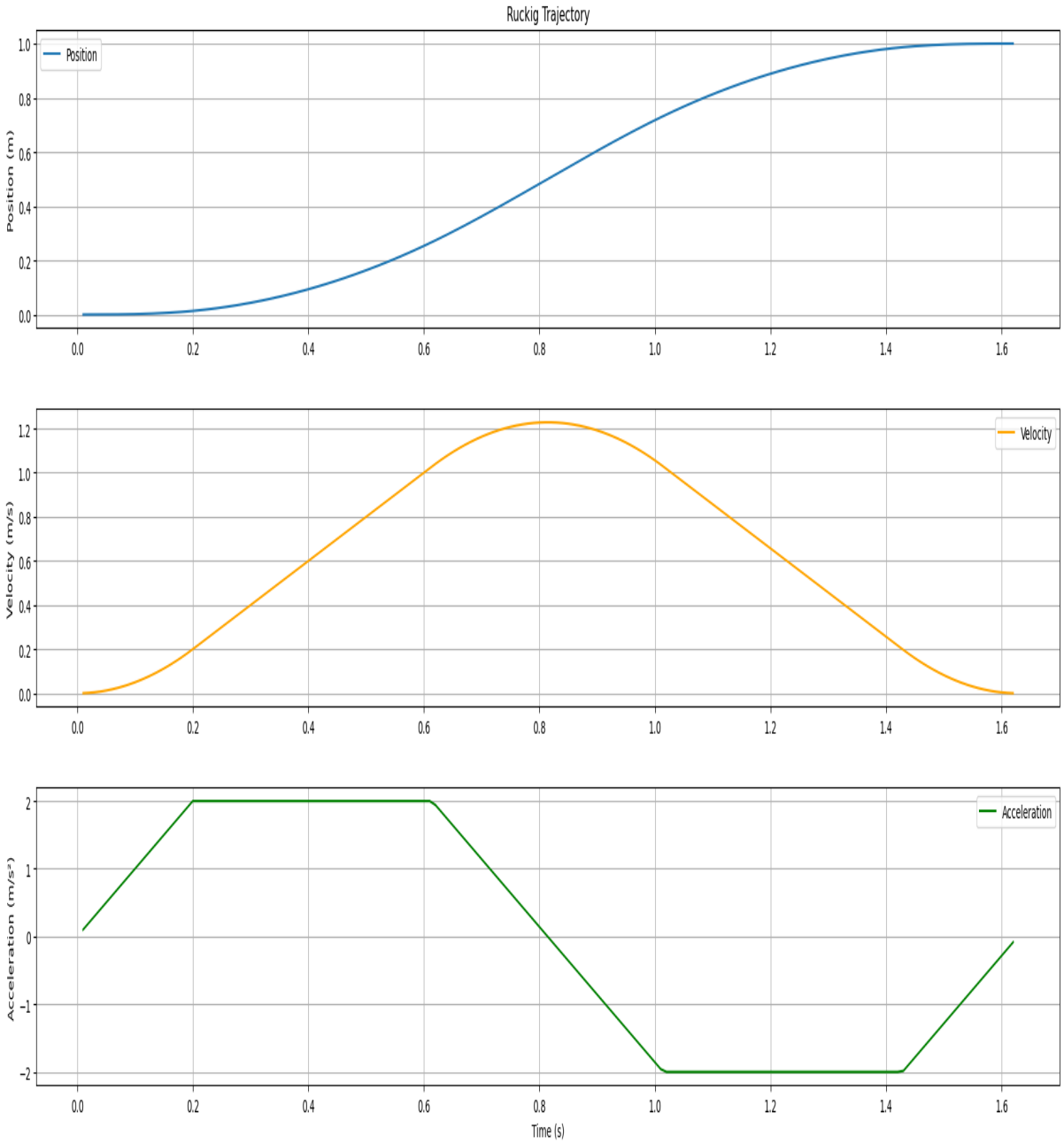

Figure 17: Position, velocity and acceleration of the trajectory between current and desired points

0.4.2 Method 1

This section presents the code and the rqt graph of finding the trajectory between two points A and B connected by a straight line trajectory.

