Robotic Manipulation Of Deformable Objects with Franka Robot

Berk TEPEBAĞ

Institut Pascal
UNIVERSITE CLERMONT AUVERGNE (UCA)
berk.tepebag@gmail.com

Supervisor: Dr. Erol ÖZGÜR

September 23, 2019

Contents

Abstract

Although industrial robots have a long history with rigid object manipulation, deformable object manipulation is still considered 'new'. Every object has its deformability characteristic which makes each situation unique. This paper presents a novel controller that combines a joint position controller with joint velocity controller. Later, the novel controller has been integrated with the 'MoveIt!' library to ease the process of developing a method to find the material characteristics of the deformable object using machine learning techniques. For this project, Franka-Emika Panda robotic arm has been used as the manipulator. Paper briefly shows the hardware and software setups of the system, how to write joint position and velocity controllers and finally how to combine novel controller with MoveIt! library and benefit from its GUI and inverse kinematics solver.

1 Introduction

Rigid object manipulation by robotic arms exists for more than 50 years while deformable object manipulation is still considered "new". Rigid object manipulation focuses on the heavy industry (car production lines, phone assembly lines, etc.) while deformable object manipulation focuses on softer materials (e.g. human body) which "deforms" under applied force. Due to the nature of deformable materials, they cannot be treated as easily as the rigid objects. To be able to interact with deformable materials we have to know deformation characteristics. This is not an easy task because every material has unique deformation characteristics. This report presents the work done with Franka-Emika Panda robotic arm (from now on will be called "Franka robot") to find properties of deformable material using machine learning algorithms. To achieve this; First Franka robot's hardware and software setups had been explained. Later, Joint Position Controller's theory had been discussed and a simple Joint Position Controller had been written to show how the controller works. Afterward, a teleoperation function which relies on "ROS topics" had been developed to send commands to the controllers. Then, a Joint Velocity Controller had been written. Finally, "Joint Position Controller" had been combined with MoveIt! library to control the Franka robot from the RVIZ simulator.

2 Hardware Setup for Vision-based Control using Franka Robot - Introduction

This report informs about the hardware used in the project and the setup procedures of the hardware.

The rest of the report is organized as follows: Section 3 lists the hardware, section 4 presents Franka robot and section 5 explains the setup procedures.

3 Hardware list

Franka robot: Franka Emika GmbH is a German robotics company. In this project Panda model robotic arm will be used. It will be called as "Franka robot" from now on.

Workstation PC: Dell OptiPlex 5060, 6 Core Intel i7 CPU, 16 GB Ram, 2 TB Harddrive, Gigabit Ethernet adapter.

Control: Control is the black box which turns commands from user to electric currents and controls Franka robot.

Camera: Intel® RealSense^{TM} Camera. The camera will be used for vision-based control of Franka robot.

4 Franka robot

4.1 Technical details

Franka robot is a 7 DOF robotic arm equipped with torque sensors in all 7 axes. It weighs 18 kg. It can lift up to 3 kg. Franka robot has a gripper which weighs 0.7 kg. The gripper can apply continuous grasping force of 70N and a maximum of 140N. Franka robot is controlled with Control which is connected to Workstation PC via Ethernet cable. Control communicates at 1 kHz frequency. Energy consumption is 300 watt on average and 600 watt at peak which makes it possible to be used with standard 2 phase electric power.

4.2 Geometry

The geometry of Franka robot is explained in this section. Franka-Emika company preferred Denavit–Hartenberg parameters to explain the geometry of Franka robot. Thus same method has been followed to explain Franka robot's transformation parameters. The Denavit–Hartenberg parameters for Franka robot's kinematic chain can be seen in figure 2. Figure 1 shows original positions of the motors and their coordinate frames. To simplify the Denavit-Hartenberg parametrization, some coordinate frame placements have been changed by Franka-Emika GmbH.

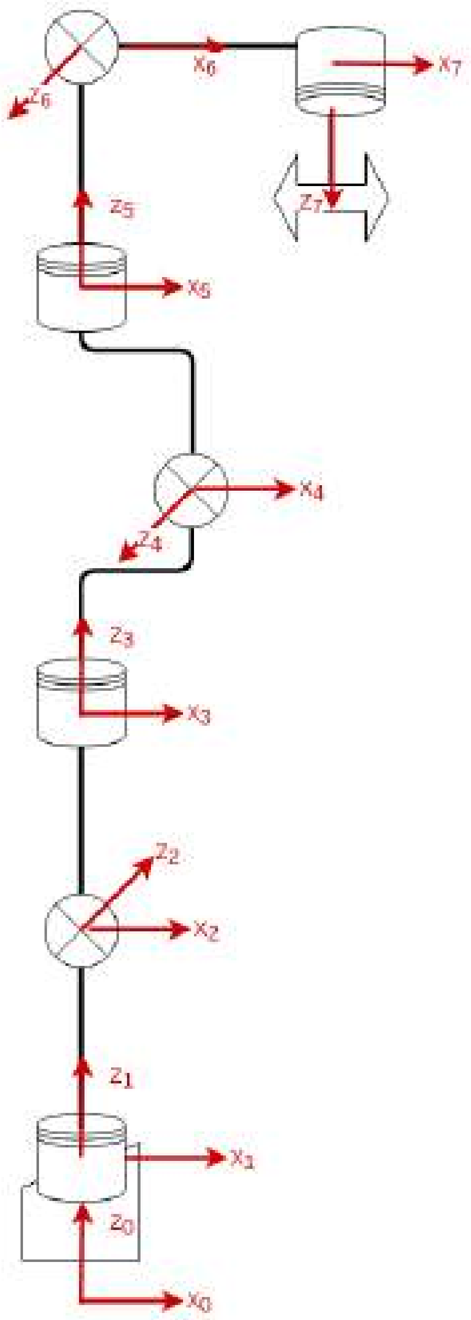


Figure 1: Geometry and motor placements.

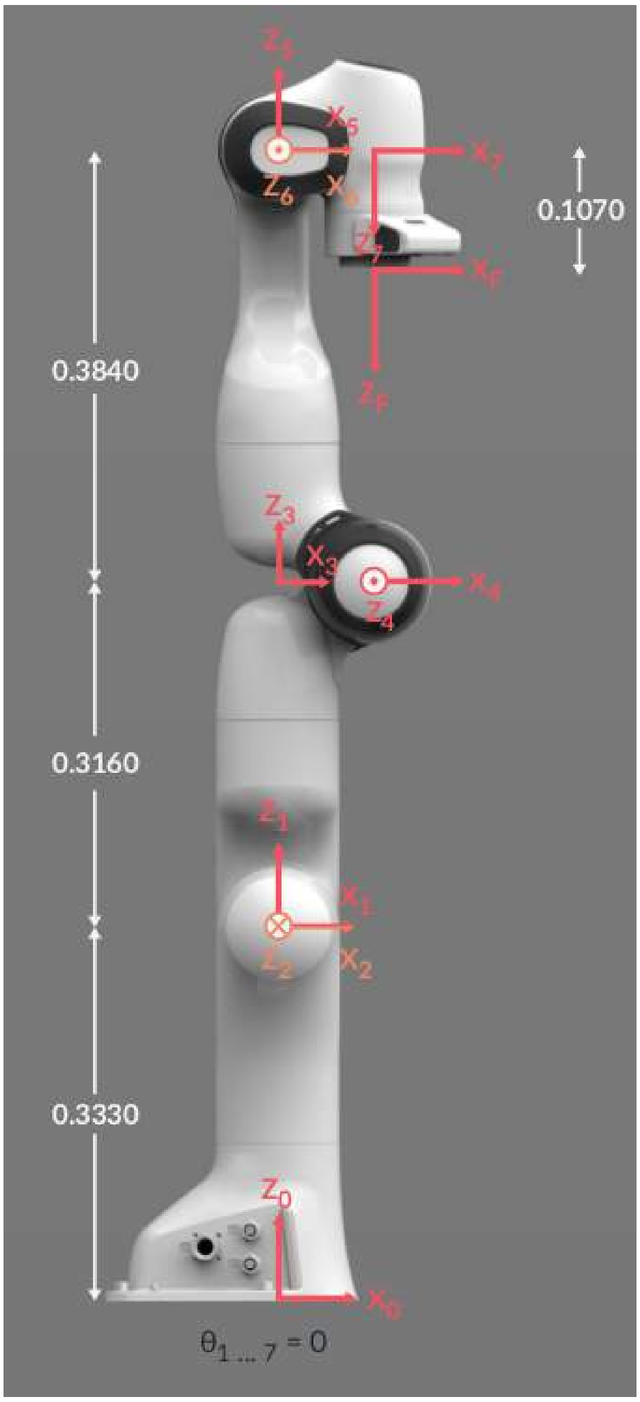


Figure 2: Denavit-Hartenberg parameters.

Denavit-Hartenberg parameters: The following four parameters are known as Denavit–Hartenberg parameters [1]. Each joint's a, d, α, θ values can be found in Table 1.

- common normal: The shortest line which is perpendicular to the two consecutive joints' z-axes. Next joint's origin is at this point on it's z-axis.
- a: The length of the common normal or the radius which next joint is rotating about the previous joint's z-axis.
- d: Depth along the previous joint's origin to the common normal.
- α: The angle between the two consecutive joints' z-axes about the next joint's x-axis.
- θ: The angle between the two consecutive joints' x-axes about previous joint's z-axis.

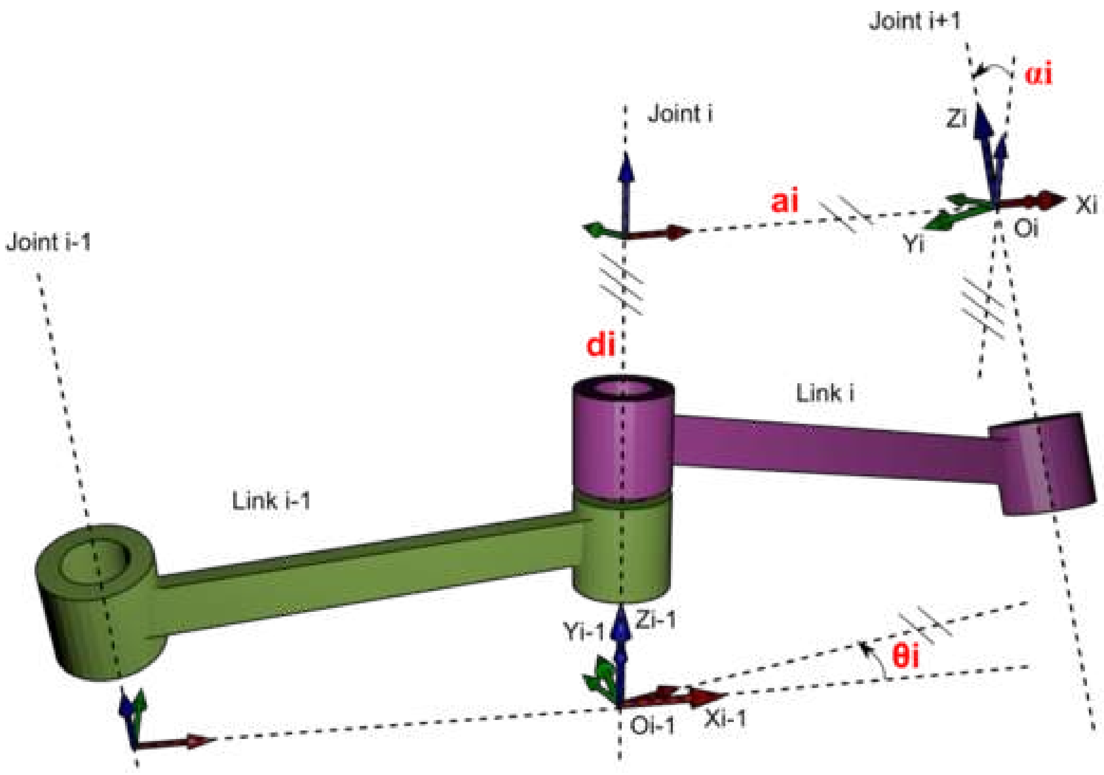


Figure 3: Denavit-Hartenberg parameters [2].

Joint	a(m)	d(m)	α(rad)	θ(rad)
Joint 1	0	0.333	0	θ_1
Joint 2	0	0	$-\pi/2$	θ_2
Joint 3	0	0.316	$\pi/2$	θ_3
Joint 4	0.0825	0	$\pi/2$	θ_4
Joint 5	-0.0825	0.384	$-\pi/2$	θ_2
Joint 6	0	0	$\pi/2$	θ_7
Joint 7	0.088	0	$\pi/2$	θ_7
Flange	0	0.107	0	0

Table 1: Denavit-Hartenberg parameters of Franka robot [4].

4.3 Workspace

Franka robot's workspace determined by the combination of the joints' space limits. It can reach up to 855 mm in horizontal radius. In vertical direction it can reach to the points, 1190 mm and -360 mm from mounting point, depicted in figures 4 and 5.

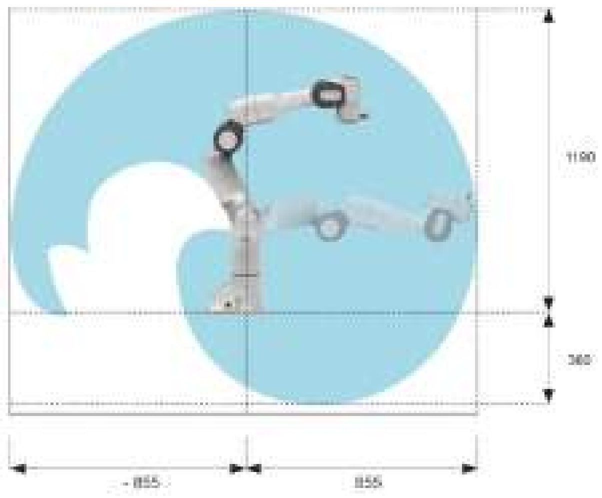


Figure 4: Cartesian Workspace from side view.

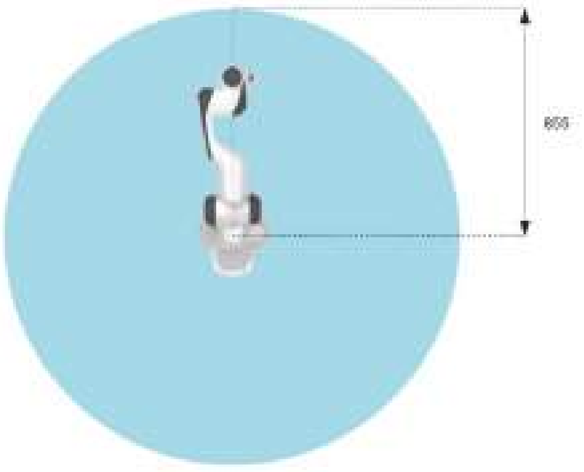


Figure 5: Cartesian Workspace from top view.

4.4 Motion limits of Franka robot

Franka robot has physical and technical limitations. Those limitations are explained here.

4.4.1 Cartesian space limits

Maximum velocity, acceleration and jerk limits of Franka robot for translational, rotational and elbow movement are given in Table 2.

Limit	Translation	Rotation	Elbow
Max velocity	1.7000 m/s	2.5000 rad/s	2.1750 rad/s
Max acceleration	13.0000 m/s^2	25.0000 rad/s^2	10.0000 rad/s^2
Max jerk	6500.0000 m/s^3	12500.0000 m/s^3	5000.0000 rad/s^3

Table 2: Velocity, acceleration and jerk limits of translation, rotation and elbow movements [4].

4.4.2 Joint space limits

Each joint of Franka robot has minimum-maximum position, maximum velocity, maximum acceleration, maximum jerk, maximum torque and maximum rotatum limits. To prevent any damage to Franka robot, any command exceeding these limits will return an error and robot will stop. This will be explained briefly later at software report. Limits can be seen at Table 3.
Note: To define rotation direction, right hand rule is used which makes rotation in counter clockwise direction positive (+) and clockwise negative (-).

- q_{max}: Angular position limit of a joint around z-axis, in counter clock wise direction (rad).
- q_{min}: Angular position limit of a joint around z-axis, in clock wise direction (rad).
- $\dot{q}_{max}$: Angular speed limit of a joint (rad/s).
- $\ddot{q}_{max}$: Angular acceleration limit of a joint (rad/s^2).
- $\dddot{q}_{max}$: Angular jerk limit of a joint (rad/s^3).
- τ_{jmax}: Torque limit of a joint ($N \cdot m$).
- $\dot{\tau}_{jmax}$: Rotatum (change of torque per second ($N \cdot m/s$) limit of a joint.

Name	**Joint 1**	**Joint 2**	**Joint 3**	**Joint 4**	**Joint 5**	**Joint 6**	**Joint 7**	**Unit**
q_{max}	2.8973	1.7628	2.8973	-0.0698	2.8973	3.7525	2.8973	rad
q_{min}	-2.8973	-1.7628	-2.8973	-3.0178	-2.8973	-0.0175	-2.8973	rad
$\dot{q}_{max}$	2.1750	2.1750	2.1750	2.1750	2.6100	2.6100	2.6100	rad/s
$\ddot{q}_{max}$	15	7.5	10	12.5	15	20	20	rad/s^2
$\dddot{q}_{max}$	7500	3750	5000	6250	7500	10000	10000	rad/s^3
τ_{jmax}	87	87	87	87	12	12	12	$N \cdot m$
$\dot{\tau}_{jmax}$	1000	1000	1000	1000	1000	1000	1000	$N \cdot m/s$

Table 3: Joints' position, velocity, acceleration, jerk, torque, rotatum limits of Franka robot [4].

5 Setup

For the project a setup consisting of Franka robot, Control and Workstation PC was established in Sigma Clermont CTT laboratory. The setup can be seen in figure 6. Hardware list has previously been discussed in the section 3. In this section the details of the other components of the project is discussed and system setup explained.

5.1 Workstation PC

Any modern desktop or laptop computer can be chosen as Workstation PC. Communication speed is crucial for controlling the robot [5]. Our Workstation PC has Gigabit Ethernet in order to obtain fastest connection possible. Minimum system requirements for Workstation PC are given in Table 4.

	Minimum System Requirements
Operating system	Linux with PREEMPT RT patched kernel
Network card	100BASE-TX

Table 4: Minimum system requirements for Workstation PC.

As mentioned on Franka Emika website [6]: "Since the robot sends data at 1 kHz frequency, it is important that the workstation PC is configured to minimize latencies." No problems related to this situation occurred, therefore no extra steps has been taken. Preferably, Workstation PC and Control should be connected directly without any router or repeater that may increase latency and cause errors. A Workstation PC with a second Ethernet adapter or Wi-Fi adapter is advised to be able to reach the Internet while controlling Franka robot.

5.2 Network connection

Workstation PC and Control connected directly via CAD 5 Ethernet cable. Since connection quality is dependent on connection between Workstation PC and Control [6] it is essential to have shortest and highest quality Ethernet cable. As mentioned before Gigabit Ethernet interface is being used to achieve highest possible connection speeds.

5.3 Intel REALSENSE Camera

The Intel® RealSense™ depth camera D435 is a stereo camera, is a good choice for robotic applications. It has a range up to 10 meter and with it's small form factor it can be placed on the hand without causing disturbance in movement. It is connected with USB 3.0 to the Workstation PC [10]. Camera can be seen in figure 6 at the tip of the gripper and in figure 7.

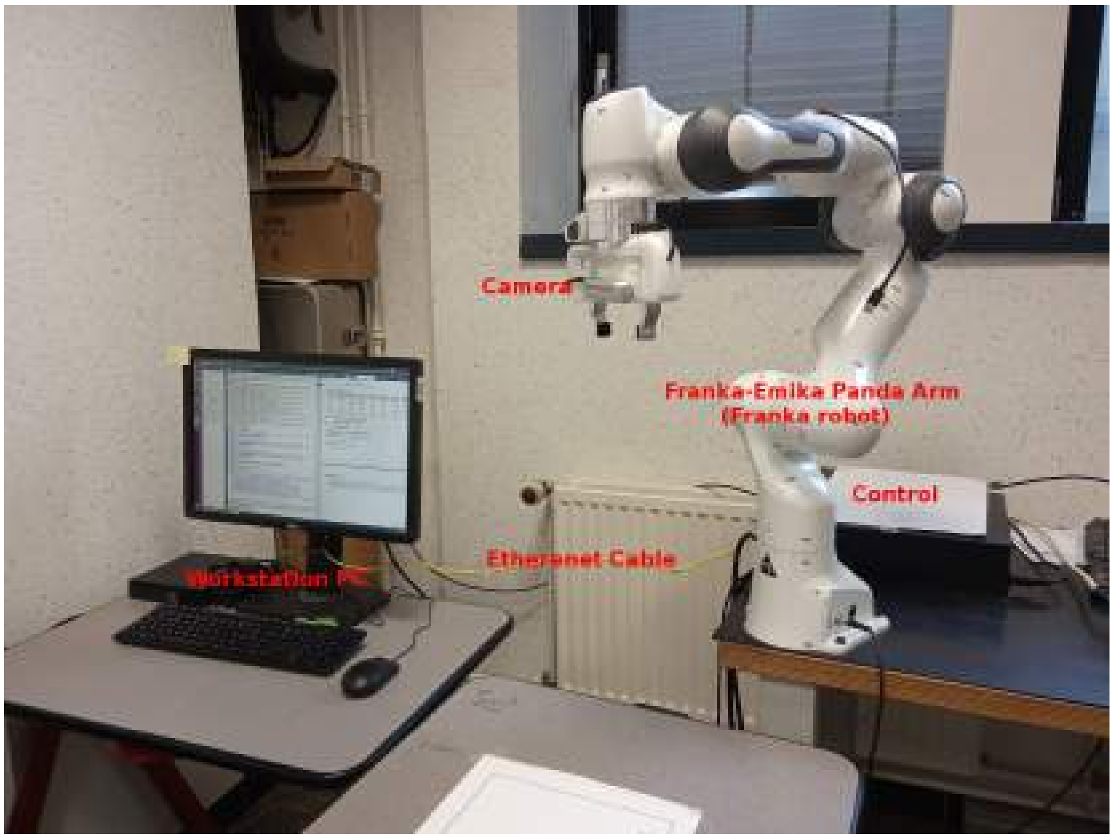


Figure 6: Project setup: Left, Workstation PC. Between Control and Workstation PC, yellow Ethernet cable. Right, behind Franka robot, the Control. Franka robot with camera attached to the gripper is located in front of the Control.

Figure 7: Intel REALSENSE 3D Camera.

References

[1] Spong, Mark W.; Vidyasagar, M. (1989). Robot Dynamics and Control. New York: John Wiley & Sons. ISBN 9780471503521.

[2] Wikipedia, Denavit-Hartenberg parameters. (2019, April 5) Retrieved from `https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg_parameters#/media/File:Classic-DHparameters.png`

[3] Denavit-Hartenberg Parameters Note. Reprinted from Franka Control Interface (FCI) (2019, April 5) Retrieved from `https://frankaemika.github.io/docs/control_parameters.html`

[4] Franka Control Interface (FCI) Overview. (2019, April 5) Retrieved from `https://frankaemika.github.io/docs/overview.html`

[5] Franka Control Interface (FCI) Workstation PC. (2019, April 5) Retrieved from `https://frankaemika.github.io/docs/requirements.html#workstation-pc`

[6] Franka Control Interface (FCI) Network. (2019, April 5) Retrieved from
`https://frankaemika.github.io/docs/requirements.html#network`

[7] Franka Control Interface (FCI) Getting Started. (2019, April 5) Retrieved from
`https://frankaemika.github.io/docs/getting_started.html`

[8] Franka Emika, Panda Web Site. (2019, April 5) Retrieved from
`https://www.franka.de/technology/`

[9] Franka Emika, Panda Panda TECHNICAL DATA. (2019, April 5) Retrieved from
`https://s3-eu-central-1.amazonaws.com/franka-de-uploads-staging/uploads/2018/05/2018-05-datasheet-panda.pdf`

[10] Intel® RealSense^{TM} Depth Camera D435 Website. (2019, April 5) Retrieved from
`https://www.intelrealsense.com/depth-camera-d435/`

6 Software Setup for Vision-based Control using Franka Robot - Introduction

Franka robot software package consists of applications and libraries in order to control Franka robot. A wide variety of software libraries are made available by Franka-Emika GmbH. This report informs about usage of those libraries. For further information Franka Emika's web-page, forums and Github page can be visited.

The rest of the report is organized as follows: Section 7 presents the softwares used in the development, section 8 gives information about libfranka C++ library and section 9 gives information and shows the setup of the ROS software.

7 Software list

In this section general information will be provided about the softwares used in the project. A brief layout of softwares is depicted at figure 8.

OS Ubuntu 16.04 Xenial Xerxus - PREEMPT RT Kernel: 4.14.12-rt10. ROS versions are specific to Linux versions and Franka robot is officially supporting ROS - Kinetic Kame [1], that is why Ubuntu 16.04 is chosen as OS in the Workstation PC. In order to control Franka robot with libfranka, the controller program on the workstation PC must run with real-time priority under a PREEMPT_RT kernel [12].