We can see in Fig. [21] that the **Planner** node is subscribed to both the **/desired_pose** and **end_effector_pos** topics, and based on them the **Planner** generates the corresponding joint commands and publishes them to **/joint_states**. It also publishes the corresponding markers to **/trajectory_marker** in order to visualize the path on rviz.

Now, let's dive into the code related to the planner that works using linear interpolation.

```
def __init__(self):
    super().__init__("Planner")

    # Declare and retrieve the URDF_PATH parameter
    self.declare_parameter("urdf_path", "/home/ralph/ros2_ws/EECE_661/src/cr10_description/urdf/CR10URDF.urdf")
    urdf_path = self.get_parameter("urdf_path").get_parameter_value().string_value

    try:
        with open(urdf_path, "rb") as f:
            urdf_text = f.read()
        self.chain = kp.build_serial_chain_from_urdf(urdf_text, END_EFFECTOR_LINK)
    except FileNotFoundError:
        self.get_logger().error(f"URDF file not found at path: {urdf_path}")
        raise

    self.end_effector_pos = [0] * 7
    self.joint_trajectory = []

    self.subscriber_ = self.create_subscription(EndEffectorCommand, "/desired_pos", self.desired_position_callback, 10)
    self.subscriber_pose = self.create_subscription(JointInterface, "/end_effector_pos", self.update_position, 10)
    self.publisher_ = self.create_publisher(JointState, "/joint_states", 10)
    self.marker_publisher_ = self.create_publisher(Marker, "/trajectory_marker", 10)

    # Timer callback for periodic joint state publishing
    self.timer = self.create_timer(0.1, self.publish_joint_states)  # Publish at 10 Hz
```

Figure 19: Linear planner part 1

```
geometry_msgs/Point point_a
geometry_msgs/Point point_b
float64 linear_speed
```

Figure 18: EndEffectorCommand custom interface

First, we create a custom interface named **EndEffectorCommand**. It can be seen in Fig. [18]

We start by initializing all the publishers and subscribers for the node. We also set a timer to publish periodically the joint states.

```python
def desired_position_callback(self, msg):
    point_a = msg.point_a
    point_b = msg.point_b
    linear_speed = msg.linear_speed

    # Get the inverse kinematics of the start point and end point
    pose_at_a = kp.Transform(rot=[0, 0, 0, 1], pos=[point_a.x, point_a.y, point_a.z])
    ika_solution = self.chain.inverse_kinematics(pose_at_a)

    pose_at_b = kp.Transform(rot=[0, 0, 0, 1], pos=[point_b.x, point_b.y, point_b.z])
    ikb_solution = self.chain.inverse_kinematics(pose_at_b)

    # Check that both the start and end point are reachable in the task space
    if ika_solution is not None and ikb_solution is not None:
        print("\nCalculated Inverse Kinematics")
    else:
        print("\nInverse kinematics did not converge to a solution.")

    # Find the distance between the two points
    distance = math.sqrt(
        (point_b.x - point_a.x) ** 2 +
        (point_b.y - point_a.y) ** 2 +
        (point_b.z - point_a.z) ** 2
    )

    if linear_speed <= 0:
        self.get_logger().error("Linear speed must be positive.")
        return

    # Compute the time for the path
    duration = distance / linear_speed
    steps = max(1, int(duration * 10))  #  at least one step

    # Calculate each subpoint path
    delta = [
        (point_b.x - point_a.x) / steps,
        (point_b.y - point_a.y) / steps,
        (point_b.z - point_a.z) / steps
    ]
```

Figure 20: Linear Planner part 2

In Fig. [20], we first make sure that the inverse kinematics at point A and point B are valid. We then calculate the linear interpolation between the two points.

The rest of the function injects several points inside of the line between A and B, and then calculates the inverse kinematics at each point. All the joint states are saved in a list. After that, the elements of the list are published one by one into the topic **/joint_states**. The corresponding example video can be found here: `https://drive.google.com/file/d/1VTLVcu16E65oTGjbQB5J_kf3nGRJxOAq/view?usp=sharing`