Robotic Operating System (ROS) Version: Kinetic Kame. ROS is a flexible framework for writing robot software. Robotic programmers can use built-in libraries such as ROS control, visualization, deep learning etc. to reduce time and effort spend to start using robot and developing projects [2].

Franka Control Interface (FCI) provides the current status of the robot and enables its direct control with the Workstation PC connected via Ethernet.

libfranka is the C++ implementation of the client (user) side of the FCI. It handles the network communication with Control and provides interfaces between client and Franka robot.

franka_ros is a meta-package that integrates libfranka library into ROS and ROS control.

Desk is a web page like, easy to use interface that used to create from simple to very complicated robotic tasks. Since it does not need any coding knowledge it is good for creating static jobs. It is not feasible for creating non-static jobs.

8 libfranka

As shown in the figure 12, libfranka takes a role as a medium layer between FCI and franka_ros. It is also possible to use directly libfranka to command Franka robot without franka_ros if ROS functionalities are not necessary. In this project, franka_ros is used due to usage of the computer vision and deep learning applications which are mostly dependent on Python language. libfranka is responsible from taking user commands and turning into the code that FCI can understand, so it has to be installed in order to use franka_ros. In this section, installation process will be explained briefly.

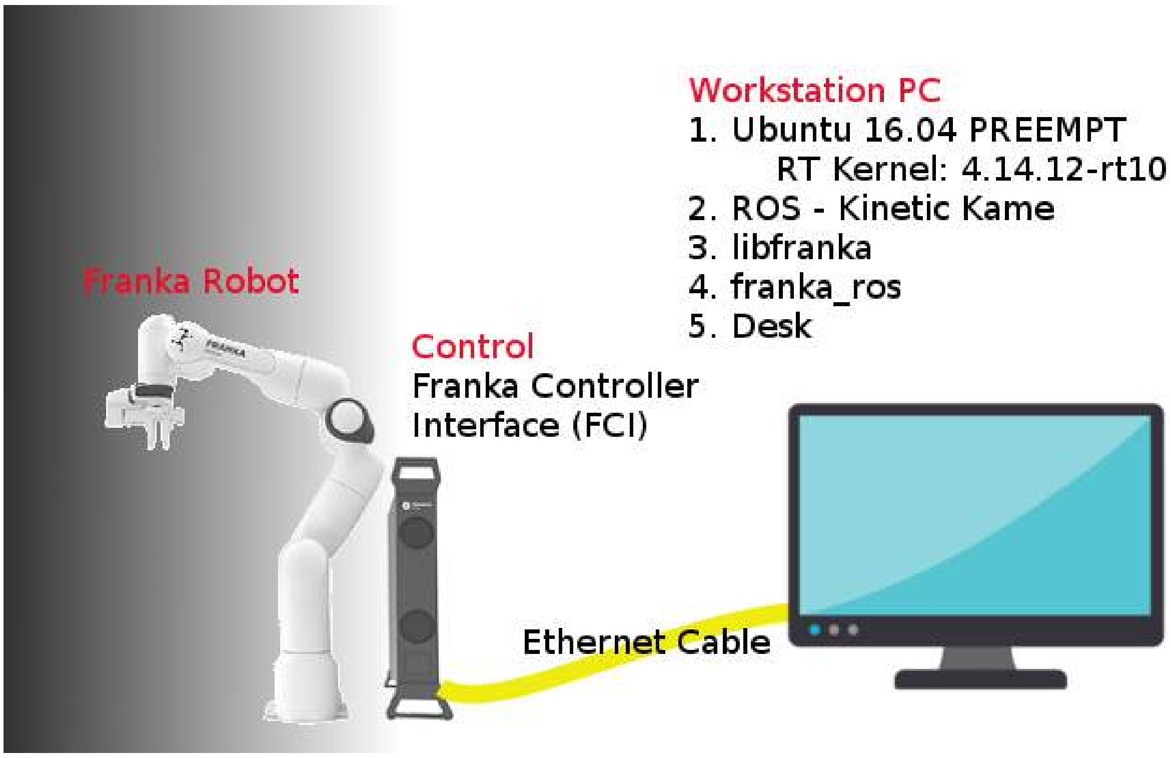


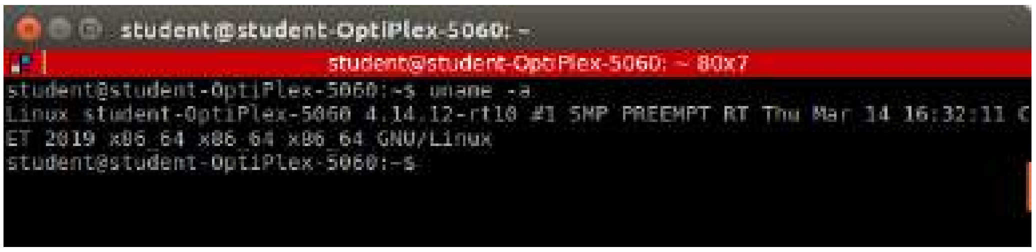
Figure 8: Software and hardware relations.

8.1 Installation

To be able to use libfranka it must be built by following the instructions at the Franka-Emika, Franka Control Interface (FCI) website [3]. In this project, version 0.6.0 is used.

Setting up the real-time kernel To be able to use libfranka in Workstation PC, controller program must run with real-time priority under a PREEMPT_RT kernel. Follow the steps from the website [12] to patch the kernel and create an installation package.

After following the instructions and checking it at command line with `uname -a` command, output should contain `PREEMPT RT` and the version number you chose. Additionally, `/sys/kernel/realtime` should exist and contain the the number `1` (figure 9).



```
student@student-OptiPlex-5060: ~
student@student-OptiPlex-5060: ~ 80x7
student@student-OptiPlex-5060:~$ uname -a
Linux student-OptiPlex-5060 4.14.12-rt10 #1 SMP PREEMPT RT Thu Mar 14 16:32:11 C
ET 2019 x86_64 x86_64 x86_64 GNU/Linux
student@student-OptiPlex-5060:~$
```

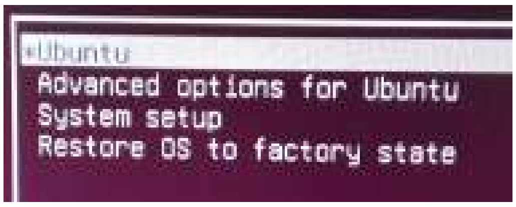
Figure 9: PREEMPT RT - Kernel Version Output.

Then restart the computer. Now GRUB screen should let you choose the Kernel. If you do not see GRUB screen press ’ESC’ button while Ubuntu is loading, select "Advanced options for Ubuntu" (figure 10) and then from new screen select "Linux 4.14.12-rt10" (figure 11).

8.2 Overview of libfranka

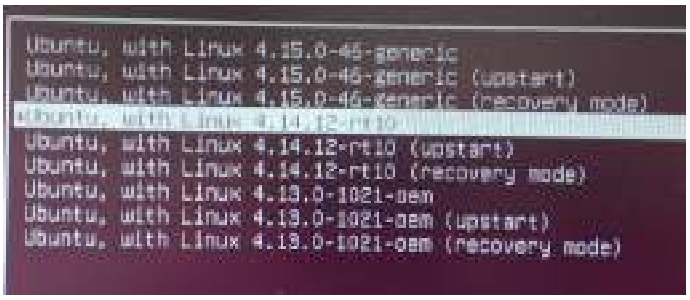
libfranka [4] takes inputs from client side of the FCI and sends them to the Control via network connection (figure 12). Interfaces are provided by libfranka in order to:

1. Execute non-realtime commands.
2. Execute realtime commands at 1 kHz.



```
*Ubuntu
 Advanced options for Ubuntu
 System setup
 Restore OS to factory state
```

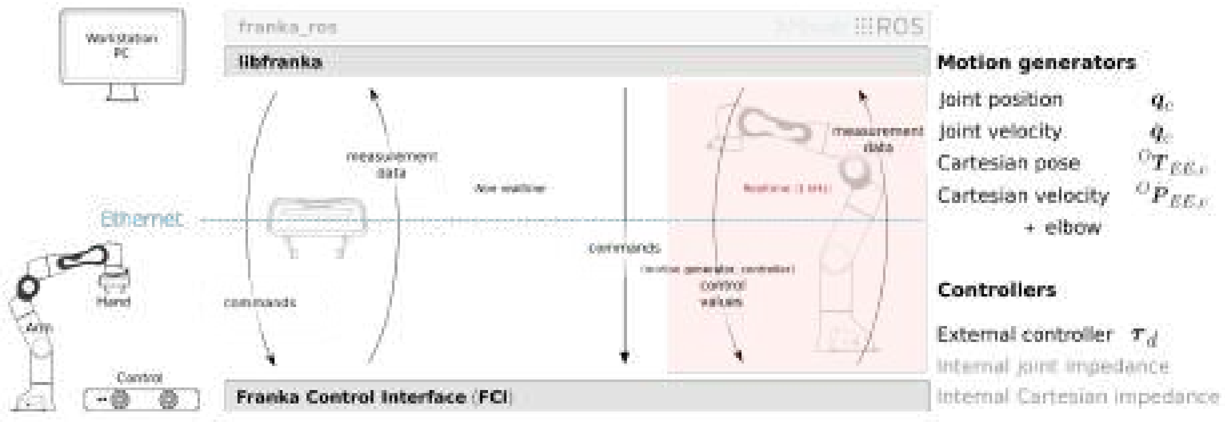
Figure 10: Ubuntu GRUB Loading screen.



```
 Ubuntu, with Linux 4.15.0-46-generic
 Ubuntu, with Linux 4.15.0-46-generic (upstart)
 Ubuntu, with Linux 4.15.0-46-generic (recovery mode)
*Ubuntu, with Linux 4.14.12-rt10
 Ubuntu, with Linux 4.14.12-rt10 (upstart)
 Ubuntu, with Linux 4.14.12-rt10 (recovery mode)
 Ubuntu, with Linux 4.13.0-1021-oem
 Ubuntu, with Linux 4.13.0-1021-oem (upstart)
 Ubuntu, with Linux 4.13.0-1021-oem (recovery mode)
```

Figure 11: Inside the "Advanced options for Ubuntu" menu.

3. Getting sensor data at 1 kHz by reading robot state.
4. Computes desired kinematic and dynamic parameters by accessing to the model library.



Workstation PC
franka_ros
ROS
libfranka
measurement data
Non realtime
Ethernet
commands
Hand
Arm
Control
Realtime (1 kHz)
measurement data
commands (motion generator, controller)
control values
Franka Control Interface (FCI)
Motion generators
Joint position q_c
Joint velocity $\dot{q}_c$
Cartesian pose ${}^O T_{EE,c}$
Cartesian velocity ${}^O \dot{P}_{EE,c}$
+ elbow
Controllers
External controller τ_d
Internal joint impedance
Internal Cartesian impedance

Figure 12: On the left: Schematic overview of the communication between the software and hardware. On the right: Real-time interfaces; motion generators and controllers.

8.3 Non-realtime commands

Non-realtime commands are always executed outside of any realtime control loop. They are blocking and TCP/IP-based. Gripper commands and some configuration-related commands for the Franka-robot are included here.

8.4 Realtime commands

Realtime commands requires 1 kHz connection [2]. There are two types of realtime interfaces:

1. **Motion generators**, which define a robot motion in joint or Cartesian space.
2. **Controllers**, which define the torques to be sent to the robot joints.

There are 4 types of external motion generators and 3 types of controllers (one external and 2 internal) that depicted in figure 12.

Either a single interface or combination of two different interfaces can be used. Specifically :

1. Joint Position or Joint Velocity or Cartesian pose or Cartesian velocity + elbow and Internal joint impedance or Internal Cartesian impedance
2. Only an external controller, i.e. torque control
3. Joint Position or Joint Velocity or Cartesian pose or Cartesian velocity + elbow and an external controller to use inverse kinematics of control in your external controller.

8.5 Signal processing

Connection between Workstation PC and Control is not always perfect. Due to connection problems data packages can get lost, or connection speed does not satisfy the minimum requirements [20]. These can cause problems such as, acceleration, jerk limit violations. libfranka has built in signal processing functions make sure that user commands are in the limits of the interface. In realtime control loops 2 options are available:

- A first-order **low-pass filter** to smooth the user-commanded signal.
- A **rate limiter**, that saturates the time derivatives of the user-commanded values.

Rate limiters: It is known as "safe controllers" and are running by default with version 0.4. All realtime interfaces have rate limiters. Rate limiters will limit the rate of the change of the signals sent by the user in order to prevent the violation of the limits of the interface. Rate limiter, limit's the motion generators' (Joint position, Joint velocity controllers), acceleration and jerk rates. Also it limit's external controller's (Torque controller) torque rate. Main purpose of the limiters is to increase the robustness of the users control loop. If packet losses occur, even if commands stay in the limits of the interface, control may detect a violation of velocity, acceleration or jerk limits. Rate limiting prevents this situation by adapting the user commands.

Low-pass filter: It filters the commands under 100 Hz to smoothen the commanded signals which provides more stable robot motions. Filtering does not cause a violation to the limits of the interface. It is introduced with the version 0.5 (figure 13)

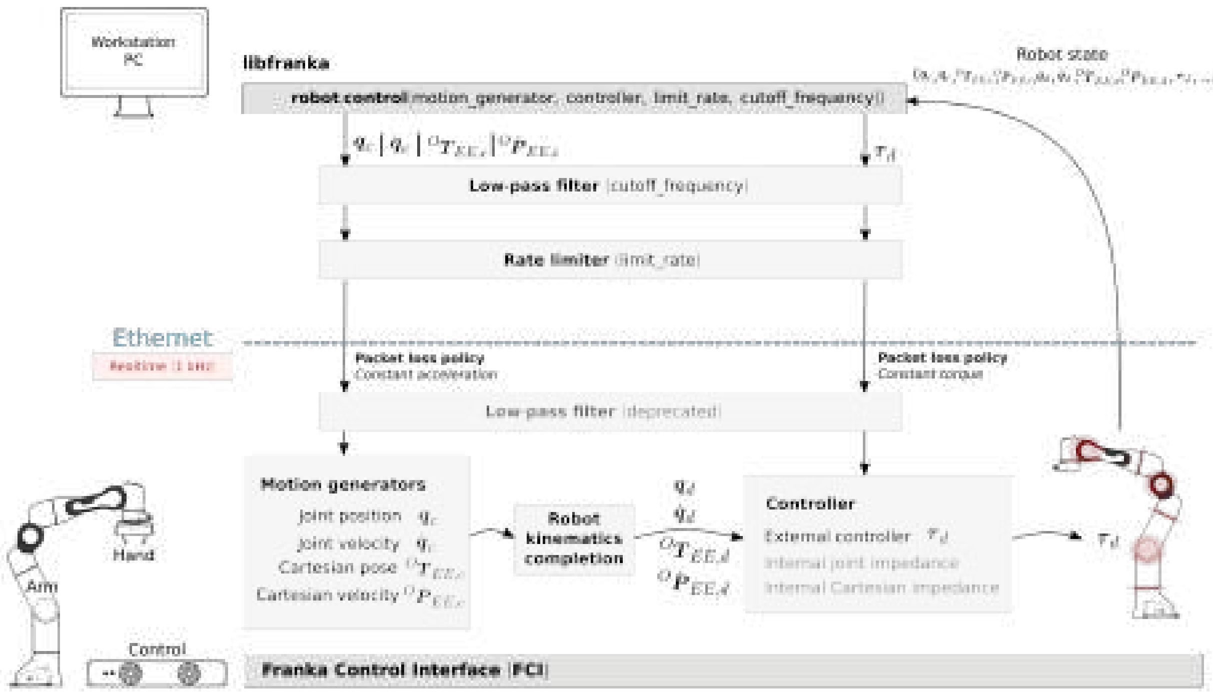

Workstation PC
libfranka
Robot state
robot.control(motion_generator, controller, limit_rate, cutoff_frequency)
$q_c \mid \dot{q}_c \mid {}^O T_{EE,c} \mid {}^O \dot{P}_{EE,c}$
τ_d
Low-pass filter (cutoff_frequency)
Rate limiter (limit_rate)
Ethernet
Realtime (1 kHz)
Packet loss policy
Constant acceleration
Packet loss policy
Constant torque
Low-pass filter (deprecated)
Motion generators
Joint position q_c
Joint velocity $\dot{q}_c$
Cartesian pose ${}^O T_{EE,c}$
Cartesian velocity ${}^O \dot{P}_{EE,c}$
Robot kinematics completion
q_d
$\dot{q}_d$
${}^O T_{EE,d}$
${}^O \dot{P}_{EE,d}$
Controller
External controller τ_d
Internal joint impedance
Internal Cartesian impedance
τ_d
Hand
Arm
Control
Franka Control Interface (FCI)

Figure 13: Franka robot control scheme in realtime loop.

To control the signal processing functions, all robot.control() function calls have two additional optional parameters.

1. Activates/deactivates the rate limiter.
2. Specify cut off frequency. If cut off frequency is greater or equal to 1000, deactivates the first-order low-pass filter.

Attention! If more than 20 packets are lost in a row the control loop is stopped with the "communication constraints violation" exception.

It is advised by Franka-Emika GmbH to use rate limiter and low-pass filter after a smooth motion generator or controller has been designed. For the first tests of the control loops, advised to deactivate the rate limiter and low-pass filter.

8.6 Robot state

The robot state reads and estimates sensor readings at 1 kHz [7], providing:

1. Joint level signals: motor and estimated joint angles and their derivatives, joint torque and derivatives, estimated external torque, joint collision/contacts.
2. Cartesian level signals: Cartesian pose, configured end effector and load parameters, external wrench acting on the end effector, Cartesian collision.
3. Interface signals: the last commanded and desired values and their derivatives, as explained in the previous subsection.

Complete list of robot states can be reached from the related web page [8].

8.7 Model library

The robot model library provides [9]:

1. The forward kinematics of Franka robot.
2. The body Jacobian matrices and zero Jacobian matrices for all joints of Franka robot.
3. Dynamic parameters: Inertia matrix, Coriolis and centrifugal vector & gravity vector.

8.8 Errors

During use of the FCI user may encounter errors due to: non compliant commands sent by the user, communication problems, the robot behavior (i.e collision detection). The complete list of errors can be reached from Franka robot's web-page [10].

9 franka_ros

The franka_ros meta-package makes possible to use libfranka with ROS and ROS control. An overview of the franka_ros architecture depicted in figure 14. Next, installation process and usage of franka_ros will be explained.

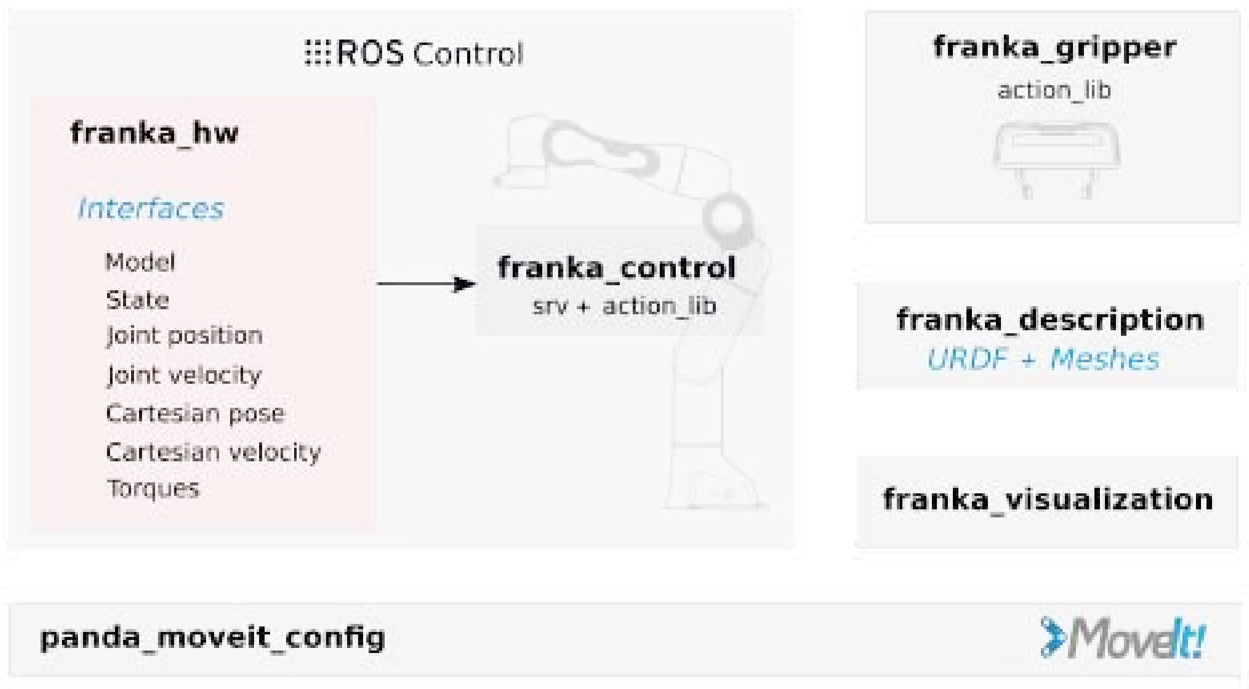


Figure 14: ROS architecture scheme.

9.1 Installation

To be able to control the robot using ROS, following instructions must be followed [11].

Before starting installation, OS, ROS and libfranka must be installed. Although libfranka and franka_ros can work on different Linux distributions and ROS versions, only listed ones are officially supported by Franka-Emika.

franka_ros can be installed using:
`sudo apt install ros-kinetic-libfranka ros-kinetic-franka-ros`

9.1.1 Building the ROS packages

If ROS is installed and working properly, it is time to create a Catkin Workspace. Follow the steps from building the ROS packages website [21].

9.2 franka_ros packages

Franka Emika created franka_ros packages in order to provide advantages of the ROS with built in libfranka integration.

To prevent sourcing with every new terminal screen add 2 lines to ~/.bashrc with your favorite text editor (In this project Sublime Text is being used as text editor):

- `source /opt/ros/kinetic/setup.bash`
- `source ~/catkin_ws/devel/setup.bash`

With every new terminal screen it will automatically source so user does not have to do it.
Also add 3 shortcuts which will be used regularly:

- `alias cm='cd  /catkin_ws/ && catkin_make'`
- `alias cs='cd  /catkin_ws/src'`
- `alias cw='cd  /catkin_ws && source /opt/ros/kinetic/setup.bash'`

9.2.1 franka_description

Description package can be reached from:
`/catkin_ws/src/franka_ros/franka_description/robots`. This package involves URDF files to describe Franka robot's kinematics, joint limits, visual surfaces and collision space. Visual description is simplified to obtain the collision space, that is used to improve performance of collision checks.

9.2.2 franka_gripper

To be able to use gripper from ROS, franka_gripper_node has been defined. The node publishes the state of the gripper and offers the actions servers [13] such as moving, grasping etc.

9.2.3 franka_hw

franka_hw package contains hardware abstraction of the robot for the ROS control framework based on the libfranka API. Interfaces available to the controllers provided by the hardware class franka_hw::FrankaHW are shown in Table 5. In this project, `VelocityJointInterface` and `PositionJointInterface` will be used to control Franka robot.

To use controller interfaces, user should retrieve resource handles by name, shown in Table 6.

9.2.4 franka_control

franka_control provides a variety of ROS services to expose the full libfranka API in the ROS ecosystem. Check the website [15] for the details. One of the most useful service is recovering from errors or reflexes with `franka_control::ErrorRe`
It can be used by simply publishing on the action goal topic:

```
   rostopic pub -1 /franka_control/error_recovery/goal
franka_control/ErrorRecoveryActionGoal "{}"
```

By this way, user does not have to exit and restart controller from terminal manually.

Interface	Function
hardware_interface::JointStateInterface	Reads joint states.
hardware_interface::VelocityJointInterface	Commands joint velocities and reads joint states.
hardware_interface::PositionJointInterface	Commands joint positions and reads joint states.
hardware_interface::EffortJointInterface	Commands joint-level torques and reads joint states.
franka_hw::FrankaStateInterface	Reads the full robot state.
franka_hw::FrankaPoseCartesianInterface	Commands Cartesian poses and reads the full robot state.
franka_hw::FrankaVelocityCartesianInterface	Commands Cartesian velocities and reads the full robot state.
franka_hw::FrankaModelInterface	Reads the dynamic and kinematic model of the robot.

Table 5: List of the available interface to the controllers [14].

Interface	Resource handle name
hardware_interface::JointStateInterface	"<arm_id>_joint1" to "<arm_id>_joint7"
hardware_interface::VelocityJointInterface	"<arm_id>_joint1" to "<arm_id>_joint7"
hardware_interface::PositionJointInterface	"<arm_id>_joint1" to "<arm_id>_joint7"
hardware_interface::EffortJointInterface	"<arm_id>_joint1" to "<arm_id>_joint7"
franka_hw::FrankaStateInterface	"<arm_id>_robot"
franka_hw::FrankaPoseCartesianInterface	"<arm_id>_robot"
franka_hw::FrankaVelocityCartesianInterface	"<arm_id>_robot"
franka_hw::FrankaModelInterface	"<arm_id>_robot"

Table 6: List of the resource handle names of the controller interfaces [14].

9.2.5 franka_visualization

Visualization package [16] publishes Franka robot's and grippers joint states in order to visualize in Rviz. Rviz shows realtime position of Franka robot and gripper.

9.2.6 franka_example_controllers

This package [17] contains example controllers, in this project

1. joint_position_example_controller.cpp
2. joint_velocity_example_controller.cpp

will be used as a guide to write our own controllers.

9.2.7 panda_moveit_config

Franka-Emika provides built in MoveIt! [18] configuration for Franka robot and gripper. Due to some unknown reasons MoveIt! was not working during writing of this report. This part will be updated if any solution is found.

References

[1] Franka, Emika. (2019, April 15). libfranka, Installation. Retrieved from https://frankaemika.github.io/docs/installation.html

[2] Open Source Robotics Foundation (2019, May 3). About ROS. Retrieved from https://www.ros.org/about-ros/

[3] Franka, Emika. (2019, April 15). libfranka, Installation. Retrieved from https://frankaemika.github.io/docs/installation.html#building-libfranka

[4] Franka, Emika. (2019, April 12). libfranka, Introduction. Retrieved from https://frankaemika.github.io/docs/libfranka.html#

[5] Franka, Emika. (2019, April 12). libfranka, Real Time Commands. Retrieved from https://frankaemika.github.io/docs/libfranka.html#realtime-commands

[6] Franka, Emika. (2019, April 12). libfranka, Real Time Commands. Retrieved from https://frankaemika.github.io/docs/libfranka.html#under-the-hood

[7] Franka, Emika. (2019, April 15). libfranka, Sensor Readings, Robot State. Retrieved from https://frankaemika.github.io/docs/libfranka.html#robot-state

[8] Franka, Emika. (2019, April 15). libfranka, Robot state list. Retrieved from https://frankaemika.github.io/libfranka/structfranka_1_1RobotState.html

[9] Franka, Emika. (2019, April 15). libfranka, Model library. Retrieved from https://frankaemika.github.io/docs/libfranka.html#model-library

[10] Franka, Emika. (2019, April 15). libfranka, Errors. Retrieved from https://frankaemika.github.io/docs/libfranka.html#errors

[11] Franka, Emika. (2019, April 15). franka_ros, Installation. Retrieved from https://frankaemika.github.io/docs/installation.html

[12] Franka, Emika. (2019, April 15). franka_ros, Setting up the real-time kernel. Retrieved from https://frankaemika.github.io/docs/installation.html#setting-up-the-real-time-kernel

[13] Franka, Emika. (2019, April 15). franka_ros, franka_gripper. Retrieved from https://frankaemika.github.io/docs/franka_ros.html#franka-gripper

[14] Franka, Emika. (2019, April 15). franka_ros, franka_gripper. Retrieved from https://frankaemika.github.io/docs/franka_ros.html#franka-hw

[15] Franka, Emika. (2019, April 15). franka_ros, franka_gripper. Retrieved from https://frankaemika.github.io/docs/franka_ros.html#franka-control

[16] Franka, Emika. (2019, April 15). franka_ros, franka_gripper. Retrieved from https://frankaemika.github.io/docs/franka_ros.html#franka-visualization

[17] Franka, Emika. (2019, April 15). franka_ros, franka_gripper. Retrieved from https://frankaemika.github.io/docs/franka_ros.html#franka-example-controllers

[18] Franka, Emika. (2019, April 15). franka_ros, franka_gripper. Retrieved from https://frankaemika.github.io/docs/franka_ros.html#panda-moveit-config

[19] Franka, Emika. (2019, April 12). franka_ros, Writing your own controller. Retrieved from https://frankaemika.github.io/docs/franka_ros.html#writing-your-own-controller

[20] Franka, Emika. (2019, April 12). franka_ros, Signal processing. Retrieved from https://frankaemika.github.io/docs/libfranka.html#signal-processing

[21] Franka, Emika. (2019, April 12). franka_ros, building the ROS packages. Retrieved from https://frankaemika.github.io/docs/install_linux.html#building-the-ros-packages

10 Single Joint Position Control with Constant Velocity Model for Franka Robot - Introduction

Franka-Emika provides "position joint interface" to control Franka robot with joint position control. Via FCI, user given joint position goals in radians are sent to the Control and joints are being moved to the goal positions. But this method is used when joint position control is sufficient and velocity control is not needed. **Our contribution**, with novel "joint position control with constant velocity model" is using Franka robot's joint position interface to control the joints positions while regulating the velocity of the joints. This gives users more control over the joints without compromising the precision of the joints, unlike using velocity joint interface with constant velocities.

The rest of the report is organized as follows: Section 11 gives information about theory of the joint position control with constant velocity model, section 12 shows how to implement novel model into the code, section 27 shows the experiments to show the difference's between the usage of different methods, section 34 discusses the advantages and disadvantages of the novel joint position controller. Finally, section 42 shows how the controller works on Franka robot with a video.

11 Theory of the joint position control with constant velocity model

Objective of joint position control with constant velocity model. When precise joint position control is necessary "joint position interface" is preferred. But joint position interface, that converts and sends the user commands to the hardware, has only one input which is the absolute joint position in radians. So, it is not possible to control the joints' velocities with joint position control. However, sometimes it is not sufficient to use joint position control without controlling the velocity. For example when user desires Franka robot to carry a glass of liquid, it will be logical to not exceed certain velocity, acceleration and jerk limits in order to prevent spillage. Objective of this report is to explain how our novel model is working with joint position interface while controlling the velocity and acceleration of the joints.

Given values. As inputs of the command, user will give exact "joint goal position" in degrees and desired velocity in degrees per second as the "joint's goal velocity". Beware that joint commands has to be in radians while sending to the hardware via position joint interface, but for better user experience degrees are preferred as input from user interface. During runtime they are converted to the radians.

Known values. Current position and velocity of the joints' are known and can be reached from position joint interface's joint handles' that receives readings from `robot_hw`.

Position joint interface works with only one command, `setCommand()`, that takes absolute joint position in radians as input. When a command has been sent to a joint, FCI will try to complete it as soon as possible within the velocity, acceleration and jerk limits of the joints [2, 3]. Limits of the joints are provided in the Hardware Setup for Vision-based Control using Franka Robot Table 3 [1]. Instead of sending one large step, sending small steps with known velocity will let joint to arrive at the given point on time. Here, joint position control with given constant velocity model will be explained.

In ideal conditions where velocity is constant, distance traveled can be calculated as:

$$d_{total} = v_{constant} \cdot t_{total} \tag{1}$$

where d_{total} is the total distance traveled in radians, $v_{constant}$ is the constant velocity in rad/s and t_{total} is the total time spent during this movement in seconds.

$$d_{total} = p_{goal} - p_{current} \tag{2}$$

d_{total} is calculated from the difference between user given joint goal position and current position of the joint read from the `robot_hw`. From (1), equation for the total time in seconds needed to complete the joint movement can be derived as:

$$t_{total} = d_{total}/v_{constant} \tag{3}$$

Under the ideal communication conditions, 1000 user commands per second or 1 message per millisecond are sent from FCI to the Control. But communication is not always at 1 kHz. To prevent confusion under non-ideal states, instead of using numbers, a variable -period- will be used in further explanation of the concept.

$$T = 1/f \tag{4}$$

where period, T, is the time spent for sending one message in seconds and frequency, f, is the number of messages per second (communication rate). Distance traveled during one period can be calculated as:

$$x_{period} = v_{constant} \cdot T = v_{constant}/f \tag{5}$$

x_{period} can also be considered as velocity per period, which denoted as v_{period} (rad/T).
Equation (1) and (5) can be combined to get:

$$d_{total} = v_{period} \cdot f \cdot t_{total} \tag{6}$$

and it can be re-written as:

$$d_{total} = \Sigma_{t=1}^{n} v_{period} \tag{7}$$

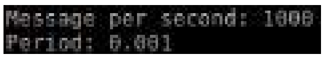
where n equals to $f \cdot t_{total}$ which is the number of the total periods needed for the joint to go to the desired goal point.

12 Implementation of the joint position control with constant velocity model

Since theoretical values are known, now the model will be implemented into the code.

As mentioned in section 11, theoretically user commands are sent at 1 kHz to the Control but it is not always the case. As the load on CPU gets higher this value can change. The time passed between two messages can be obtained from `period` reference.

If we print out using
`std::cout « "Message per second:: " « (1/period.toSec()) « std::endl;` and
`std::cout « "period: " « (period.toSec()) « std::endl;`



```
Message per second: 1000
Period: 0.001
```

Figure 15: Number of messages send per second and seconds passed for sending one message.

seconds passed for sending one message, and number of messages sent per second can be obtained (figure: 15).

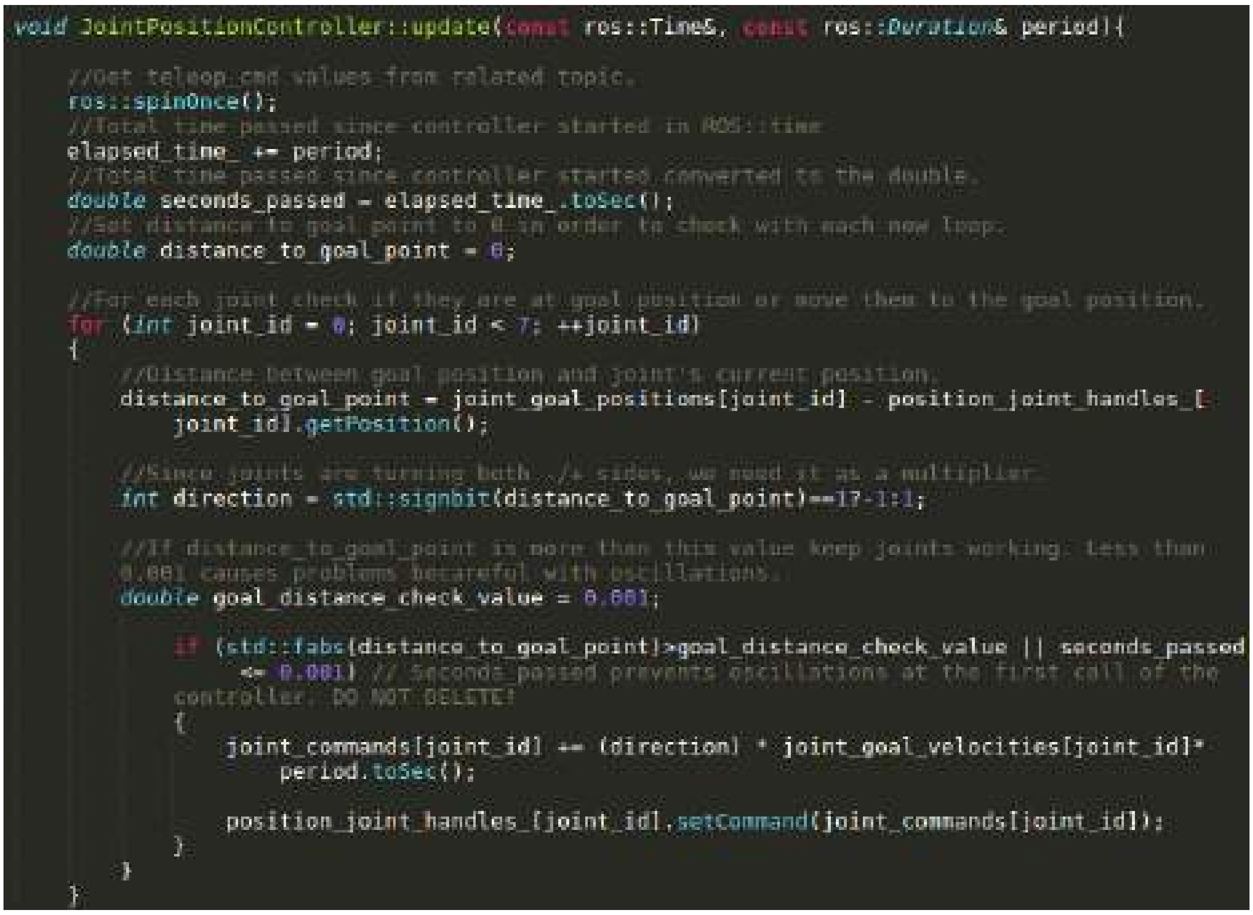
joint_goal_positions[joint_id] is the message of the goal position of the joint that has been given by the user in degrees, which converted into radians, received from teleop_cmd node. It is mentioned in equation (2) as p_{goal}. (Details of the teleop_cmd will be given in the next report.)

position_joint_handles_[joint_id].getPosition() is the current position of the joint, read by the joint position interface's handle from `robot_hw`. It is mentioned in equation (2) as $p_{current}$.

distance_to_goal_point is the difference between joint's goal and current positions that mentioned in equation (2).

direction is the sign of the `distance_to_goal_point`. Since joints can rotate in clock wise and counter clock wise directions `distance_to_goal_point` can be positive or negative. It is used for adding or subtracting the new position to the current position of the joint.

joint_goal_velocities[joint_id] is the message of the goal velocity of the joint that has been given by the user in *degrees/second*, which converted into *radians/second*, received from teleop_cmd node. When multiplied with $period.toSec()$, x_{period} mentioned in equation (5) will be achieved.



```
void JointPositionController::update(const ros::Time&, const ros::Duration& period){

    //Get teleop_cmd values from related topic.
    ros::spinOnce();
    //Total time passed since controller started in ROS::time
    elapsed_time_ += period;
    //Total time passed since controller started converted to the double.
    double seconds_passed = elapsed_time_.toSec();
    //Set distance to goal point to 0 in order to check with each new loop.
    double distance_to_goal_point = 0;

    //For each joint check if they are at goal position or move them to the goal position.
    for (int joint_id = 0; joint_id < 7; ++joint_id)
    {
        //Distance between goal position and joint's current position.
        distance_to_goal_point = joint_goal_positions[joint_id] - position_joint_handles_[
            joint_id].getPosition();

        //Since joints are turning both -/+ sides, we need it as a multiplier.
        int direction = std::signbit(distance_to_goal_point)==1?-1:1;

        //If distance_to_goal_point is more than this value keep joints working. Less than
        0.001 causes problems becareful with oscillations.
        double goal_distance_check_value = 0.001;

            if (std::fabs(distance_to_goal_point)>goal_distance_check_value || seconds_passed
              <= 0.001) // Seconds_passed prevents oscillations at the first call of the
            controller. DO NOT DELETE!
            {
                joint_commands[joint_id] += (direction) * joint_goal_velocities[joint_id]*
                    period.toSec();

                position_joint_handles_[joint_id].setCommand(joint_commands[joint_id]);
            }
        }
    }
```

Figure 16: update() function, calculates error and moves the joints towards the goal position.

For each joint, the `distance_to_goal_point` is checked. Direction of the rotation will be needed when calculating the command sent to the joint. Only difference between model and code is, total time needed to arrive to the goal position is not calculated. Instead, it is being checked if joint arrived at goal position or not. Till we reach the goal position, $direction \cdot x_{period}$ added to the last command as it is derived in equation (7).

```
    joint_commands[joint_id] +=
(direction) *joint_velocity_limits[joint_id]*period.toSec();
```

Since `period` is in `ros::Time` format which cannot be multiplied with other formats, `toSec()` method is used to convert it into the seconds which is in double format.

13 The Experiments

13.1 Comparison Experiments:

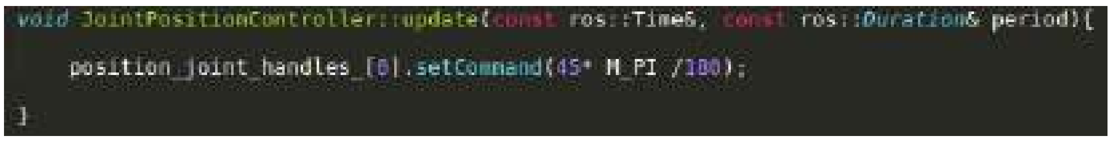
Comparison experiments has been held to demonstrate the differences between Desk!, directly setting command with `setCommand()` and our novel model.

Comparison experiment setup: With each method, joint 1 is sent from 0-45 and 0-90 degrees. Desk! is set to 100% speed and acceleration. `setCommand()` is given exact joint positions of 45 and 90 degrees converted to radian. When position joint interface's `setCommand()` is used, Franka robot will try to complete it as soon as possible within the velocity, acceleration and jerk limits of the joints [1, 2, 3], so it reaches it's maximum velocity as soon as possible. Our novel method is given exact joint positions of 45 and 90 degrees converted to radians. For velocity 120 degrees (2.1 radians) is given.

Name	Joint 1	Unit
maximum position in ccw direction	2.8973	rad
minimum position in cw direction	-2.8973	rad
maximum velocity	2.1750	rad/s
maximum acceleration	15	rad/s^2
maximum jerk	7500	rad/s^3
maximum torque	87	$N \cdot m$
maximum rotatum	1000	$N \cdot m/s$

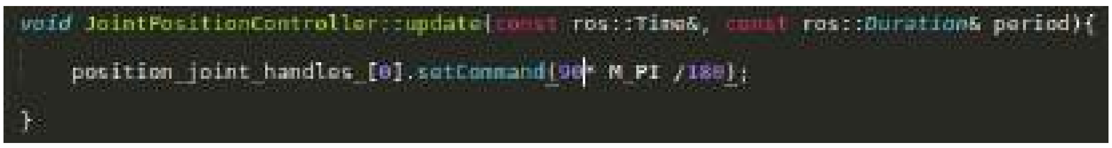
Table 7: Position, velocity, acceleration, jerk, torque, rotatum limits of each joint 1 of the Franka robot [4].

Joint 1 is preferred for this comparison because observing the first joint is easier than the other joints.



```
void JointPositionController::update(const ros::Time&, const ros::Duration& period){
    position_joint_handles_[0].setCommand(45* M_PI /180);
}
```

Figure 17: Using `setCommand()`, joint 1 sent to 45 degrees (converted to radian) command to joint 1.



```
void JointPositionController::update(const ros::Time&, const ros::Duration& period){
    position_joint_handles_[0].setCommand(90* M_PI /180);
}
```

Figure 18: Using `setCommand()`, joint 1 sent to 90 degrees (converted to radian) command to joint 1.

In figure 17 and 18 code for changing position of the joint 1 from 0 degree to 45 and 90 degrees using `setCommand()` has been shown.

There are 3 phases of joint movement with position control:

1. Constant acceleration
2. Constant velocity
3. Constant deceleration

13.1.1 Desk!

It is not possible to print out joint velocity / time graph of the motion from Desk! using rqt_plot. In order to make the results as close as possible to the real results, tests of the each angle were repeated 3 times and time to complete each step is measured with a chronometer. Then average of the 3 trials were taken as the result.

13.1.2 setCommmand()

When exact goal position of the joint is sent to the `setCommand()`, FCI tries to complete it as soon as possible. Graphs are made using rqt_plot so they are showing the results as close as possible to the reality.

setCommmand(45) 45 degrees converted into radians and then command sent. The graph of velocity (*rad/s*) and position (*radians*) of the joint 1 has been plotted (figure 20) with rqt_plot. In figure 75 zones mentioned above has been depicted.

Now let's focus on figure 75: Red zone on the left side shows the acceleration, blue zone in the middle shows the constant velocity, red zone on the right side depicts the constant deceleration.

And average velocity from equation (1):

$$v_{average} = 45 * \pi/180/(44.7 - 44.15) = 1.4280 rad/s \tag{8}$$

From table 7, velocity limit of joint 1 is 2.1750 rad/s. 1.4280/2.1750 = %65.655 of $v_{maximum}$ of joint 1.

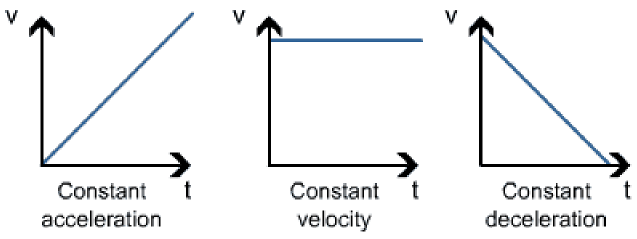


Figure 19: 3 phases of the joint movement: Constant acceleration, constant velocity, constant deceleration [5].

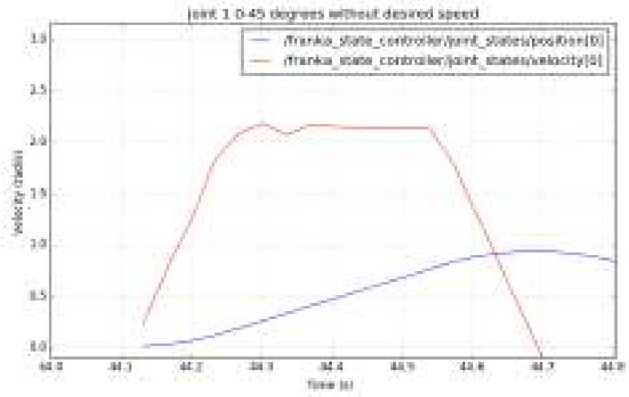


Figure 20: Joint 1 0-45 degree movement without user specified constant velocity.

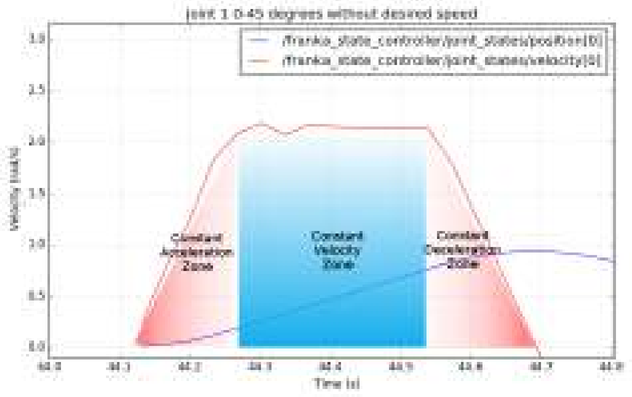


Figure 21: Constant acceleration, velocity and deceleration zones.

setCommmand(90) 90 degree converted into radians and then command sent to the joint 1. With rqt_plot, the graph of velocity (rad/s) and position ($radians$) of the joint 1 has been plotted (figure 22). In figure 23 zones mentioned above has been depicted.

Indicated at the figure 23: Red zone on the left side shows the acceleration, blue zone in the middle shows the constant velocity, red zone on the right side depicts the constant deceleration.

And average velocity from equation (1):

$$v_{average} = 90 * \pi/180/(218.0 - 217.10) = 1.41 rad/s \tag{9}$$

1.41/2.1750 = %64.827 of $v_{maximum}$ of joint 1.

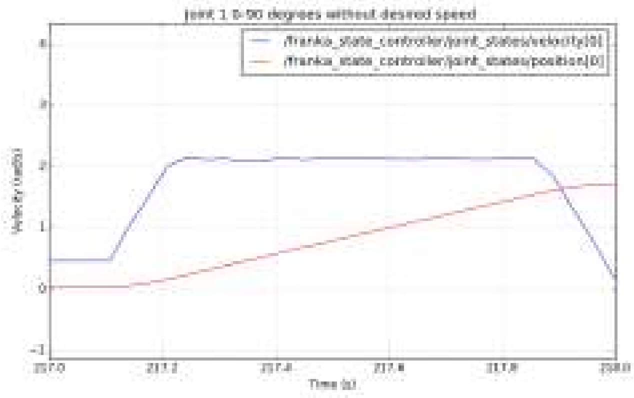


Figure 22: Joint 1 0-90 degree movement without user specified constant velocity.

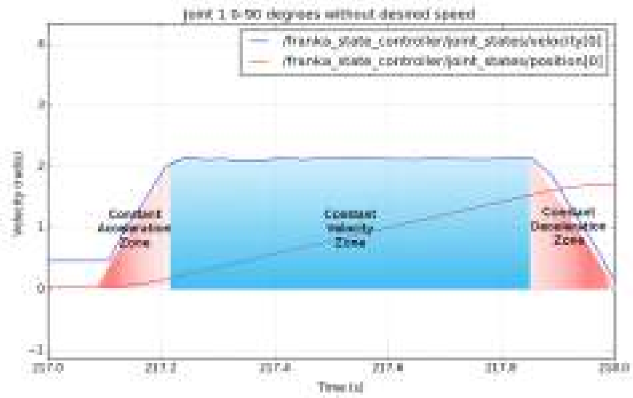


Figure 23: Constant acceleration, velocity and deceleration zones.

For both trials it was not possible to control the velocity of the joint. Joint reached it's maximum velocity after acceleration, and started to decelerate. It can be observed that even if the goal position changes, $v_{average}$ is getting close to the 1.4 rad/s. From the graphs, 0-45 degree command is completed in 0.6 seconds and 0-90 degree command is completed in 1.0 second. In both cases when the joint reached to the goal position it exceeded the goal point and this caused an oscillation. When relatively large distances are direclty sent as a command, working with `setCommand()` is not possible.

13.1.3 Novel model

Our novel model is tested with 45 and 90 degrees with 120 deg/s (2.1 rad/s) commands. 0-45 degree command is completed in 0.6 second (figure 24) and 0-90 degree command is completed in 0.9 second (figure 25). In both cases when the joint reached to goal position it exceeded the goal point and this caused an oscillation. Results show that our novel method is not suitable for working with high velocities.