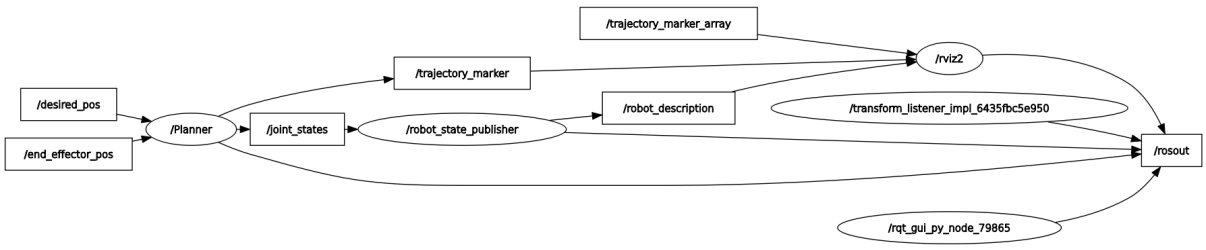

Figure 21: RQT Graph of Method 1

0.4.3 Method 2

For the second method, the same rqt graph explained in method 1 is also applicable. In this method, instead of performing a linear interpolation, we used the library **ruckig**. The corresponding code can be found in the file **trajectory_with_library.py** inside of

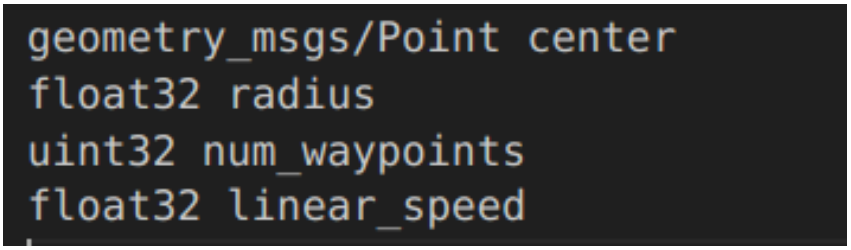

Figure 22: ParametricCommand custom interface

the **trajectory_planner** package. We input to it the joint limits, and the initial and final state of the end effector. We got the joint limits from the official CR10 documentation [6].

The last part of the project required us to draw a parametric function, like a circle. We first implemented a custom interface named **ParametricCommand**, seen in Fig. [22].

It is then processed to generate waypoints, and the path is applied using the library as explained in the previous part. The corresponding video can be found on: `https://drive.google.com/file/d/1-hG9cxXuUUJYvEQjM718cm2hcpw0FxW3/view?usp=sharing`

0.5 Challenges and Solutions

The main challenges we faced was initially with the integration of the rviz and URDF, so the solution was found on the repo of Dobot.

Another main challenge we faced was with integrating the ruckig. Therefore, this gave us the idea of first implementing method 1, which was linear interpolation, and then method 2, which was with ruckig.

Finally we integrated a launch file that allows us to launch everything all together. It can be found inside of **trajectory_planner** package.

It basically just runs all of the previously discussed nodes at once as seen in Fig. [23].

```python
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node
from launch.substitutions import LaunchConfiguration, TextSubstitution


def generate_launch_description():
    urdf_path = os.path.join(
      get_package_share_directory('trajectory_planner'),
      'urdf',
      'CR10URDF.urdf'
      )

    cr10_description = IncludeLaunchDescription(
      PythonLaunchDescriptionSource([os.path.join(
         get_package_share_directory('cr10_description'), 'launch'),
         '/display2.launch.py'])
      )

    forward_kinematics = Node(
         package='forward_kinematics',
         executable='forward_kinematics',
         name='ForwardKinematics',
         parameters=[{
            'urdf_path': urdf_path,
         }])

    trajectory_planner = Node(
         package='trajectory_planner',
         executable='trajectory_with_library',
         name='Planner',
         parameters=[{
            'urdf_path': urdf_path,
         }])


    return LaunchDescription([
      cr10_description,
      forward_kinematics,
      trajectory_planner,
   ])
```

Figure 23: Full system launch file

Bibliography

[1] https://www.dobot-robots.com/products/cr-series/cr10.html

[2] https://github.com/neka-nat/kinpy

[3] https://pypi.org/project/ruckig

[4] https://docs.ros.org/en/humble/index.html

[5] https://github.com/DobotArm/DOBOT_6Axis_ROS2_V4/blob/main/dobot_rviz/launch/dobot_rviz.launch.py

[6] https://unchainedrobotics.de/en/products/robot/cobot/dobot-cr10