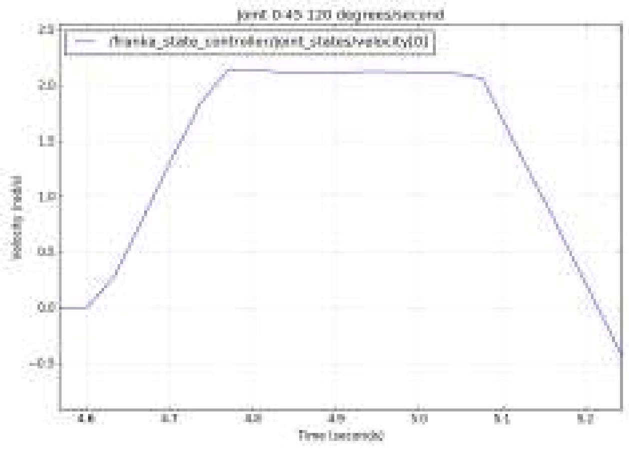


Figure 24: Joint 1 0-45 degree movement with user specified 120 rad/s constant velocity.

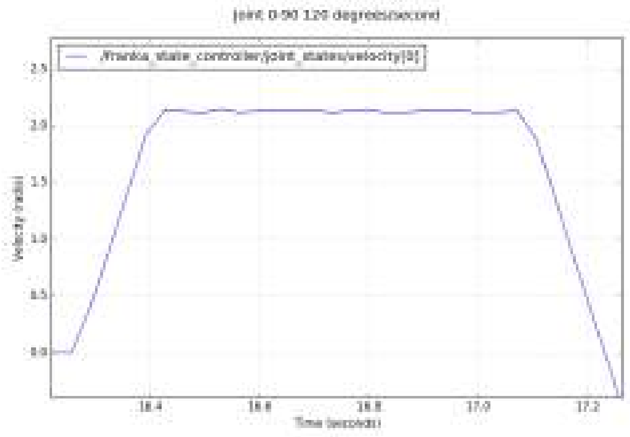


Figure 25: Joint 1 0-90 degree movement with user specified 120 rad/s constant velocity.

13.1.4 Comparison table

In order to compare the results, a comparison table has been prepared.

	0-45	**0-90**	**Oscillation**
Desk! (second)	1.22	1.87	No
setCommand() (second)	0.6	0.9	Yes
Novel model (second)	0.6	1.0	Yes

Table 8: Desk!, setCommand(), Novel model comparison, 0-45 and 0-90 degree distance completion times in seconds.

As it can be seen from table 8, Desk! is almost 50% slower than sending commands with `setCommand()` and our novel model. But both `setCommand()` and our novel model is starting to oscillate when they reach to the joint 1's goal position. It can be said that both methods cannot be used like this. Now we will try to find the maximum velocities that Franka robot's joints can work with our novel model.

13.2 Novel joint position control with constant velocity:

In order to find highest velocities possible every joint will be tested with 3,5,10 deg/s commands if it is possible. If oscillation reaches to a point which it is not safe to use joint at that velocity trial will be stopped.

Joint 1 is tested with 45 degree movement by 3,5,10 deg/s commands.

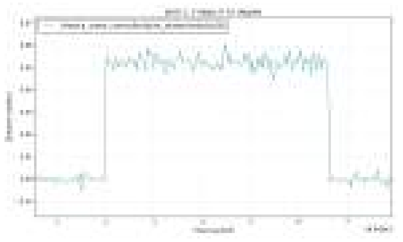


Figure 26: Joint 1: 0-15 degrees with 3 deg/s velocity. Completed in less than 5 seconds.

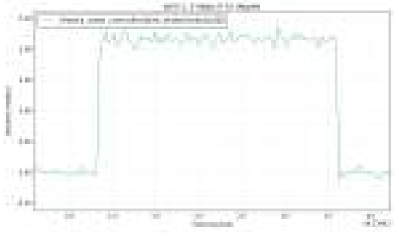


Figure 27: 0-15 degrees with 5 deg/s velocity. No visible oscillation. Completed in less than 3 seconds.

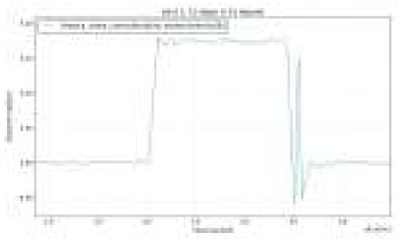


Figure 28: 0-15 degrees with 10 deg/s velocity. Completed in 1.5 seconds. Oscillation is visible on the right side of the chart.

From the experiments, it can be seen that our novel joint position control method works good when joint velocity is less than 10 deg/s (figures 80, 81). Oscillation is becoming visible at the right side of the chart on figure 82.

Joint 2: Joint 2 is prone to the oscillation when given joint velocity is over 3 deg/s. Only joint velocities around 3 deg/s is safe to use 83. Large waves of oscillation depicted in figures 84 and 85.

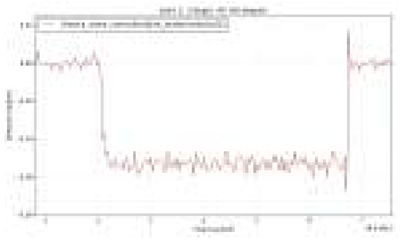


Figure 29: Joint 2: -45-60 degrees with 3 deg/s velocity. Completed in almost 5 seconds.
Slightly visible oscillation.

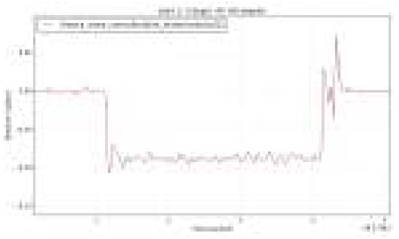


Figure 30: 0-15 degrees with 5 deg/s velocity. Completed in almost 3 seconds. Visible oscillation.

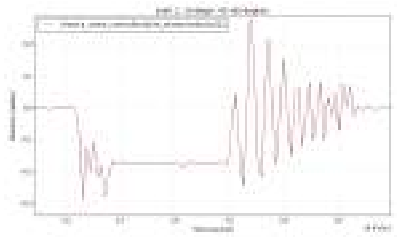


Figure 31: 0-15 degrees with 10 deg/s velocity. Completed in almost 1.5 seconds. Oscillation is too high for safe working.

Joint 3: Joint 3 starts oscillating when given joint velocity is more than 3 deg/s. Only joint velocities less than 10 deg/s are safe to use (figures 32, 33). Large wave of oscillation can be seen in figure 34.

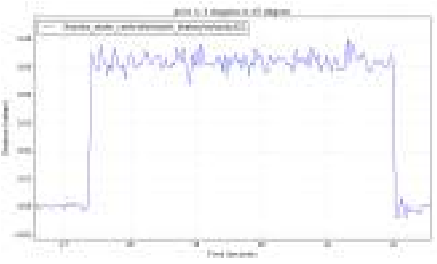


Figure 32: Joint 3: 0-15 degrees with 3 deg/s velocity. Completed in less than 5 seconds. A small wave of oscillation is visible on the right side of the chart.

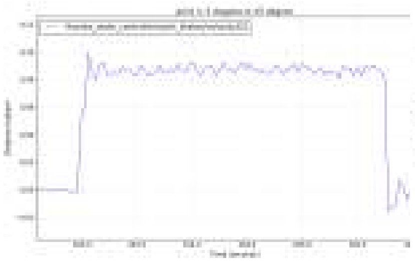


Figure 33: Joint 3: 0-15 degrees with 5 deg/s velocity. Completed in less than 3 seconds. A wave of oscillation is visible on the right side of the chart.

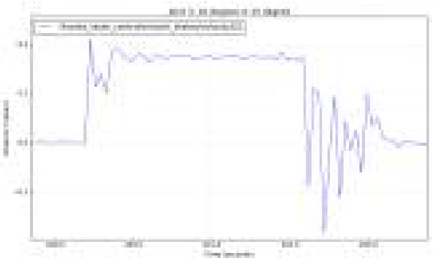


Figure 34: Joint 3: 0-15 degrees with 10 deg/s velocity. Completed in almost 1 second. Large wave of oscillation is visible on the right side of the chart.

Joint 4: A medium wave of oscillation on the right side of the image 90 is visible. On image 91, a large wave of oscillation is visible, 10 *deg*/*s* is not safe to use for joint 4.

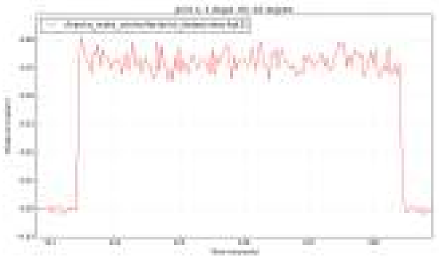


Figure 35: Joint 4: -60 to -45 degrees with 3 *deg*/*s* velocity. Completed in almost 5 seconds. No visible oscillation.

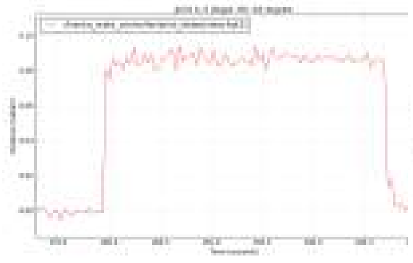


Figure 36: Joint 4: -60 to -45 degrees with 5 *deg*/*s* velocity. Completed in almost 3 seconds. No visible oscillation.

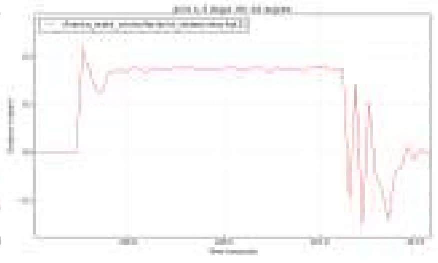


Figure 37: Joint 4: -60 to -45 degrees with 10 *deg*/*s* velocity. Completed in almost 1.5 seconds. Large wave of oscillation on the right side of the chart.

Joint 5: Joint 5 has a small wave of oscillation at 3 and 5 *deg*/*s* (figures 92 and 93). Relatively large wave at 10 *deg*/*s* (figure 94). Velocities less than 10 degree/second is safe to use with Joint 5.

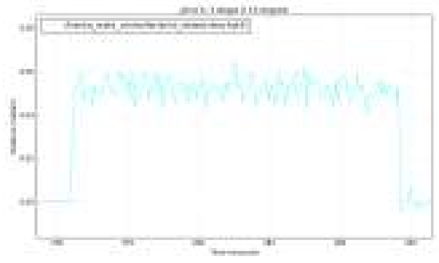


Figure 38: Joint 5: 0-15 degrees with 3 *deg*/*s* velocity. ompleted in less than 5 seconds. A small wave of oscillation is visible on the right side of the chart.

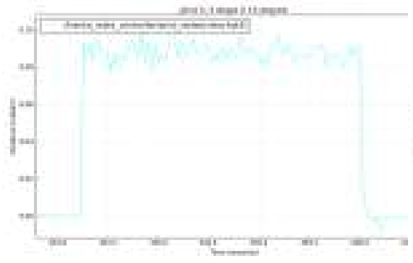


Figure 39: Joint 5: 0-15 degrees with 5 *deg*/*s* velocity. ompleted in almost 1.5 seconds. A small wave of oscillation is visible on the right side of the chart.

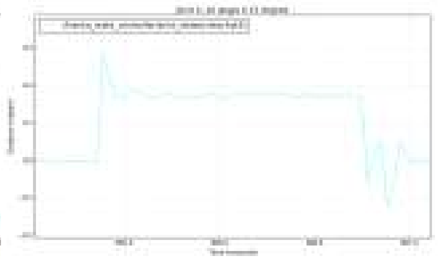


Figure 40: Joint 5: 0-15 degrees with 10 *deg*/*s* velocity. A large wave of oscillation is visible on the left side of the image.

Joint 6: Joint 6 does not show any sign of oscillation even at high velocities such as 10 *deg*/*s*. Timings are close to the theoretical values so that joint 6 can be used with velocities up to 10 *deg*/*s* without problem.

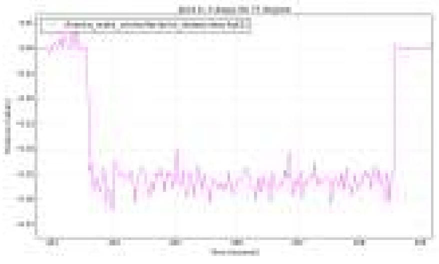


Figure 41: Joint 6: 90-75 degrees with 3 *deg*/*s* velocity. Completed in less than 5 seconds. No oscillation is visible on the chart.

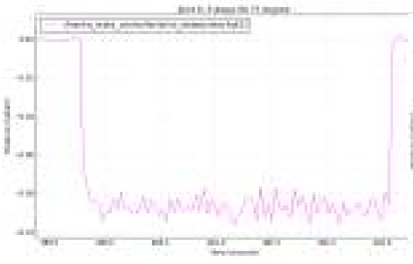


Figure 42: Joint 6: 90-75 degrees with 5 *deg*/*s* velocity. Completed in less than 3 seconds. No oscillation is visible on the chart.

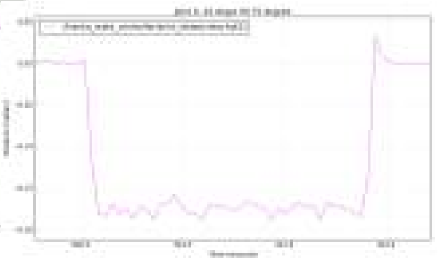


Figure 43: Joint 6: 90-75 degrees with 10 *deg*/*s* velocity. Completed in almost 1.5 seconds. A small wave of oscillation is visible on the left side of the image.

Joint 7: Joint 7 also does not show any sign of oscillation even at high speeds such as 10 *deg*/*s*. Timings are close to the theoretical values so that joint 7 can be used with speeds up to 10 *deg*/*s* without problem.

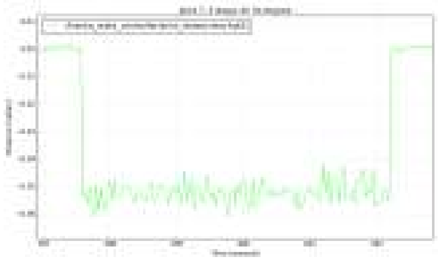


Figure 44: Joint 6: 45-30 degrees with 3 deg/s velocity. Completed in less than 5 seconds. No oscillation is visible on the chart.

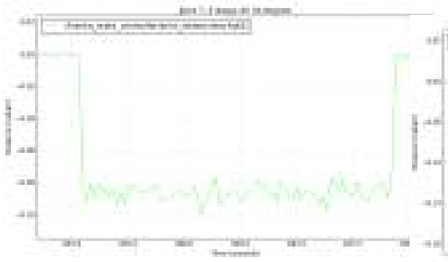


Figure 45: Joint 6: 45-30 degrees with 5 deg/s velocity. Completed in less than 3 seconds. No oscillation is visible on the chart.

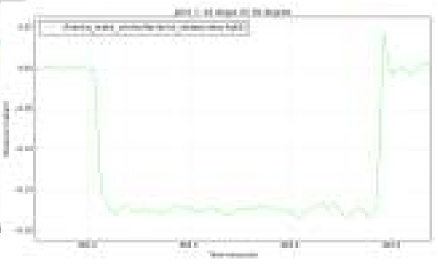


Figure 46: Joint 6: 45-30 degrees with 10 deg/s velocity. Completed in exactly 1.5 seconds. A small wave of oscillation is visible on the left side of the image.

14 Conclusion

Our novel method of "joint position control with constant velocity" is working well with relatively slow velocities. Under 5 deg/s every joint is acting precise in velocity and in position. Joint 6 and 7 is working without oscillation at 10 deg/s velocity. When Desk! is used as a reference, Joint 1 reached to 45 degree in 1.22 seconds and 90 degree in 1.87 seconds when the speed and the acceleration is chosen maximum. This shows that our novel model is working around 75% slower than possible limits of the joints. In the original franka_controller_examples' joint position and joint velocity controllers, movements are spherical and velocity is not changing drastically when the joints change the direction of the movement. In order to solve this, our controller should send slower speeds during acceleration and deceleration.
Next report shows the details of the coding a joint position controller and then the other report, a new method including softer acceleration and deceleration to solve oscillation problems will be proposed.

15 Video review

I made a review video about joint position control with constant velocity model. Oscillation of the joints are visible in the video.
You can reach the video from this link: `https://youtu.be/sFvnDI_F4NA`

Figure 47: A screenshot from review video.

16 Code

Code can be reached from:
`https://github.com/berktepebag/franka_robot_controllers/commits/joint_position_controller`

References

[1] Tepebag, B. (2019, May 20). Hardware Setup for Vision-based Control using Franka Robot

[2] Franka, Emika. (2019, April 12). libfranka, Real Time Commands. Retrieved from
https://frankaemika.github.io/docs/libfranka.html#realtime-commands

[3] Franka, Emika. (2019, April 12). Real-time joint space control. Retrieved from
https://github.com/frankaemika/franka_ros/issues/35

[4] Denavit-Hartenberg Parameters Note. Reprinted from Franka Control Interface (FCI) (2019, April 5) Retrieved from
https://frankaemika.github.io/docs/control_parameters.html

[5] S-cool. (2019, June 06). The Basics of Linear Motion and Displacement and Velocity Time-Graphs. Retrieved from
https://www.s-cool.co.uk/a-level/physics/vectors-and-scalars-and-linear-motion/revise-it/the-basics-of-linear-motion-and-disp

[6] Tepebag, B. (2019, May 20). Software Setup for Vision-based Control using Franka Robot

17 Writing Joint Position Controller for Franka Robot - Introduction

This report gives information about ros_control package and shows how to write a "Joint Position Controller". At the end of this report you will be able to control Franka robot with sending joint positions' commands.

The rest of the report is organized as follows: Section 18 gives information about ros_control package and section 32 shows how to write your own joint position controller.

18 ros_control Package

ros_control is a C++ library that has been developed to decrease the time spent on writing controllers. With standardization of controllers users can focus on solving their problems rather than solving the problems of controlling the robot [1, 2].

Franka-Emika GmbH provides ros_control framework based on the libfranka API, called franka_hw [3]. A list of the available controllers to the user are given in the Software Setup for Vision-based Control using Franka Robot report, Table 1 [6].

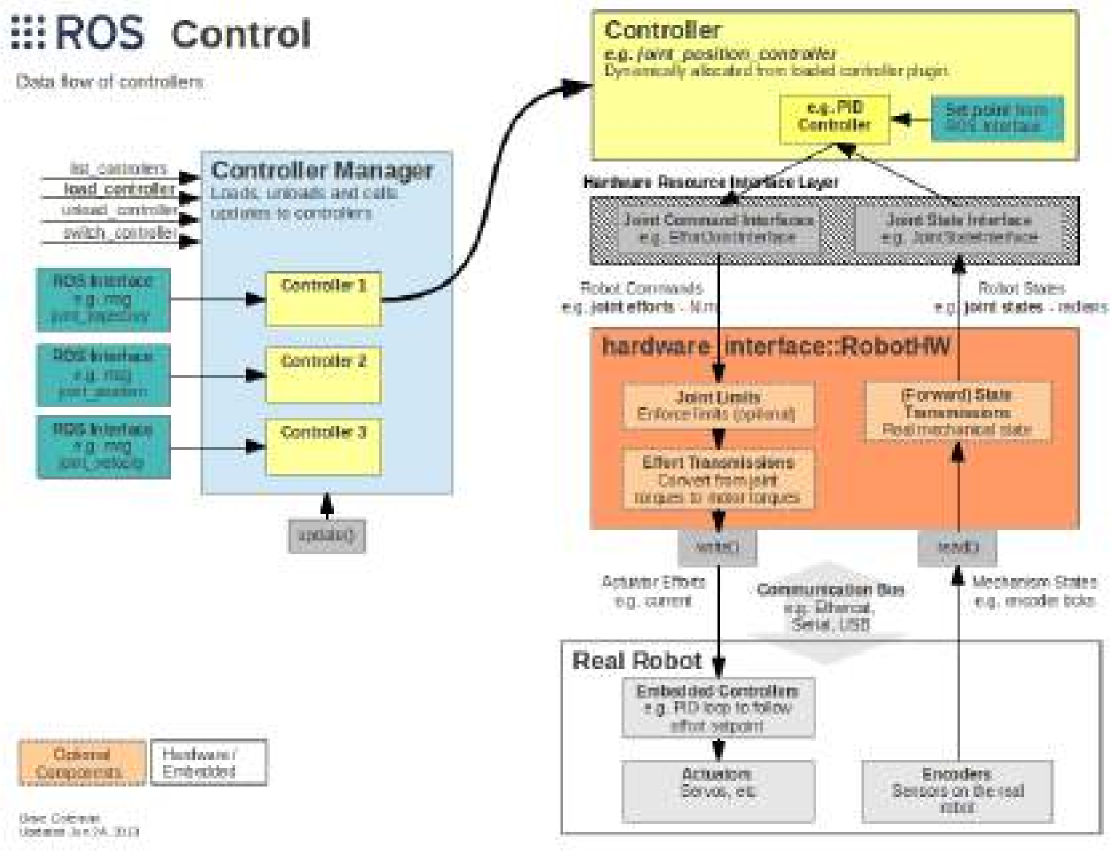

ROS Control
Data flow of controllers
list_controllers
load_controller
unload_controller
switch_controller
Controller Manager
Loads, unloads and calls updates to controllers
ROS Interface e.g. msg joint_trajectory
ROS Interface e.g. msg joint_position
ROS Interface e.g. msg joint_velocity
Controller 1
Controller 2
Controller 3
update()
Controller
e.g. joint_position_controller
Dynamically allocated from loaded controller plugin
e.g. PID Controller
Set point from ROS interface
Hardware Resource Interface Layer
Joint Command Interface e.g. EffortJointInterface
Joint State Interface e.g. JointStateInterface
Robot Commands e.g. joint efforts - N.m
Robot States e.g. joint states - radians
hardware_interface::RobotHW
Joint Limits Enforce limits (optional)
Effort Transmissions Convert from joint torques to motor torques
(Forward) State Transmissions Read mechanical state
write()
read()
Actuator Efforts e.g. current
Communication Bus e.g. Ethercat, Serial, USB
Mechanism States e.g. encoder ticks
Real Robot
Embedded Controllers e.g. PID loop to follow effort setpoint
Actuators Servos, etc
Encoders Sensors on the real robot
Optional Components
Hardware / Embedded
Dave Coleman
Updated Jun 24, 2013

Figure 48: Overview of the ros_control package [1].

Controller is a dynamically loadable plug-in which receives commands and sends it to robot hardware via ros_control interfaces (depicted on the right top corner of figure 48). Controller do not talk to robot hardware directly and requires resources in order to send commands to the joints [1, 5].

hardware_interface is the middleware between ros_control and the hardware (depicted on the right middle of figure 48). In order to control a robot, ros_control uses a controller or combination of controllers and hardware interface to send and receive commands to the robot hardware [1]. Franka-Emika GmbH, provides franka_hw package, which involves the hardware abstraction of the Franka robot for the ros_control framework based on the libfranka API [3].

Controller manager is a ROS node that can access to the joint states of the robot, and to the commands sent to it (depicted on the left side of figure 48). Controller manager, loads, unloads and calls updates to the controllers [6]. During their initialization, controllers request resources from the hardware interface. Controller manager is responsible from keeping the track of those requests and knows which controller requested which resources. By this information controller manager prevents different controllers to use the same resource at the same time. Relation between higher to lower level languages depicted in the figure 49.

NodeHandle is responsible from startup and shutdown of the internal node inside a roscpp program. To be able to use ros_control interfaces, resource handles must be retrieved by name. In JointPosisitonInterface example, joint names are panda_joint1, panda_joint2, ..., panda_joint7. A list of resource handle names is given in the Software Setup for Vision-based Control using Franka Robot report [6], Table 2: List of the resource handle names of the controller interfaces.

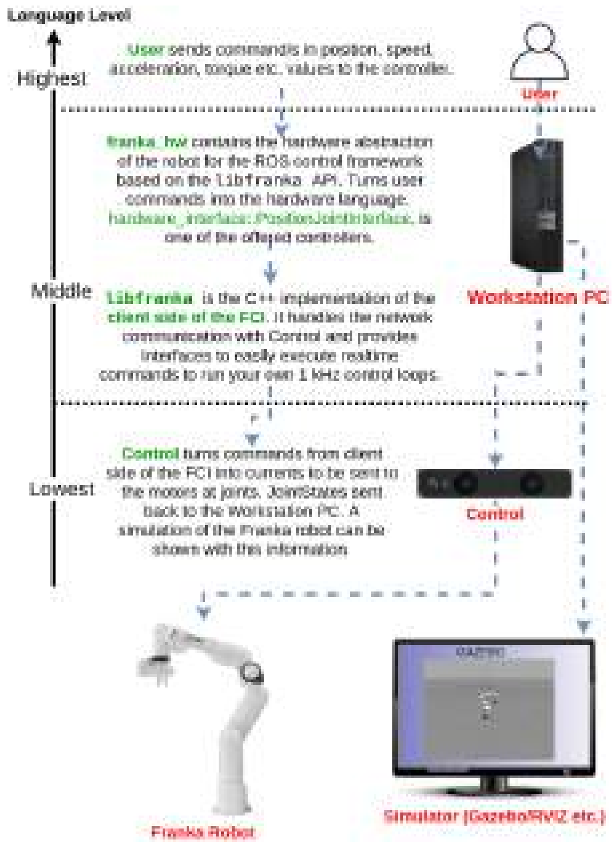
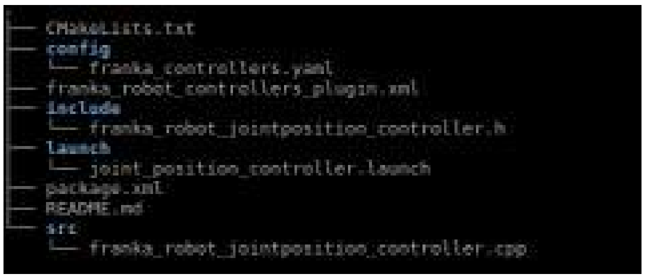

Language Level
Highest
User sends commands in position, speed, acceleration, torque etc. values to the controller.
User
franka_hw contains the hardware abstraction of the robot for the ROS control framework based on the libfranka API. Turns user commands into the hardware language. hardware_interface::PositionJointInterface, is one of the offered controllers.
Middle
libfranka is the C++ implementation of the client side of the FCI. It handles the network communication with Control and provides interfaces to easily execute realtime commands to run your own 1 kHz control loops.
Workstation PC
Lowest
Control turns commands from client side of the FCI into currents to be sent to the motors at joints. JointStates sent back to the Workstation PC. A simulation of the Franka robot can be shown with this information.
Control
Franka Robot
Simulator (Gazebo/RVIZ etc.)

Figure 49: Language level diagram showing communication between hardware and software.

19 Writing your own "Joint Position Controller"

In order to write "Joint Position Controller", Franka-Emika GmbH's "Writing your own controller" tutorial [7] and ros_control tutorial [8] is followed . It took a while to make controller work due to naming conventions in both tutorials were not clear. This problem is taken into consideration and naming of the files and variables done in order to make it easy to follow this tutorial. Joint Position Controller node directory will look like figure 50 at the end of this report.



```
├── CMakeLists.txt
├── config
│   └── franka_controllers.yaml
├── franka_robot_controllers_plugin.xml
├── include
│   └── franka_robot_jointposition_controller.h
├── launch
│   └── joint_position_controller.launch
├── package.xml
├── README.md
└── src
    └── franka_robot_jointposition_controller.cpp
```

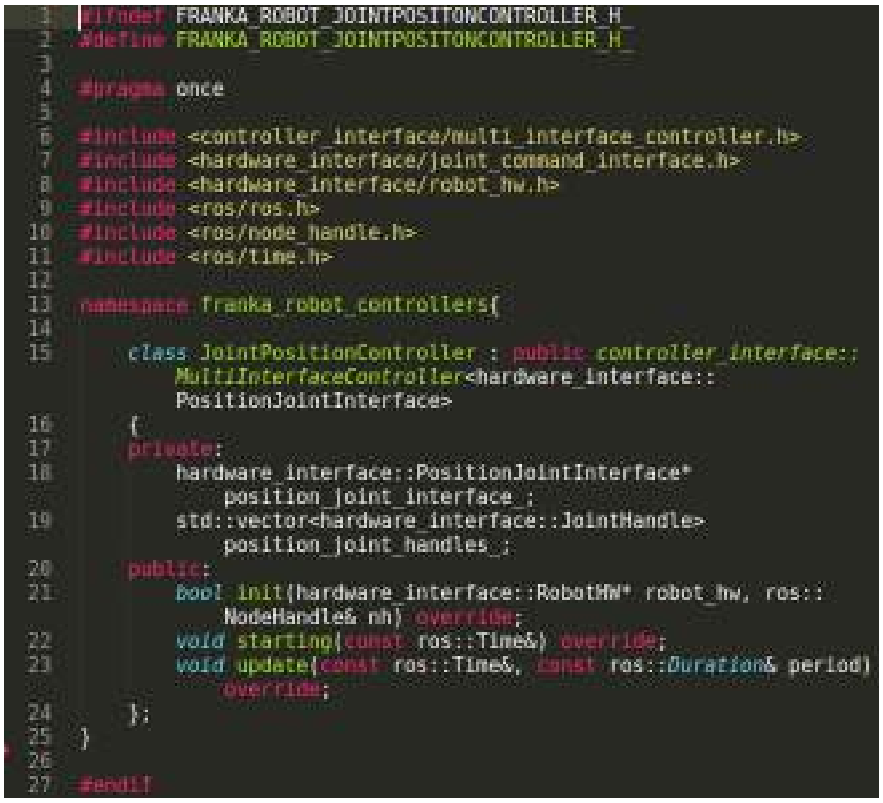
Figure 50: Package directory, file tree.

ROS Package: First create a new ROS package named "franka_robot_controller_pi" under franka_ros folder in catkin workspace. (To be exact, it is possible to chose any name but be careful not to use same naming for namespace in the header file and the project/package name.)

- `cd ~/catkin_ws/src/franka_ros`
- `catkin_create_pkg franka_robot_controller_pi`

Here, "pi" stands for Pascal Institute and it discriminates the package from original franka_ros packages. There is no need to declare dependencies right now since they will be added as needed.

19.1 franka_robot_jointposition_controller.h



```cpp
#ifndef FRANKA_ROBOT_JOINTPOSITONCONTROLLER_H_
#define FRANKA_ROBOT_JOINTPOSITONCONTROLLER_H_

#pragma once

#include <controller_interface/multi_interface_controller.h>
#include <hardware_interface/joint_command_interface.h>
#include <hardware_interface/robot_hw.h>
#include <ros/ros.h>
#include <ros/node_handle.h>
#include <ros/time.h>

namespace franka_robot_controllers{

    class JointPositionController : public controller_interface::MultiInterfaceController<hardware_interface::PositionJointInterface>
    {
    private:
        hardware_interface::PositionJointInterface* position_joint_interface_;
        std::vector<hardware_interface::JointHandle> position_joint_handles_;
    public:
        bool init(hardware_interface::RobotHW* robot_hw, ros::NodeHandle& nh) override;
        void starting(const ros::Time&) override;
        void update(const ros::Time&, const ros::Duration& period) override;
    };
}

#endif
```

Figure 51: franka_robot_jointposition_controller.h file.

First create the include folder under the package folder and the header file inside the folder:

- `mkdir include`
- `cd include`
- `subl franka_robot_jointposition_controller.h` (instead of subl you can use your favorite text editor, gedit, nano etc.)

Header file is depicted in figure 51. Let's discuss the code.

Included libraries are:

<controller_interface/multi_interface_controller.h> Allows to claim up to four interfaces in one controller instance. Available interfaces have been given in Software Setup for Vision-based Control using Franka Robot report [6], section 4.2.3 franka_hw. Possible combinations of the interfaces are:

1. all possible single interface claims
2. EffortJointInterface + PositionJointInterface
3. EffortJointInterface + VelocityJointInterface
4. EffortJointInterface + FrankaCartesianPoseInterface
5. EffortJointInterface + FrankaCartesianVelocityInterface
6. EffortJointInterface
7. EffortJointInterface + FrankaCartesianPoseInterface
8. EffortJointInterface + FrankaCartesianVelocityInterface

For "Joint Position Controller" PositionJointInterface is sufficient. Combinations of the interfaces are not necessary in our project.

<hardware_interface/joint_command_interface.h> Hardware interface to support commanding an array of joints [9].

<hardware_interface/robot_hw.h> Robot Hardware Interface and Resource Manager. This class provides a standardized interface to a set of robot hardware interfaces (PositionJointInterface, VelocityJointInterface, etc.) to the controller manager. It checks conflicts between controllers and provides a map of hardware interfaces. It is intended to be used as a base class for abstracting custom robot hardware [10].

<ros/ros.h> generic ROS library

<ros/node_handle.h> roscpp's interface for creating subscribers, publishers, etc. This class is used for writing nodes. It provides RAII interface which is acronym of the "resource acquisition is initialization" technique. The idea is representing a resource by a local object, so that whenever a NodeHandle is created for the first time, it instantiates necessary resources. And when destructor called it releases the resources. This prevents problems caused by unreleased resources [11, 12, 13].

<ros/time.h> Time library where user can get a specific moment or a period of time.

These are the libraries needed to start writing a controller. Now declare the namespace, class and the functions.

namespace Naming of the namespace is very important. This is what plugin is looking for with the class name. In the tutorials they chose same name for the project name and namespace which causes a great confusion. Give a different name than project name and call it "franka_robot_controllers".

class `JointPositionController` class is inherited from `controller_interface::MultiInterfaceController`'s `PositionJointInterface`. It will be used to send joint position commands to the `hardware_interface` and then `hardware_interface` sends commands to the control.

Position Joint Interface and Handle A private pointer to the position joint interface and a private JointHandle vector declared. Their usage will be explained at .cpp file.

Functions To be able to create a controller, `update()` and `init()` functions must be implemented [7]. `starting()` and `stopping()` are good for robot to start and stop safely and smoothly, but they are not a must.

init() `init()` function takes `hardware_interface` and `NodeHandle` as inputs.
This is where `node_handles` used for getting the joint names and filling the `position_joint_handles_` vector with joint handles.

starting() `starting()` takes `Time` reference as input.

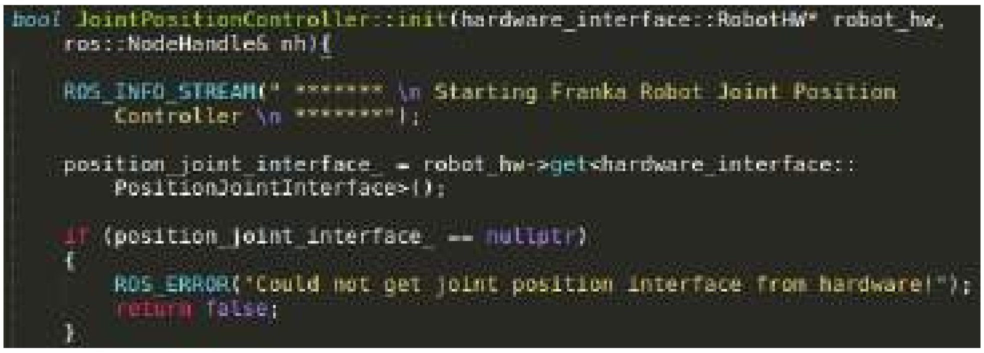
update() `update()` takes `Time` and `Duration` reference as input. With duration reference it is possible to check duration of the command completion. Duration was mentioned in "Single Joint Position Control with Constant Velocity Model for Franka Robot" report for calculating joint position commands per millisecond.

19.2 franka_robot_jointposition_controller.cpp

Necessary class and function decelerations made in the header file. Now it is time to fill in these functions to control Franka robot with `JointPositionController`.

19.2.1 JointPositionController::init() function

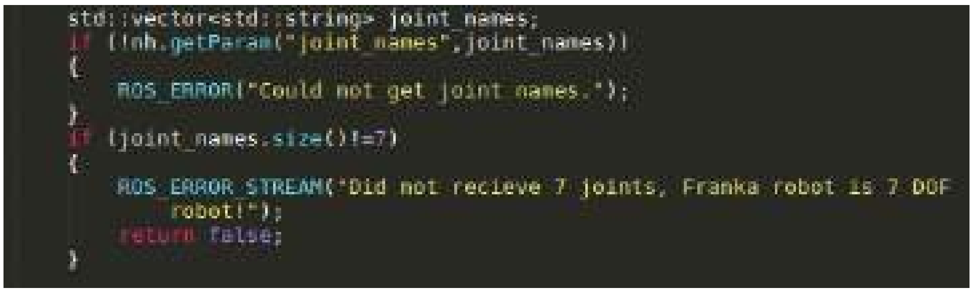
position_joint_interface_ is used to reach `hardware_interface` in order to get resource handles. Check if `PositionJointInterface` exists, if it does not exist it will return a null pointer which means there is no reachable hardware interface so it is not possible to control the Franka robot (figure 52).



```
bool JointPositionController::init(hardware_interface::RobotHW* robot_hw,
    ros::NodeHandle& nh){

    ROS_INFO_STREAM(" ******* \n Starting Franka Robot Joint Position
        Controller \n *******");

    position_joint_interface_ = robot_hw->get<hardware_interface::
        PositionJointInterface>();

    if (position_joint_interface_ == nullptr)
    {
        ROS_ERROR("Could not get joint position interface from hardware!");
        return false;
    }
```

Figure 52: franka_robot_jointposition_controller.cpp file, initialization of position_joint_interface_ and checks if it exists.

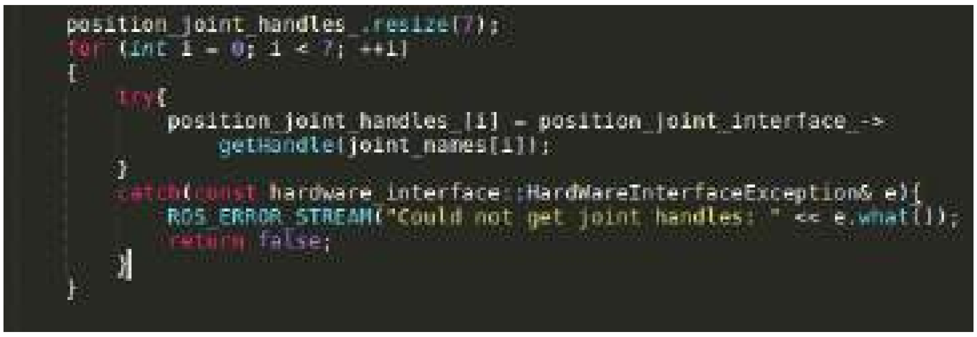
joint_names vector Create a vector to get the joint names from the node_handle. It returns a ROS_ERROR if it cannot get the names. Then check if vector size equals to seven, if not return false (figure 53).



```
std::vector<std::string> joint_names;
if (!nh.getParam("joint_names",joint_names))
{
    ROS_ERROR("Could not get joint names.");
}
if (joint_names.size()!=7)
{
    ROS_ERROR_STREAM("Did not recieve 7 joints, Franka robot is 7 DOF
        robot!");
    return false;
}
```

Figure 53: franka_robot_jointposition_controller.cpp file, getting joint names.

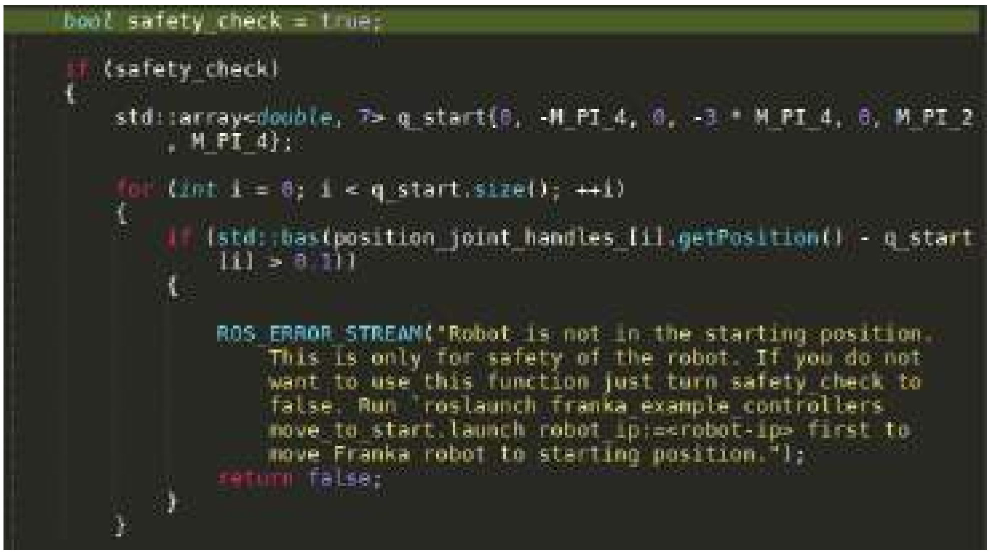
position_joint_handles Resize the position_joint_handles type vector and try to fill it from `hardware_interface`. If cannot get handles, return false (figure 54).



```
position_joint_handles_.resize(7);
for (int i = 0; i < 7; ++i)
{
    try{
        position_joint_handles_[i] = position_joint_interface_->
            getHandle(joint_names[i]);
    }
    catch(const hardware_interface::HardwareInterfaceException& e){
        ROS_ERROR_STREAM("Could not get joint handles: " << e.what());
        return false;
    }
}
```

Figure 54: franka_robot_jointposition_controller.cpp file, getting joint handles from position_joint_interface.

safety_check Check if Franka robot is in starting position. This function decreases the chance of robot making a dangerous or harmful movement. If you know how the robot will move at the beginning you can cancel it by making `safety_check = false` (figure 55).



```
bool safety_check = true;

if (safety_check)
{
    std::array<double, 7> q_start{0, -M_PI_4, 0, -3 * M_PI_4, 0, M_PI_2
        , M_PI_4};

    for (int i = 0; i < q_start.size(); ++i)
    {
        if (std::bas(position_joint_handles_[i].getPosition() - q_start
            [i] > 0.1))
        {

            ROS_ERROR_STREAM("Robot is not in the starting position.
                This is only for safety of the robot. If you do not
                want to use this function just turn safety_check to
                false. Run `roslaunch franka_example_controllers
                move_to_start.launch robot_ip:=<robot-ip> first to
                move Franka robot to starting position.");
            return false;
        }
    }
```

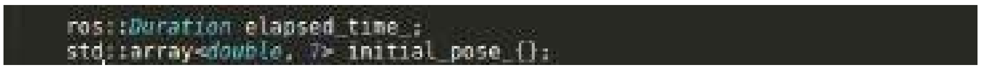
Figure 55: franka_robot_jointposition_controller.cpp file, safety check for starting position of Franka robot.

Initializing the joint position controller finished. Now, `starting()` and `update()` functions will be implemented.

19.2.2 JointPositionController::starting() function

`starting()` function is called right after `init()` function and it is called only once. It can be used for getting joints into the desired starting position or saving the joints current positions, etc.

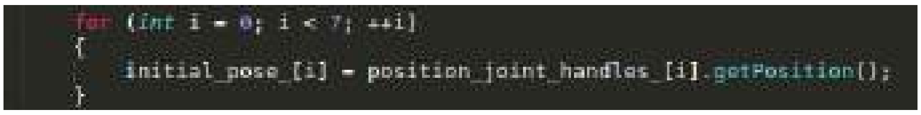
Add new variables `initial_pose array` and `ros::Duration elapsed_time` to the header file. `initial_pose array` will be used to store initial positions of the joints at the beginning and duration will be used for making a circler motion. Later we can use duration also for checking how long it takes between command and completion of the movement (figure 56).



```
ros::Duration elapsed_time_;
std::array<double, 7> initial_pose_{};
```

Figure 56: franka_robot_jointposition_controller.h file, add initial_pose array to the header file.

initial_pose array Fill in the `initial_pose` array with the current positions of the joints by `position_joint_handles_` `getPosition()` function (figure 57).



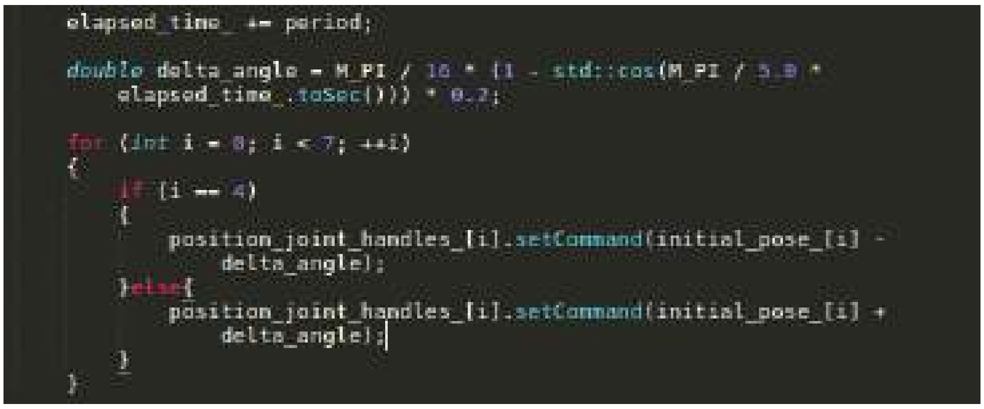
```
for (int i = 0; i < 7; ++i)
{
    initial_pose_[i] = position_joint_handles_[i].getPosition();
}
```

Figure 57: franka_robot_jointposition_controller.cpp file, fill `initial_pose` array with current positions of the joints.

19.2.3 JointPositionController::update() function

`update()` function is a loop that is running until the controller is shut down. Realtime controlling is happening here.

Send new positions to the joints. With `setCommand()` function of the `position_joint_handles_`, new joint positions are sent to the each joint. This formula is dependent on time so it should make Franka robot a elliptic motion. Fourth joint is limited between maximum -0.0698 rad to min -3.0178 rad (Hardware Setup for Vision-based Control using Franka Robot, Table 3 [1]) so it is checked and delta_angle is subtracted (figure 58).

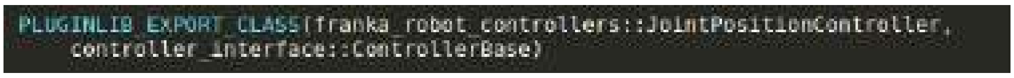


```
elapsed_time_ += period;

double delta_angle = M_PI / 16 * (1 - std::cos(M_PI / 5.0 *
    elapsed_time_.toSec())) * 0.2;

for (int i = 0; i < 7; ++i)
{
    if (i == 4)
    {
        position_joint_handles_[i].setCommand(initial_pose_[i] -
            delta_angle);
    }else{
        position_joint_handles_[i].setCommand(initial_pose_[i] +
            delta_angle);
    }
}
```

Figure 58: franka_robot_jointposition_controller.cpp file, send joints new position commands.

19.2.4 PLUGINLIB_EXPORT_CLASS.

A class must be marked as an "exported class" in order to be dynamically loaded. This is possible with the special macro PLUGINLIB_EXPORT_CLASS [15]. This macro added to the end of the .cpp file which the exported class defined. (figure 59).



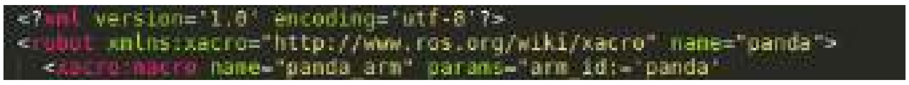
```
PLUGINLIB_EXPORT_CLASS(franka_robot_controllers::JointPositionController,
    controller_interface::ControllerBase)
```

Figure 59: franka_robot_jointposition_controller.cpp file, exporting PLUGIN to the .

Until now controller is defined but configuration of the robot has not been yet done.

19.3 Controller Configuration (yaml file)

In order to control Franka robot, `position_joint_handles_` must be filled with `position_joint_interface_->getHandle()` function which takes joint names as input (figures 53, 54). From Software Setup for Vision-based Control using Franka Robot report, Table 2 [6] it can be seen that for JointStateInterface, handle names are ranging from "<arm_id>_joint1" to "<arm_id>_joint7". `arm_id` is declared in panda_arm.xacro file (figure 60) under franka_description/robots folder as "panda". This brings us to the conclusion that joint names should be panda_joint1 - panda_joint7 (figure 61).

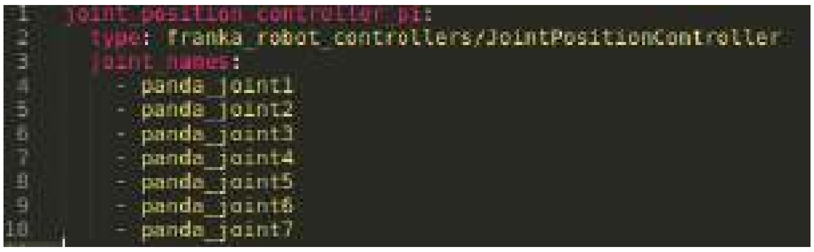


```
<?xml version='1.0' encoding='utf-8'?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="panda">
  <xacro:macro name="panda_arm" params="arm_id:='panda'
```

Figure 60: franka_description/robots/panda_arm.xacro file naming joints with macro, `params="arm_id:='panda`.

Create franka_example_controllers.yaml file. First create a new folder called config.

- `cd ~/catkin_ws/src/franka_ros/franka_robot_controller_pi`
- `mkdir config && cd config`
- `subl franka_controllers.yaml`



```
joint_position_controller_pi:
  type: franka_robot_controllers/JointPositionController
  joint_names:
    - panda_joint1
    - panda_joint2
    - panda_joint3
    - panda_joint4
    - panda_joint5
    - panda_joint6
    - panda_joint7
```

Figure 61: config/franka_controllers.yaml file defining the joint position controller.

franka_controllers.yaml is the general file for declaring controllers. For now there is only one definition. Later, `joint_velocity_controller_pi` will also be added here. First line is the name of the controller, give it a unique name otherwise it can be confusing when there are many controllers in the system. Second line is the type of the controller which is explained as `type="name_of_your_controller_package::NameOfYourControllerClass"` in franka_ros - writing your own controller tutorial [7].

Warning! This is not the same package with the package name mentioned in the package.xml or CMakeLists.txt. This is the **namespace** declared in header file which is "`franka_robot_controllers`" (figure 51).

NameOfYourControllerClass is relatively easy when previous information is known, which is "JointPositionController". Complete the configuration file with adding joint names.

19.4 plugin Description File

All the information about a plugin stored in an plugin_description.XML file (figure 62) that is in a machine readable format [15].

To be able to export `franka_robot_controllers::JointPositionController` plugin, it has to be defined. At the top of the package folder create franka_robot_controllers_plugin.xml file:

- `subl franka_robot_controllers_plugin.xml`

Start adding tags:

- `<library>` tag defines the library in which plugin classes live. For now, there is only JointPositionController class but later JointVelocityController class will be added. It is possible to add as many controllers as needed to the library [16].

  path is the location where the library will be created. Created .so file can be found under ~/catkin_ws/catkin_ws/devel/lib with the name given here.

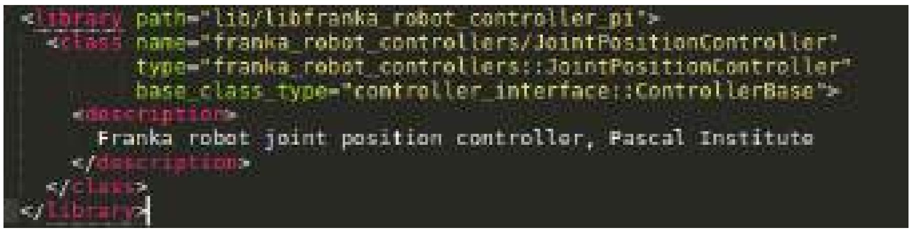
- `<class>` tag is the description of a class provided by a library.

  name is the lookup name of the class. An identifier for the plugin which is used by the pluginlib tools. It is optional in pluginlib 1.9 or higher (ROS Groovy or higher, Kinetic is higher version of Groovy.) But to make code clear, it is named same with the type attribute.

  type is the fully qualified class type. This is the namespace::class from .cpp file.

  base_class_type is the fully qualified type of the base class.

  description is the explanation of the plugin.



```
<library path="lib/libfranka_robot_controller_pi">
  <class name="franka_robot_controllers/JointPositionController"
         type="franka_robot_controllers::JointPositionController"
         base_class_type="controller_interface::ControllerBase">
    <description>
      Franka robot joint position controller, Pascal Institute
    </description>
  </class>
</library>
```

Figure 62: franka_robot_controllers_plugin.xml file defining the joint position controller plugin.

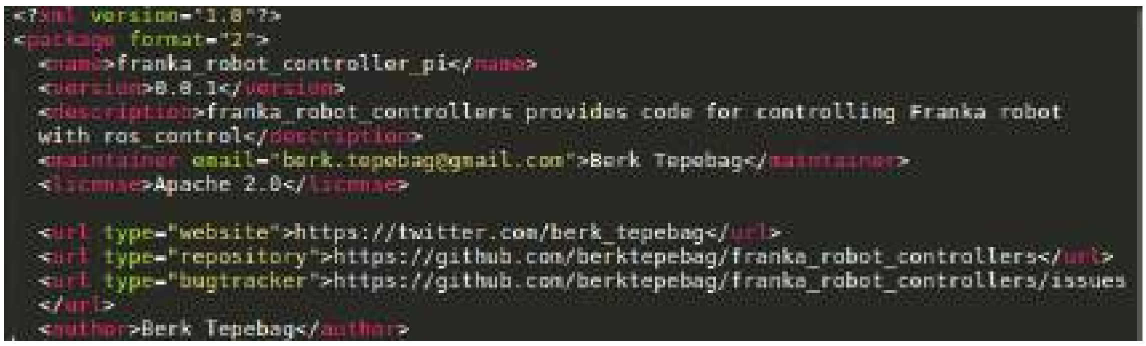
19.5 CMakeList.txt and Package.xml

19.5.1 Modifications in Package.xml

First make a copy of CMakeList.txt and Package.xml from /franka_ros/franka_example_controllers into the top of the package folder and make the necessary changes. You can either delete originally created files before pasting or replace them while pasting.

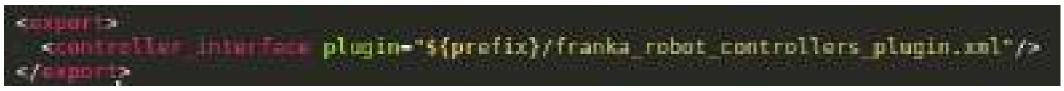
Warning! `<name>franka_robot_controller_pi</name>` must be the name of your package. Other informations will not cause a compiler error but name will (Third line on figure 63)! In this report dependencies will not be discussed since they should be added as libraries are being imported to the .h and .cpp files.

In order to use controller it has to be exported from Package.xml. Add the name of the plugin file into the plugin attribute (figure 64).



```
<?xml version="1.0"?>
<package format="2">
  <name>franka_robot_controller_pi</name>
  <version>0.0.1</version>
  <description>franka_robot_controllers provides code for controlling Franka robot
  with ros_control</description>
  <maintainer email="berk.tepebag@gmail.com">Berk Tepebag</maintainer>
  <license>Apache 2.0</license>

  <url type="website">https://twitter.com/berk_tepebag</url>
  <url type="repository">https://github.com/berktepebag/franka_robot_controllers</url>
  <url type="bugtracker">https://github.com/berktepebag/franka_robot_controllers/issues
  </url>
  <author>Berk Tepebag</author>
```

Figure 63: Package.xml file defining the informations and dependencies of the project.



```
<export>
  <controller_interface plugin="${prefix}/franka_robot_controllers_plugin.xml"/>
</export>
```

Figure 64: Package.xml, export the controller to be able to use it.

19.5.2 Modifications in CMakeList.txt

Warning! `project(franka_robot_controller_pi)` must be the name of your package. Also have to change rest of the "franka_example_controllers" with `${PROJECT_NAME}`.

For now we do not need messages, comment out message lines:

- `#add_message_files(FILES JointTorqueComparison.msg)`
- `#generate_messages()`
- `#generate_dynamic_reconfigure_options (# cfg/compliance_param.cfg # cfg/desired_mass_param.cfg)`

Change the name of the .cpp file with your .cpp file's name:

- `add_library($PROJECT_NAME src/franka_robot_jointposition_controller.cpp)`

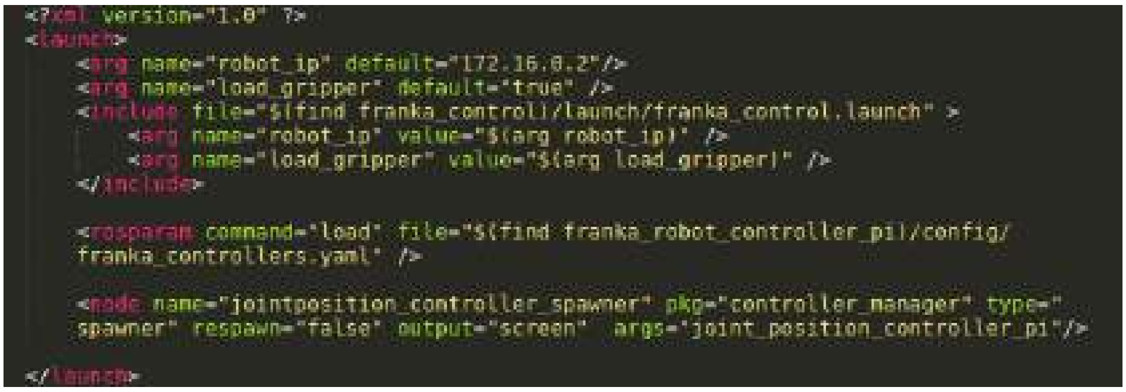
And finally comment out python packages which are not needed right now:

```
#catkin_install_python(
# PROGRAMS scripts/interactive_marker.py
# scripts/move_to_start.py
# DESTINATION $CATKIN_PACKAGE_BIN_DESTINATION)
```

19.6 joint_position_controller.launch file

Finally, create the launch file as it depicted at figure 65:

- `cd ~/catkin_ws/src/franka_ros/franka_robot_controller_pi`
- `mkdir launch && cd launch`
- `subl joint_position_controller.launch`



```
<?xml version="1.0" ?>
<launch>
    <arg name="robot_ip" default="172.16.0.2"/>
    <arg name="load_gripper" default="true" />
    <include file="$(find franka_control)/launch/franka_control.launch" >
        <arg name="robot_ip" value="$(arg robot_ip)" />
        <arg name="load_gripper" value="$(arg load_gripper)" />
    </include>

    <rosparam command="load" file="$(find franka_robot_controller_pi)/config/
    franka_controllers.yaml" />

    <node name="jointposition_controller_spawner" pkg="controller_manager" type="
    spawner" respawn="false" output="screen"  args="joint_position_controller_pi"/>

</launch>
```

Figure 65: joint_position_controller.launch file starts the controller.

- argument `robot_ip` is the IP of the control. If your IP is different than 172.16.0.2 you should change it. Or if you want to declare it every time while running launch file add argument `robot_ip:="your IP here"` at the end of launch file call.
- `load_gripper` shows gripper in RVIZ or/and Gazebo if true.
- franka_control.launch file is responsible from launching the Franka robot.
- `rosparam command="load"` will load parameters from franka_controrllers.yaml file.
- `jointposition_controller_spawner` is the control_manager spawner. Remember to be able to use controller a controller manager is needed.

19.7 Making the project

Last step is to make file:

- `cd ~/catkin_ws/`
- `catkin_make`
- `source devel/setup.bash`

And launch the controller, if you set `safety_check=true` do not forget to bring Franka robot to starting position.

- `roslaunch franka_robot_controller_pi joint_position_controller.launch`

Franka robot should start making elliptic movement.

20 Code

Code can be reached from:
`https://github.com/berktepebag/franka_robot_controllers/commits/joint_position_controller`

References

[1] ROS Wiki. (2019, May 20). ros_control Package. Retrieved from
`http://wiki.ros.org/ros_control`

[2] Allen, Z. (2019, May 20). How to implement ros_control on a custom robot. Retrieved from
`https://slaterobots.com/blog/5abd8a1ed4442a651de5cb5b/how-to-implement-ros_control-on-a-custom-robot`

[3] Franka, Emika. (2019, May 20). franka_ros, franka_hw. Retrieved from
`https://frankaemika.github.io/docs/franka_ros.html#franka-hw`

[4] Tepebag, B. (2019, May 20). Software Setup for Vision-based Control using Franka Robot

[5] Chitta, Sachin and Marder-Eppstein, Eitan and Meeussen, Wim and Pradeep, Vijay and Rodríguez Tsouroukdissian, Adolfo and Bohren, Jonathan and Coleman, David and Magyar, Bence and Raiola, Gennaro and Lüdtke, Mathias and Fernández Perdomo, Enrique (2017). ros_control: A generic and simple control framework for ROS, The Journal of Open Source Software. Retrieved from
`http://www.theoj.org/joss-papers/joss.00456/10.21105.joss.00456.pdf`

[6] ROS Wiki. (2019, May 20). controller_manager package. Retrieved from
`http://wiki.ros.org/controller_manager?distro=melodic`

[7] Franka, Emika. (2019, April 12). franka_ros, Writing your own controller. Retrieved from
`https://frankaemika.github.io/docs/franka_ros.html#writing-your-own-controller`

[8] ROS Wiki. (2019, May 21). ros_control Tutorials. Retrieved from
`http://wiki.ros.org/ros_control/Tutorials`

[9] ROS Docs. (2019, May 21). ROS joint command interface. Retrieved from
`http://docs.ros.org/jade/api/hardware_interface/html/c++/classhardware__interface_1_1JointCommandInterface.html`

[10] ROS Docs. (2019, May 21). ROS hardware interface. Retrieved from
`http://docs.ros.org/jade/api/hardware_interface/html/c++/classhardware__interface_1_1RobotHW.html`

[11] Wikipedia. (2019, May 27). Resource acquisition is initialization. Retrieved from
`https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization`

[12] Stroustrup, Bjarne (2019, May 27). "Why doesn't C++ provide a "finally" construct?". Retrieved from
`http://www.stroustrup.com/bs_faq2.html#finally`

[13] ROS Docs. (2019, May 21). ROS node handle. Retrieved from
`http://docs.ros.org/melodic/api/roscpp/html/classros_1_1NodeHandle.html#details` abbreviation

[14] Tepebag, B. (2019, May 20). Hardware Setup for Vision-based Control using Franka Robot

[15] ROS Wiki. (2019, May 23). ROS pluginlib package website. Retrieved from
`http://wiki.ros.org/pluginlib`

[16] ROS Wiki. (2019, May 23). ROS pluginlib PluginDescriptionFile website. Retrieved from
`http://wiki.ros.org/pluginlib/PluginDescriptionFile`

21 Teleop Command of Joint Position Controller for Franka Robot - Introduction

Last report a new joint position controller has been created but it was moving in a ellipsoid trajectory. Now, we will use ROS's communication tools which are topics, actions and services in order to control the joint positions from another node.

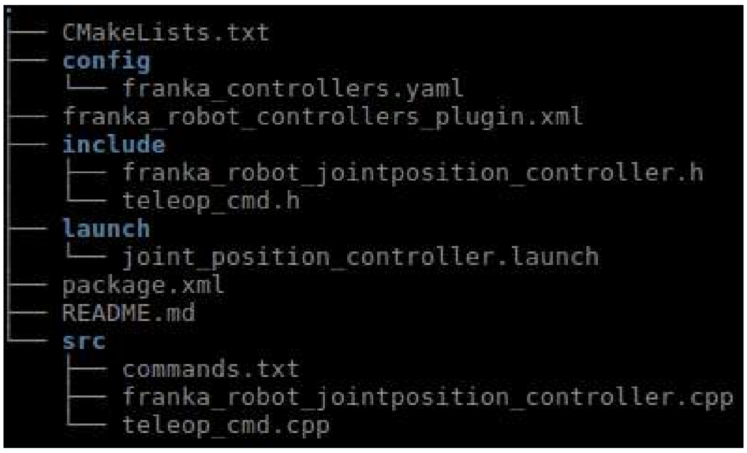
The rest of the report is organized as follows: Section 22 introduces ROS topics and section 23 shows how to write teleop command for Franka robot.

22 ROS Topics and Services

ROS nodes exchanges messages over topics. One or more node can publish information to a topic and other node or nodes can subscribe to this topic. Even sometimes no node subscribes to the topic but publisher still keeps publishing. Nodes do not have to know each other since they are interested about "what" rather than "who". Topics are intended for one-way data stream. If a response is needed from subscriber node, services should be preferred [1]. We will use topics instead of services since we do not need feedback from controller for now. And it will be possible to publish to that topic from the new nodes that we will create in the feature so we do not have to make modifications to the controller.

23 Teleop Franka robot with topics

Now we will start creating necessary files to tele-operate the Franka robot. At the end of this report our folder tree will look like figure 102.



```
.
├── CMakeLists.txt
├── config
│   └── franka_controllers.yaml
├── franka_robot_controllers_plugin.xml
├── include
│   ├── franka_robot_jointposition_controller.h
│   └── teleop_cmd.h
├── launch
│   └── joint_position_controller.launch
├── package.xml
├── README.md
└── src
    ├── commands.txt
    ├── franka_robot_jointposition_controller.cpp
    └── teleop_cmd.cpp
```

Figure 66: Folder tree after adding teleop_cmd.h and .cpp files.

23.1 teleop_cmd.h

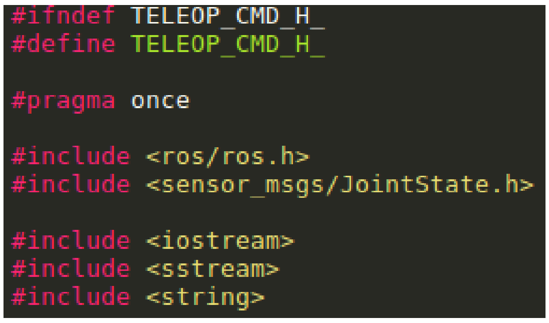
Here, a new node called franka_robot_teleop will be created. This node will take joint goal positions and joint speeds as inputs. To make input calculations easier user will give joint goal positions in degrees. Type of the message is JointState message, which takes name, position, velocity and effort as variables [2]. Joint name, goal positions and joint speeds will be published to the "franka_robot/teleop_cmd" topic. To be able to do this a new .cpp and .h file has to be created. First create teleop_cmd.h file under the /include folder (figure 102).

In teleop_cmd.h file start by getting necessary libraries that depicted in figure 67.

<ros/ros.h> In order to use publishers, subscribers, messages, actions, services etc. this library must be imported/included.

<sensor_msgs/JointState.h> In order to send JointState messages to a topic JointState from sensor_msgs has to be included [2].

<iostream> is the header file standard input/output stream objects. It's functions will be used to get keyboard inputs from terminal [3].

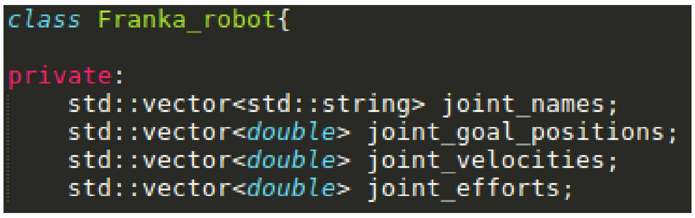


```
#ifndef TELEOP_CMD_H_
#define TELEOP_CMD_H_

#pragma once

#include <ros/ros.h>
#include <sensor_msgs/JointState.h>

#include <iostream>
#include <sstream>
#include <string>
```

Figure 67: teleop_cmd.h: Necessary libraries for teleop commanding Franka robot.

<sstream> is the header file providing string stream classes [4].

<string> is the header file for using different string types [5]



```
class Franka_robot{

private:
    std::vector<std::string> joint_names;
    std::vector<double> joint_goal_positions;
    std::vector<double> joint_velocities;
    std::vector<double> joint_efforts;
```

Figure 68: Private variables used to keep Franka robot's joints' informations.

With sensor_msg/JointState messages,

1. name
2. position
3. velocity
4. effort

vectors can be sent as commands. Joint names must be unique. Position, velocity and effort vectors are optional. They either have to be same size with the name vector or must be empty. For joint position controller, goal positions and velocities will be sent. Effort is not needed since Franka robot is calculating necessary torques within FCI when PositionJointInterface is used according to the given position command.

Private variables and functions. In order to send them as commands we should first create one `string` and 3 `double vectors` to keep the joint names and command values. (figure 68).

Public variables and functions. Public variables and functions will be explained here as they are depicted in figure 69.
Since Franka robot has 7 joints, we can declare it as a constant number:
`int const FRANKA_ROBOT_JOINT_NUMBER = 7;`

`void setJointNames()` sets joints' names as described in Software Setup for Vision-based Control using Franka Robot report [6], Table 2: List of the resource handle names of the controller interfaces.

`void setJointName(int joint_id,std::string jointName)` can be used to change name of a joint if needed.

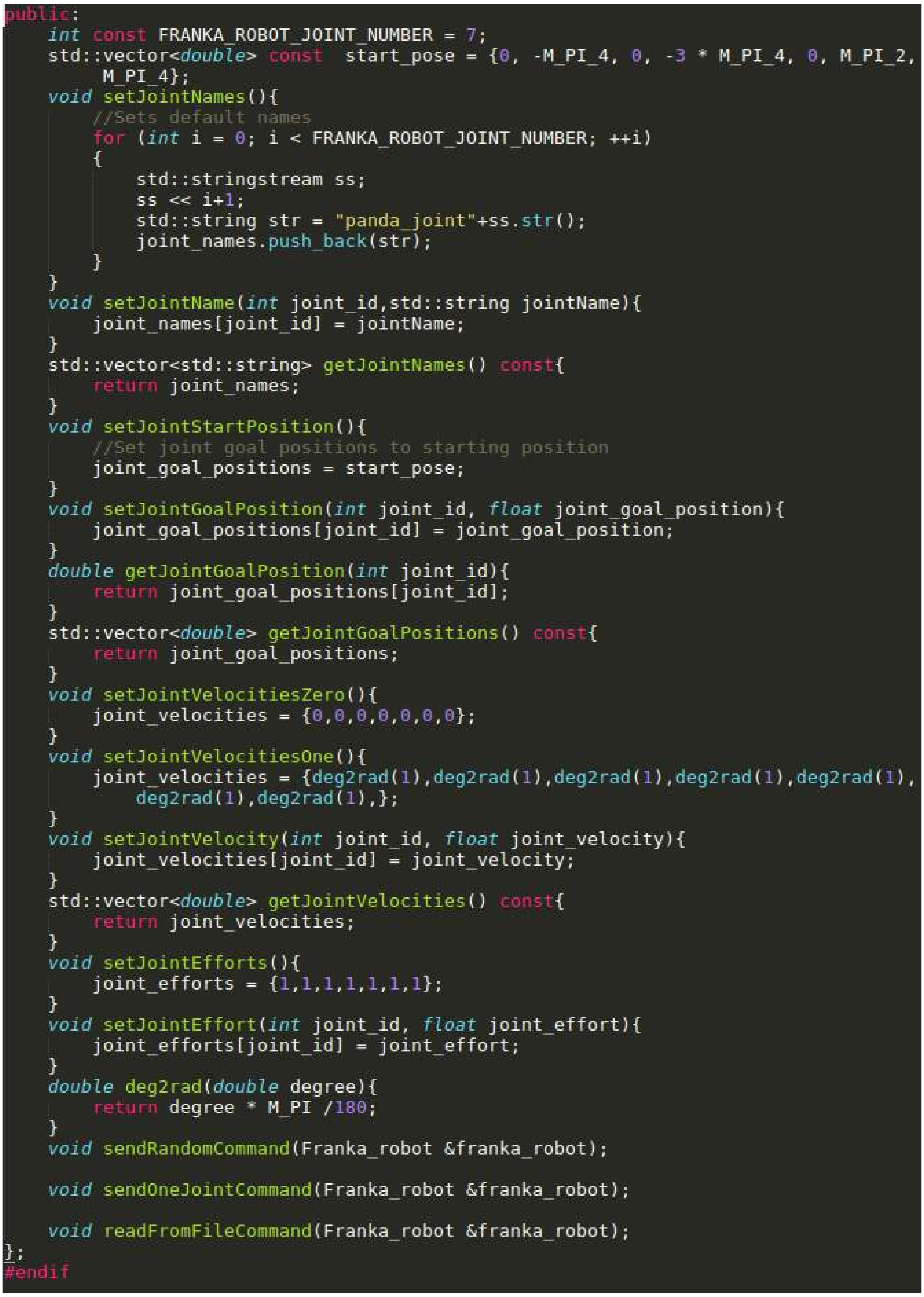
`std::vector<std::string> getJointNames() const` returns a string vector of the names of the joints.

`void setJointStartPosition()` When user wants to move joints to starting position, this function can be called. It will set joint_goal_positions to start pose.

`void setJointGoalPosition(int joint_id, float joint_goal_position)` sets a joint's goal position; takes in joint id and goal position as inputs.

`double getJointGoalPosition` returns a joint's goal position as a double value.

`std::vector<double> getJointGoalPositions()` returns all joints goal positions as a double vector.



```
public:
    int const FRANKA_ROBOT_JOINT_NUMBER = 7;
    std::vector<double> const  start_pose = {0, -M_PI_4, 0, -3 * M_PI_4, 0, M_PI_2,
     M_PI_4};
    void setJointNames(){
        //Sets default names
        for (int i = 0; i < FRANKA_ROBOT_JOINT_NUMBER; ++i)
        {
            std::stringstream ss;
            ss << i+1;
            std::string str = "panda_joint"+ss.str();
            joint_names.push_back(str);
        }
    }
    void setJointName(int joint_id,std::string jointName){
        joint_names[joint_id] = jointName;
    }
    std::vector<std::string> getJointNames() const{
        return joint_names;
    }
    void setJointStartPosition(){
        //Set joint goal positions to starting position
        joint_goal_positions = start_pose;
    }
    void setJointGoalPosition(int joint_id, float joint_goal_position){
        joint_goal_positions[joint_id] = joint_goal_position;
    }
    double getJointGoalPosition(int joint_id){
        return joint_goal_positions[joint_id];
    }
    std::vector<double> getJointGoalPositions() const{
        return joint_goal_positions;
    }
    void setJointVelocitiesZero(){
        joint_velocities = {0,0,0,0,0,0,0};
    }
    void setJointVelocitiesOne(){
        joint_velocities = {deg2rad(1),deg2rad(1),deg2rad(1),deg2rad(1),deg2rad(1),
            deg2rad(1),deg2rad(1),};
    }
    void setJointVelocity(int joint_id, float joint_velocity){
        joint_velocities[joint_id] = joint_velocity;
    }
    std::vector<double> getJointVelocities() const{
        return joint_velocities;
    }
    void setJointEfforts(){
        joint_efforts = {1,1,1,1,1,1,1};
    }
    void setJointEffort(int joint_id, float joint_effort){
        joint_efforts[joint_id] = joint_effort;
    }
    double deg2rad(double degree){
        return degree * M_PI /180;
    }
    void sendRandomCommand(Franka_robot &franka_robot);

    void sendOneJointCommand(Franka_robot &franka_robot);

    void readFromFileCommand(Franka_robot &franka_robot);
};
#endif
```

Figure 69: Public variables and functions.

`void setJointVelocitiesZero()` sets all joint velocities to zero.

`void setJointVelocitiesOne()` sets all joint velocities to one deg/s in order to start moving joints. But when sending a new command, velocity can be changed by the user.

`void setJointVelocity(int joint_id, float joint_velocity)` sets one joint's target velocity; takes in joint id and goal velocity as inputs.

`std::vector<double> getJointVelocities()` returns a double vector of all the joints' target velocities.

`void setJointEfforts()` sets joints' efforts to 1 N. But since only joint position controller is used this value is not necessary. Torques are calculated by the Franka robot itself.

`void setJointEffort(int joint_id, float joint_effort)` sets a joint's effort limit.

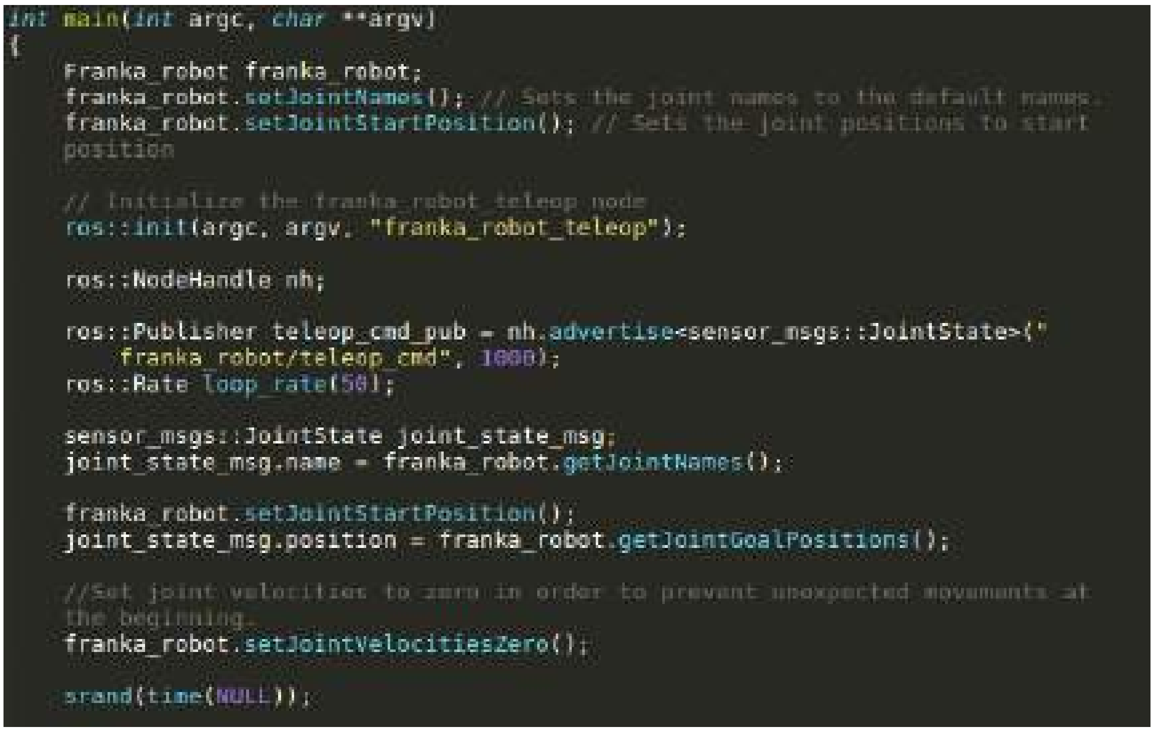
`double deg2rad(double degree)` converts degree to radians; takes in double degrees as input.

`sendRandomCommand(Franka_robot &franka_robot)` takes franka robot as a reference and sends random position and velocities in the range chosen by the user.

`setOneJointCommand(Franka_robot &franka_robot)` takes franka robot as a reference and sends chosen joint, desired position and velocity.

`readFromFileCommand(Franka_robot &franka_robot)` takes franka robot as a reference and sends command read from a text file.

23.2 teleop_cmd.cpp



```
int main(int argc, char **argv)
{
    Franka_robot franka_robot;
    franka_robot.setJointNames(); // Sets the joint names to the default names.
    franka_robot.setJointStartPosition(); // Sets the joint positions to start
    position

    // Initialize the franka_robot_teleop node
    ros::init(argc, argv, "franka_robot_teleop");

    ros::NodeHandle nh;

    ros::Publisher teleop_cmd_pub = nh.advertise<sensor_msgs::JointState>("
        franka_robot/teleop_cmd", 1000);
    ros::Rate loop_rate(50);

    sensor_msgs::JointState joint_state_msg;
    joint_state_msg.name = franka_robot.getJointNames();

    franka_robot.setJointStartPosition();
    joint_state_msg.position = franka_robot.getJointGoalPositions();

    //Set joint velocities to zero in order to prevent unexpected movements at
    the beginning.
    franka_robot.setJointVelocitiesZero();

    srand(time(NULL));
```

Figure 70: Public variables and functions.

teleop_cmd.cpp starts with including header file that recently created. Functions has been created there now will be initialized and used (figure 70).

`Franka_robot franka_robot;` Instantiates a `Franka_robot` class, so the functions can be called.

`franka_robot.setJointNames();` Sets joints' names to the default names.

`franka_robot.setJointStartPosition();` Sets joints' positions to start position

`ros::init(argc, argv, "franka_robot_teleop");` In order to start publishing topics a node initialization is needed. ROS needs a unique name for each node. Give the node an unique name.

`ros::NodeHandle nh;` Initializes the node. As the name implies, handle's the node's functions and let user reach them.

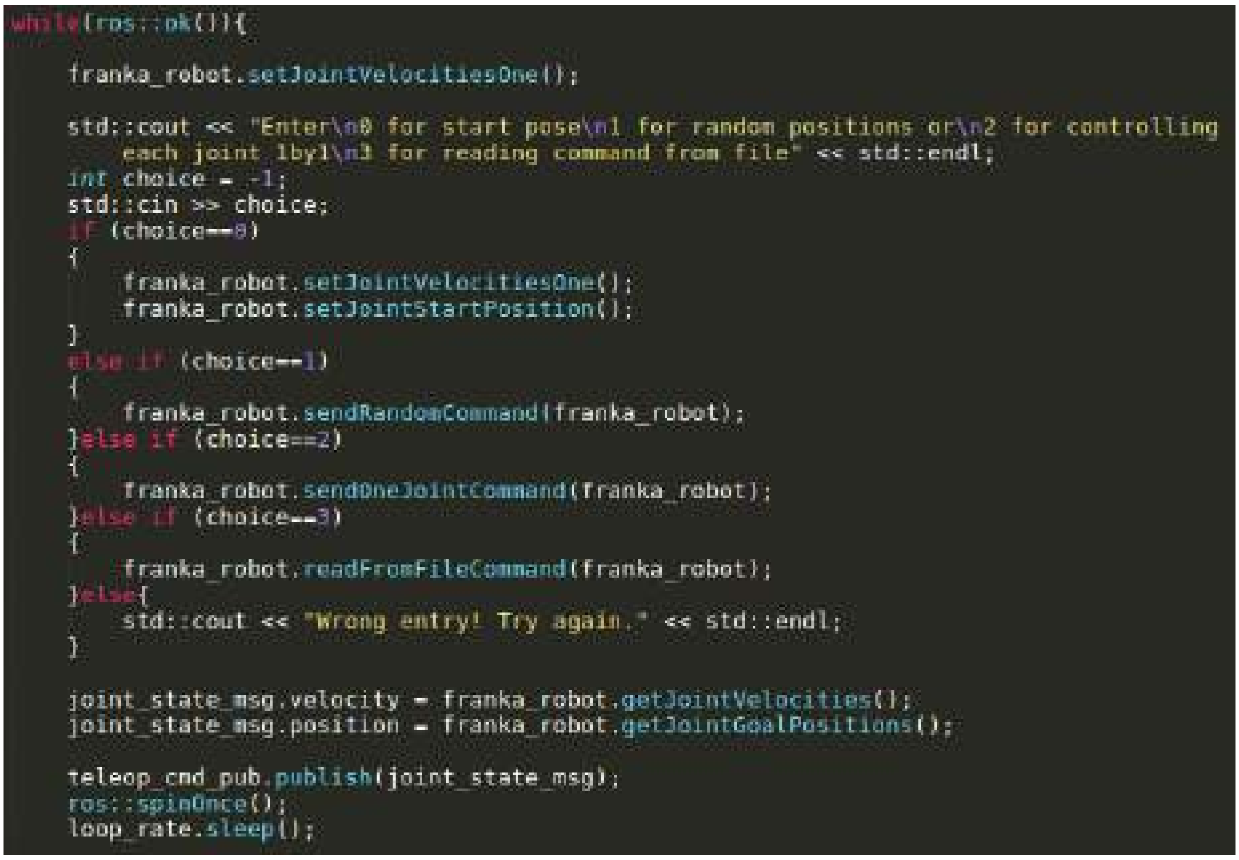
`ros::Publisher teleop_cmd_pub = nh.advertise <sensor_msgs::JointState> ("franka_robot/teleop_cmd", 1000);`
The commands will be published to the topic `franka_robot/teleop_cmd` and if commands are published too quickly 1000 messages will be buffered before being destroyed. Since FCI is communicating at 1 kHz, 1000 messages should be a good number to buffer.

`ros::Rate loop_rate(50);` Frequency that messages are published to the topics.

`sensor_msgs::JointState joint_state_msg;` Define a `JointState` message. `joint_state_msg.name = franka_robot.getJointNames();` Set the message's name variables.

`franka_robot.setJointVelocitiesZero();` Set joint velocities to zero in order to prevent unexpected movements at the beginning.

`srand(time(NULL));` Creates random numbers using current time. It will be used to create random positions and velocities.



```
while(ros::ok()){

    franka_robot.setJointVelocitiesOne();

    std::cout << "Enter\n0 for start pose\n1 for random positions or\n2 for controlling
        each joint 1by1\n3 for reading command from file" << std::endl;
    int choice = -1;
    std::cin >> choice;
    if (choice==0)
    {
        franka_robot.setJointVelocitiesOne();
        franka_robot.setJointStartPosition();
    }
    else if (choice==1)
    {
        franka_robot.sendRandomCommand(franka_robot);
    }else if (choice==2)
    {
        franka_robot.sendOneJointCommand(franka_robot);
    }else if (choice==3)
    {
        franka_robot.readFromFileCommand(franka_robot);
    }else{
        std::cout << "Wrong entry! Try again." << std::endl;
    }

    joint_state_msg.velocity = franka_robot.getJointVelocities();
    joint_state_msg.position = franka_robot.getJointGoalPositions();

    teleop_cmd_pub.publish(joint_state_msg);
    ros::spinOnce();
    loop_rate.sleep();
```

Figure 71: `ros::ok()` loop where our node is running until it is terminated.

`while(ros::ok())` checks if ros is running or not, if conditions mentioned below is not satisfied code will loop inside (figure 71). This node will keep running unless; node stopped with `CTRL+C`, another node with same name has been started, `ros::shutdown()` has been called or all `ros::NodeHandles` have been destroyed [7].

`franka_robot.setJointVelocitiesOne()` Before getting into the loop, joint velocities were set to 0 in order to prevent unexpected movements. Now we are setting 1 to start moving. If only position is given after this line default joint speed will be 1 deg/s.

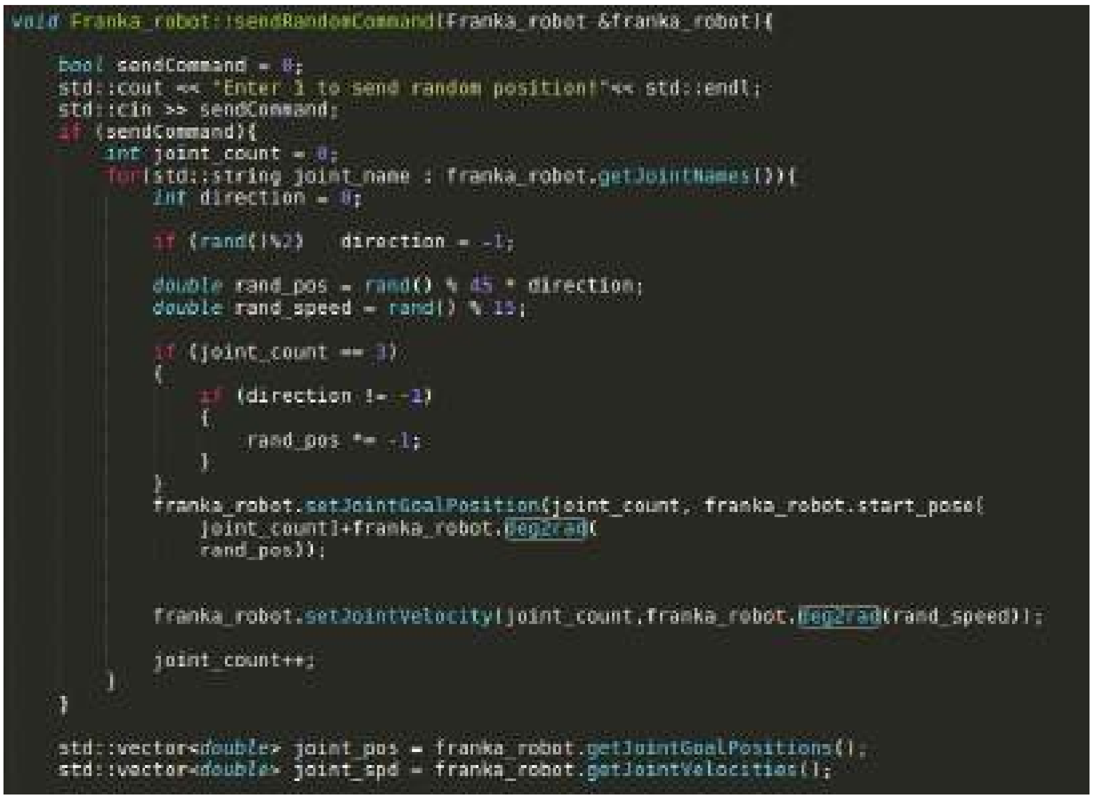
Now user has 4 choices:

1. choice 0: Sets joints' positions to start positions.
2. choice 1: Set random joint position and speeds. This is useful for testing the robustness of the controller. But be aware it is not checking soft limits of the joints. If the given random command is exceeding the soft limits FCI will stop the robot and throw an error. Then you have to restart either from command line or from Desk!.
3. choice 2: User sets each joint's position and speed one by one.
4. choice 3: User can update command.txt under src/ folder. This option reads position and speed variables from txt file. Instead of txt a .csv file or database file can be implemented as needed.

`sendRandomCommand` User can change the range joint position and joint velocities by changing values after % sign (figure 72):

`double rand_pos = rand() % 30 * direction;` and
`double rand_speed = rand() % 15;`

Direction will change by fifty percent chance to create random positions which cover most of the work space. `if (rand()%2) direction = -1;`

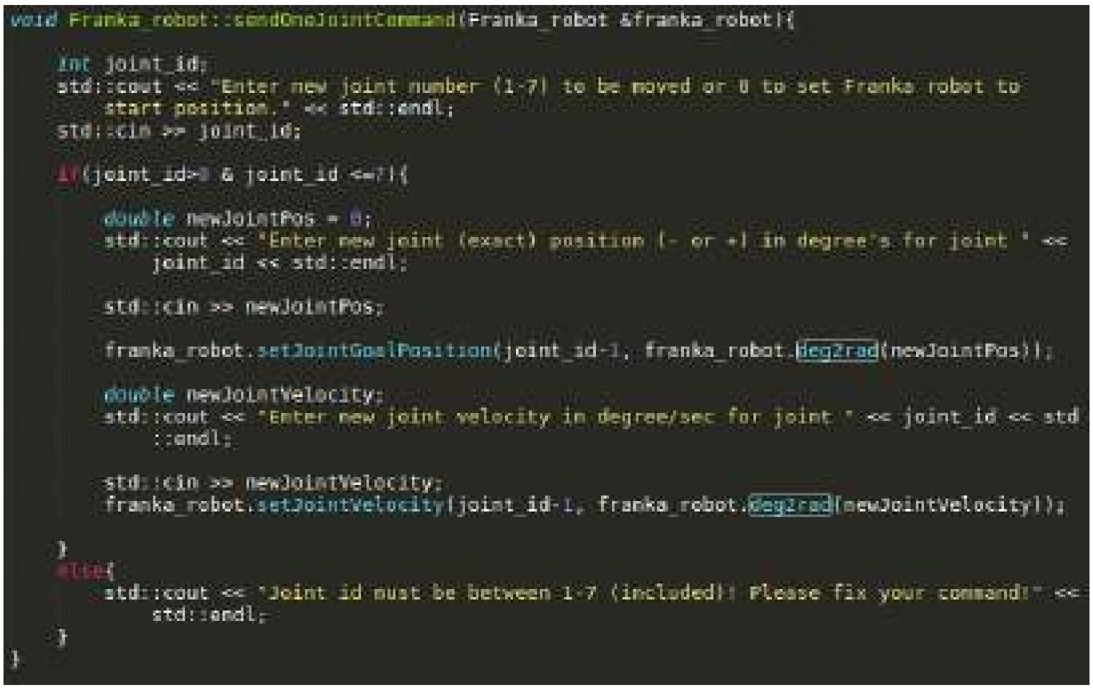


```
void Franka_robot::sendRandomCommand(Franka_robot &franka_robot){

    bool sendCommand = 0;
    std::cout << "Enter 1 to send random position!"<< std::endl;
    std::cin >> sendCommand;
    if (sendCommand){
        int joint_count = 0;
        for(std::string joint_name : franka_robot.getJointNames()){
            int direction = 0;

            if (rand()%2)   direction = -1;

            double rand_pos = rand() % 45 * direction;
            double rand_speed = rand() % 15;

            if (joint_count == 3)
            {
                if (direction != -1)
                {
                    rand_pos *= -1;
                }
            }
            franka_robot.setJointGoalPosition(joint_count, franka_robot.start_pose[
                joint_count]+franka_robot.deg2rad(
                rand_pos));


            franka_robot.setJointVelocity(joint_count,franka_robot.deg2rad(rand_speed));

            joint_count++;
        }
    }

    std::vector<double> joint_pos = franka_robot.getJointGoalPositions();
    std::vector<double> joint_spd = franka_robot.getJointVelocities();
```

Figure 72: sendRandomCommand function, sends random joint positions and speeds.

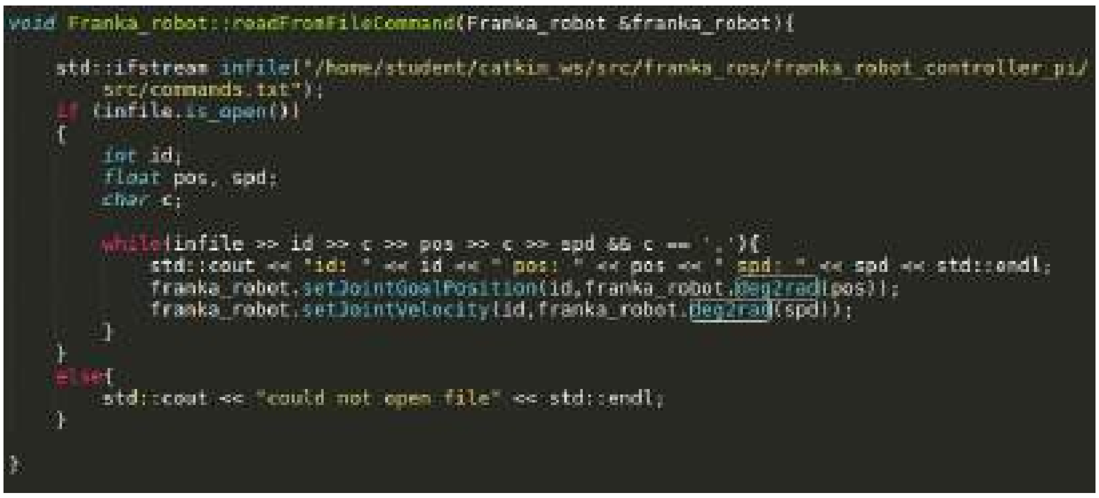
`sendOneJointCommand` User sends joint position and velocity command to the joint he/she choses. Good for checking joints' robustness after making changes in the controller (figure 73).

`readFromFileCommand` User can send joint commands via a text file. This function can be improved by making reading commands from .csv file or a database (image 74). **Note!** Do not forget to change directory of the text file or it will throw "could not open file" error.



```cpp
void Franka_robot::sendOneJointCommand(Franka_robot &franka_robot){

    int joint_id;
    std::cout << "Enter new joint number (1-7) to be moved or 0 to set Franka robot to
        start position." << std::endl;
    std::cin >> joint_id;

    if(joint_id>0 & joint_id <=7){

        double newJointPos = 0;
        std::cout << "Enter new joint (exact) position (- or +) in degree's for joint " <<
            joint_id << std::endl;

        std::cin >> newJointPos;

        franka_robot.setJointGoalPosition(joint_id-1, franka_robot.deg2rad(newJointPos));

        double newJointVelocity;
        std::cout << "Enter new joint velocity in degree/sec for joint " << joint_id << std
            ::endl;

        std::cin >> newJointVelocity;
        franka_robot.setJointVelocity(joint_id-1, franka_robot.deg2rad(newJointVelocity));

    }
    else{
        std::cout << "Joint id must be between 1-7 (included)! Please fix your command!" <<
            std::endl;
    }
}
```

Figure 73: sendOneJointCommand function, let’s user send position and speed command for each joint.



```cpp
void Franka_robot::readFromFileCommand(Franka_robot &franka_robot){

    std::ifstream infile("/home/student/catkin_ws/src/franka_ros/franka_robot_controller_pi/
        src/commands.txt");
    if (infile.is_open())
    {
        int id;
        float pos, spd;
        char c;

        while(infile >> id >> c >> pos >> c >> spd && c == ';'){
            std::cout << "id: " << id << " pos: " << pos << " spd: " << spd << std::endl;
            franka_robot.setJointGoalPosition(id,franka_robot.deg2rad(pos));
            franka_robot.setJointVelocity(id,franka_robot.deg2rad(spd));
        }
    }
    else{
        std::cout << "could not open file" << std::endl;
    }

}
```

Figure 74: readFromFileCommand function, reads position and speed commands from command.txt.

References

[1] ROS wiki. (2019, July 8). Topics. Retrieved from
`http://wiki.ros.org/Topics`

[2] ROS docs. (2019, June 11). sensor_msgs, JointState Message definition. Retrieved from
`http://docs.ros.org/melodic/api/sensor_msgs/html/msg/JointState.html`

[3] cplusplus.com. (2019, June 11). <iostream>. Retrieved from
`http://www.cplusplus.com/reference/iostream/`

[4] cplusplus.com. (2019, June 11). <sstream>. Retrieved from
`http://www.cplusplus.com/reference/sstream/?kw=sstream`

[5] cplusplus.com. (2019, June 11). <string>. Retrieved from
`http://www.cplusplus.com/reference/string/?kw=%3Cstring%3E`

[6] Tepebag, B. (2019, May 20). Software Setup for Vision-based Control using Franka Robot

[7] ROS docs. (2019, June 11). Writing a Simple Publisher and Subscriber (C++) Retrieved from
`http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29`

24 Joint Position Control with Trapezoid Velocity Model for Franka Robot - Introduction

In the last 2 reports, single joint position control with constant velocity model's theory and it's implementation have been discussed and joint position controller example has been re-written as a tutorial. In this report, a trapezoid velocity model is proposed.

The rest of the report is organized as follows: Section 25 discusses the reasons why trapezoid velocity model is necessary and the theory of it. Section 26 shows how to implement the trapezoid velocity model. Section 27 shows the experiments of the model. Section 34 makes a conclusion for the novel model according to the theory and experiments held. Finally, section 42 shows how the trapezoid velocity controller works on Franka robot with a video.

25 Trapezoid velocity model

25.1 Problems of single joint position control with constant velocity model

As it has been discussed in Single Joint Position Control with Constant velocity Model for using Franka Robot report, a joint movement consists of 3 phases; constant acceleration, constant velocity and constant deceleration (figure 75). Size and the slope of the acceleration and deceleration triangles in the velocity/time graph depend on the value of the acceleration. When the area of the triangle of the acceleration and deceleration are getting smaller, average velocity of the joint ($v_{average}$) gets closer to the velocity desired by the user ($v_{desired}$). But a problem occurs when acceleration and deceleration is not controlled. Since the last model that has been proposed does not consist a control for the acceleration and deceleration, this causes two oscillations; first, when joint velocity (v_n) reaches to the desired velocity and second, when joint reaches to the goal position which can be seen on the figure 76. As explained in "Single Joint Position Control with Constant Velocity Model for Franka Robot" report [2], commands are sent to the Control via FCI at each period. First oscillation happens because of the acceleration and inertia causes joint to exceed the n^{th} period's goal position. Since $n+1^{th}$'s period's joint position command is now behind the joint's current position, the actuator is trying to turn to the inverse direction which is not possible at relatively high velocities. If $v_{desired}$ is higher than 10 deg/s or 0.17 rad/s, this phenomenon becomes visible at the graphics. Second oscillation is happening because of the same reason as the first one; n^{th} period's command exceeds the final goal position and this results joint trying to converge it's position at the goal position by sending new commands. A trapezoid velocity model is proposed in order to control the acceleration and deceleration to prevent oscillations.

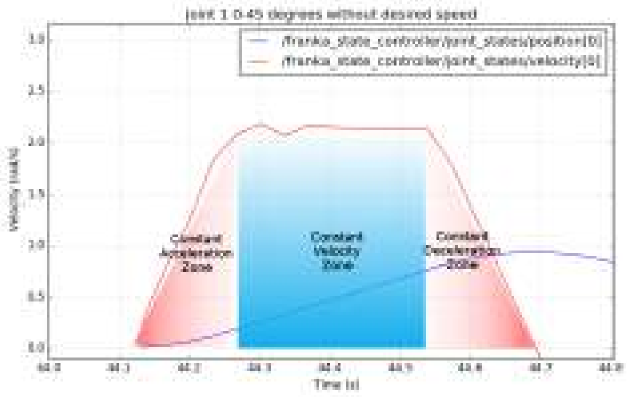


Figure 75: Constant acceleration, velocity and deceleration zones.

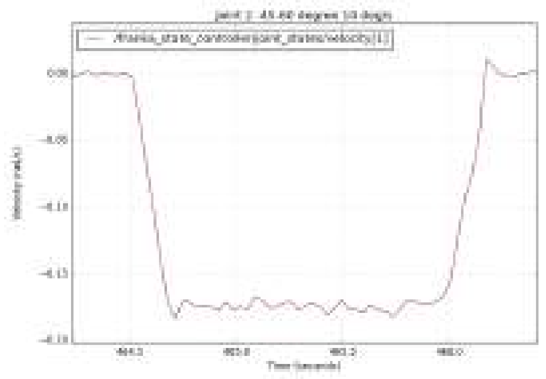


Figure 76: Joint 2, -45 to -60 degrees with 10 deg/s velocity. Oscillation is visible on the right side of the chart.

25.2 Objective, given and knowns of the trapezoid velocity model

Objective of the trapezoid velocity model. In this new model, acceleration and deceleration will also be controlled by our novel controller to prevent oscillations. In order to do this, acceleration per period will be calculated and added to or subtracted from the joint velocity (v_n) until it reaches to the desired velocity ($v_{desired}$).

Given values. Given values are the same with the previous model. Joints' goal positions and goal velocities will be given to the controller.

Known values. In addition to the desired velocities of the joints, acceleration limits has to be known in this model. Acceleration limits can be reached from "Hardware Setup" report, Table 3: Joints' position, velocity, acceleration, jerk, torque, rotatum limits of Franka robot [1]. Joints' positions are known as they are being read from the sensors in order to calculate each joint's distance to the goal position.

Calculated values. To control the deceleration, distance for each joint to start decelerating with the current velocity has to be known. For that purpose, time needed to stop is calculated and then distance needed to stop is calculated according to the time needed to stop. Then current joint position is checked to find if it passed the deceleration point and if this is the case, the current joint velocity is decreased by the rate of deceleration at each period.

25.3 Theory of the joint position control with trapezoid velocity model

Under the ideal communication conditions, 1000 user commands per second or 1 message per millisecond are sent from FCI to the Control. But communication is not always at 1 kHz. To prevent confusion under non-ideal states, instead of using numbers, a variable -period- will be used in further explanation of the concept.

$$T = 1/f \tag{10}$$

where period, T, is the time spent for sending one message in seconds and frequency, f, is the number of messages per second (communication rate). Total seconds passed in n periods duration can be calculated as:

$$t_n = \Sigma_{k=1}^{n} T_k \tag{11}$$

where t_n is the total seconds passed while sending n commands, n is the number of periods passed, and T_k is the period of the k^{th} command. During acceleration, for any given period n, velocity of the joint can be calculated as:

$$v_n = a_{constant} \cdot t_n = \Sigma_{k=1}^{n} a_{constant} \cdot T_k \tag{12}$$

where v_n is the velocity of the joint at n^{th} period in rad/s, $a_{constant}$ is the constant acceleration in rad/s^2 and t_n is the total seconds passed during acceleration which is equal to the sum of the periods. If joint reached to the $v_{desired}$, and if it has not reached to the deceleration point yet, $a_{constant}$ becomes zero. Joint velocity is set equal to the desired velocity:

$$v_n = v_{desired} \tag{13}$$

where v_n, velocity of the joint of the n^{th} period and $v_{desired}$ is desired velocity of the joint given by the user. In order to stop joint at given goal position, it has to be known how long it will take to stop with current velocity and deceleration limit:

$$t_{deceleration} = v_n/a_{constant} \tag{14}$$

where $t_{deceleration}$ is time needed in seconds for joint to stop with current joint velocity, v_n, and joint's deceleration limit $a_{constant}$. When total time needed to stop is known distance needed to stop can be calculated:

$$d_{deceleration} = \frac{1}{2} \cdot a_{constant} \cdot t_{deceleration}^2 \tag{15}$$

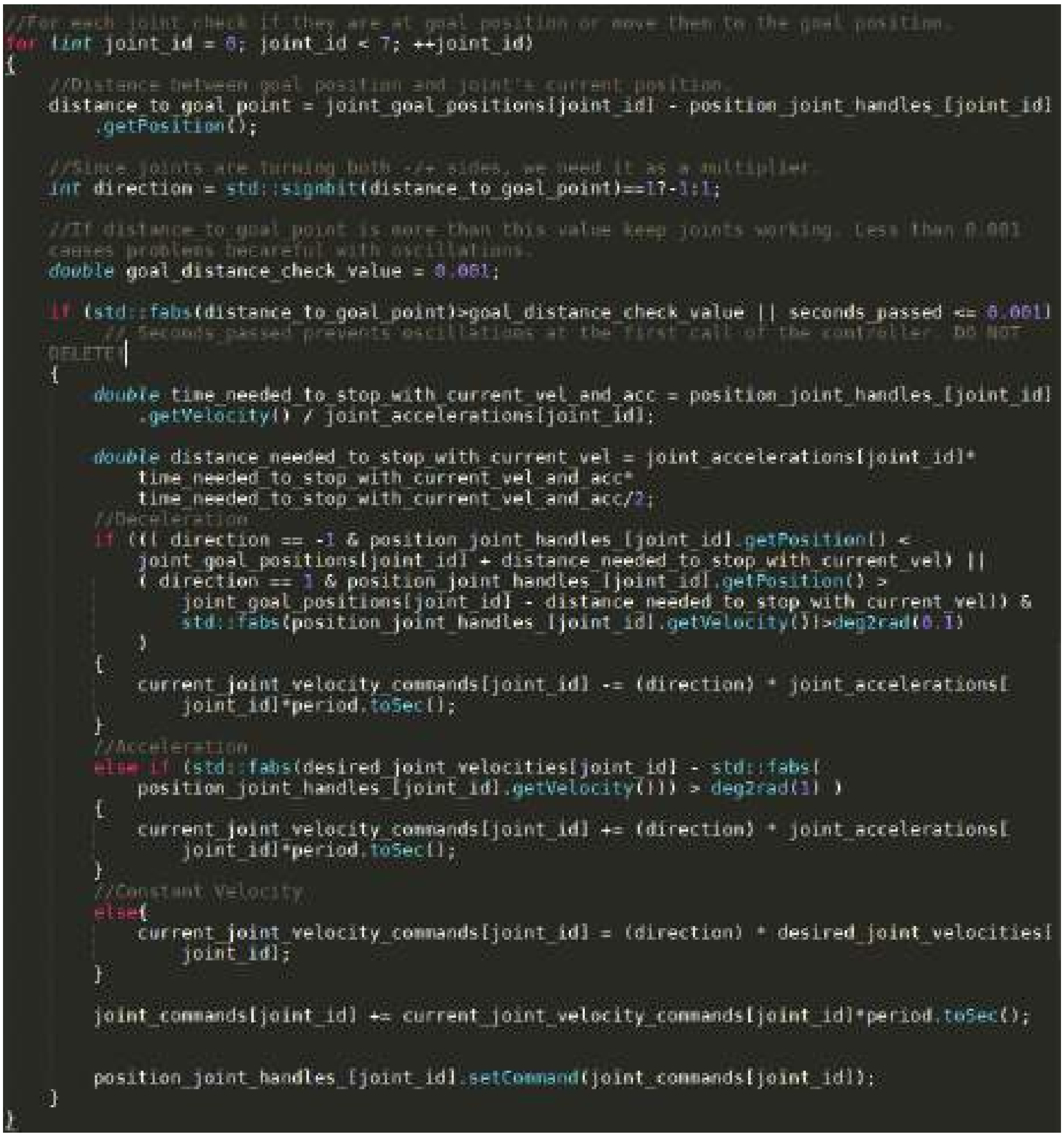
where $d_{deceleration}$ is the total distance needed to stop from current velocity with joint's deceleration limit $a_{constant}$ and deceleration time in seconds, $t_{deceleration}$.

26 Implementation of the joint position control with trapezoid velocity model

Beginning of the code is the same with the previous model. The distance to the goal position is checked, if it is over threshold then the joints are moved. In the if statement time needed to stop and distance needed to stop is calculated (figure 77).

Time needed to stop with current velocity and acceleration. From equation (14) time required to stop is found. It is assigned to the `time_needed_to_stop_with_current_vel_and_acc` variable.

Distance needed to stop with current velocity and acceleration. From equation (15) distance required to stop with joint's deceleration value is found. It is assigned to the variable `distance_needed_to_stop_with_current_vel`.



```
//For each joint check if they are at goal position or move them to the goal position.
for (int joint_id = 0; joint_id < 7; ++joint_id)
{
    //Distance between goal position and joint's current position.
    distance_to_goal_point = joint_goal_positions[joint_id] - position_joint_handles_[joint_id]
        .getPosition();

    //Since joints are turning both -/+ sides, we need it as a multiplier.
    int direction = std::signbit(distance_to_goal_point)==1?-1:1;

    //If distance to goal point is more than this value keep joints working. Less than 0.001
    causes problems becareful with oscillations.
    double goal_distance_check_value = 0.001;

    if (std::fabs(distance_to_goal_point)>goal_distance_check_value || seconds_passed <= 0.001)
        // Seconds passed prevents oscillations at the first call of the controller. DO NOT
    DELETE!
    {
        double time_needed_to_stop_with_current_vel_and_acc = position_joint_handles_[joint_id]
            .getVelocity() / joint_accelerations[joint_id];

        double distance_needed_to_stop_with_current_vel = joint_accelerations[joint_id]*
            time_needed_to_stop_with_current_vel_and_acc*
            time_needed_to_stop_with_current_vel_and_acc/2;
        //Deceleration
        if (( direction == -1 & position_joint_handles_[joint_id].getPosition() <
            joint_goal_positions[joint_id] + distance_needed_to_stop_with_current_vel) ||
            ( direction == 1 & position_joint_handles_[joint_id].getPosition() >
                joint_goal_positions[joint_id] - distance_needed_to_stop_with_current_vel)) &
                std::fabs(position_joint_handles_[joint_id].getVelocity())>deg2rad(0.1)
            )
        {
            current_joint_velocity_commands[joint_id] -= (direction) * joint_accelerations[
                joint_id]*period.toSec();
        }
        //Acceleration
        else if (std::fabs(desired_joint_velocities[joint_id] - std::fabs(
            position_joint_handles_[joint_id].getVelocity())) > deg2rad(1) )
        {
            current_joint_velocity_commands[joint_id] += (direction) * joint_accelerations[
                joint_id]*period.toSec();
        }
        //Constant Velocity
        else{
            current_joint_velocity_commands[joint_id] = (direction) * desired_joint_velocities[
                joint_id];
        }

        joint_commands[joint_id] += current_joint_velocity_commands[joint_id]*period.toSec();


        position_joint_handles_[joint_id].setCommand(joint_commands[joint_id]);
    }
}
```

Figure 77: update() function, calculates error and moves the joints towards the goal position.

Checking if joint reached to the deceleration point. Distance required to start decelerating with velocity of the joint at n^{th} period, v_n, and $a_{constant}$ is calculated at each period. An if statement checks if the joint's current distance to the goal position is less than `distance_needed_to_stop_with_current_vel`. If so, v_n is decreased with $a_{constant}$ until joint totally stops (equation 12). Joint positions to start decelerating in clockwise and counter-clockwise directions are depicted on the left and right side of the figure 78.

If joint has not reached to the desired velocity and deceleration point. Out of three phases mentioned at the beginning of the section 25.1, only acceleration satisfies this condition. From equation 12, commanded velocity, v_n, will be increased until it reaches the desired velocity or to the deceleration point.

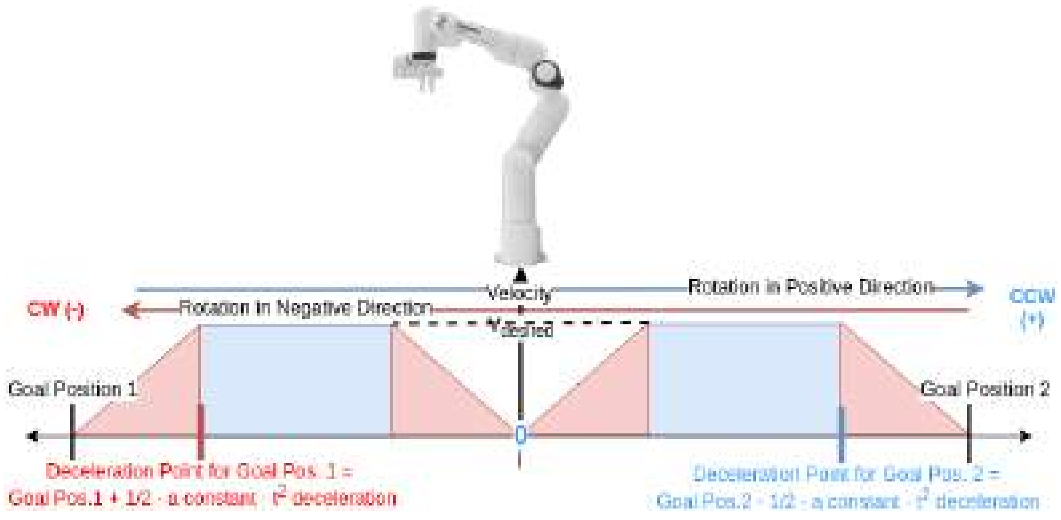
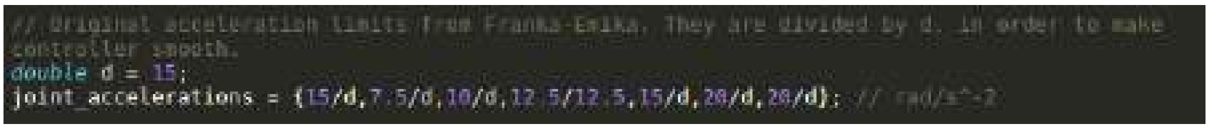

CW (-)
Rotation in Negative Direction
Velocity
Rotation in Positive Direction
CCW (+)
$V_{desired}$
Goal Position 1
Goal Position 2
0
Deceleration Point for Goal Pos. 1 = Goal Pos.1 + 1/2 · a constant · t² deceleration
Deceleration Point for Goal Pos. 2 = Goal Pos.2 - 1/2 · a constant · t² deceleration

Figure 78: Slowing down distances according to rotation directions.

If the joint reached to the desired joint velocity and not to the deceleration point. Joint will keep desired velocity. Set the v_n to $v_{desired}$.

The if statement which is satisfied calculates v_n, finally v_n is multiplied with `period.toSec()` to obtain joint command per period and sent to the Control via FCI.

Acceleration Limits. Acceleration limits are taken from Hardware Setup report, Table 3 [1]. But utilizing joints with maximum possible acceleration causes oscillations. A divider is added in order to make acceleration and deceleration smoother (figure 79). This vector is added to the header file and initialized at the `init()`.



```
// Original acceleration limits from Franka-Emika. They are divided by d. in order to make
controller smooth.
double d = 15;
joint_accelerations = {15/d,7.5/d,10/d,12.5/12.5,15/d,20/d,20/d}; // rad/s^-2
```

Figure 79: Accelerations limits from Hardware Setup report, Table 3 [1] divided by a value in order to achieve smoother acceleration and deceleration.

27 Experiments

Experiments are done in order to show the differences between sending joint position commands with and without controlling the acceleration-deceleration. At the conclusion chapter of "Single Joint Position Control with Constant Velocity Model for Franka Robot" report [2] it has been discussed that oscillations are caused by the drastic changes in velocities of the joints. By controlling acceleration, change of the speed smoothened which reduced oscillations drastically. In some of the charts a single wave of oscillation is visible. This is negligible since it is not repeating and joint is stopping at the goal position. If it is desired to prevent oscillations completely, acceleration limits can be reduced even further but this will also cause the completion times to be less accurate.

Joint 1 is robust in timings and oscillation. Only had a small oscillation at 5 deg/s but it is negligible. (figures 80, 81, 82).

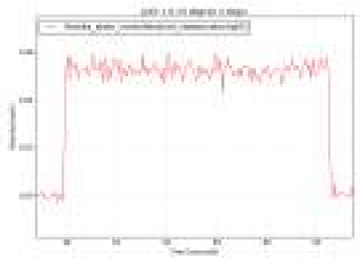


Figure 80: Joint 1: 0-15 degrees with 3 deg/s velocity. Completed in 5.2 seconds. No oscillation is visible on the chart.

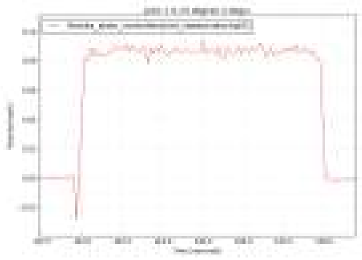


Figure 81: Joint 1: 0-15 degrees with 10 deg/s velocity. Completed in 3 seconds. A small wave of oscillation is visible on the left side of the chart but it is negligible.

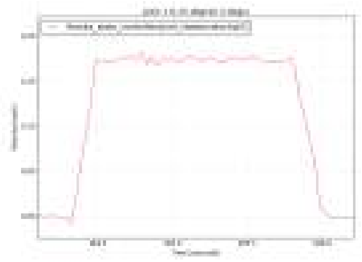


Figure 82: Joint 1: 0-15 degrees with 10 deg/s velocity. Completed in almost 3 seconds. No oscillation is visible on the chart.

Joint 2 and 4 are rotating around the global z-axis which is vertical to the gravitation. These two joints are carrying the weight of the next joints which connected to them that makes them prone to the oscillations due to the inertia of the next joints. Yet they are working precise enough, with a small wave of oscillation (figures 83, 84, 85).

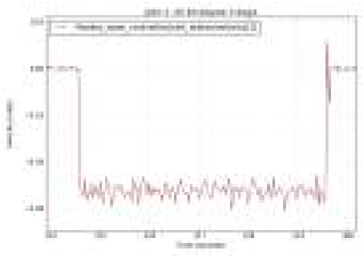


Figure 83: Joint 2: -45-60 degrees with 3 deg/s velocity. Completed in almost 5 seconds. A small wave of oscillation is visible on the left side of the chart.

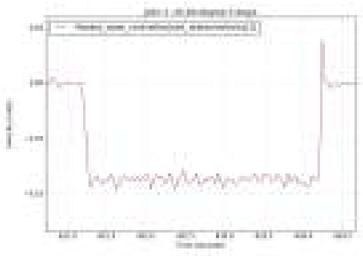


Figure 84: Joint 2: -45-60 degrees with 5 deg/s velocity. Completed in almost 3 seconds. A small wave of oscillation is visible on the left side of the chart.

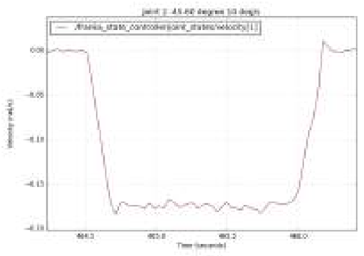


Figure 85: Joint 2: -45-60 degrees with 10 deg/s velocity. Completed in 1.65 seconds, 0.15 seconds more than expected. Negligible wave of oscillation is visible on the left side of the chart.

Joint 3 is working good. Only at 10 deg/s velocity command is not completed on exact time due to soft acceleration-deceleration (figures 86, 87, 88).

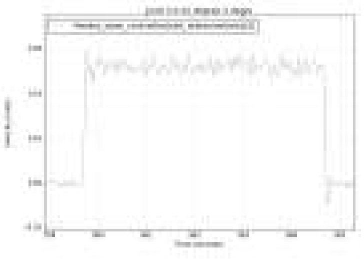


Figure 86: Joint 3: 0-15 degrees with 3 deg/s velocity. Completed in almost 5 seconds. No oscillation is visible on the chart.

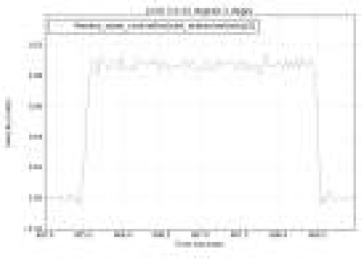


Figure 87: Joint 3: 0-15 degrees with 5 deg/s velocity. Completed in almost 5 seconds. No oscillation is visible on the chart.

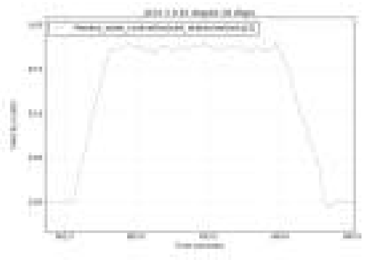


Figure 88: Joint 3: 0-15 degrees with 10 deg/s velocity. Completed in almost 1.75 seconds which is 0.25 seconds longer than expected. No oscillation is visible on the chart.

Joint 4 is rotating in global z-axis which makes it prone to the oscillations. Our command is moving joint against the gravity which makes a visible wave of oscillation at the beginning. But it is one wave which is acceptable under these conditions (figures 89, 90, 91).

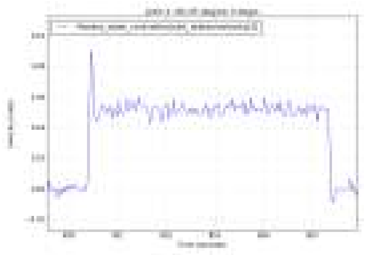


Figure 89: Joint 4: -60-45 degrees with 3 deg/s velocity. Completed in almost 5 seconds. A wave of oscillation is visible on the chart left side of the chart but it is negligible.

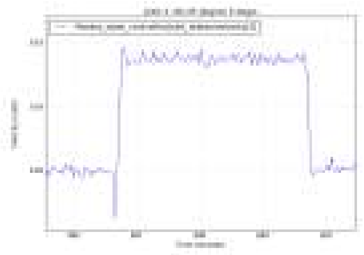


Figure 90: Joint 4: -60-45 degrees with 5 deg/s velocity. Completed in almost 3 seconds. A wave of oscillation is visible on the chart left side of the chart but it is negligible.

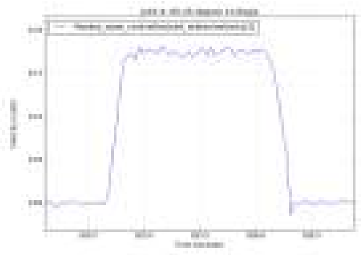


Figure 91: Joint 4: -60-45 degrees with 10 deg/s velocity. Completed in almost 1.75 seconds which is 0.25 seconds longer than expected. No visible oscillation.

Joint 5 has some oscillation waves at the end with 3 deg/s velocity but with 5 and 10 deg/s velocity oscillation is not visible (figures 92, 93, 94).

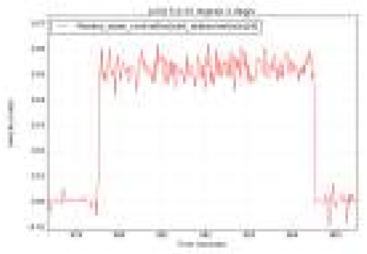


Figure 92: Joint 5: 0-15 degrees with 3 deg/s velocity. Completed in almost 5 seconds. Negligible oscillation is visible on the chart left side of the chart.

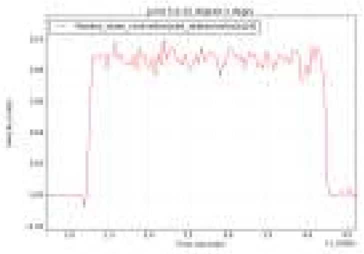


Figure 93: Joint 5: 0-15 degrees with 5 deg/s velocity. Completed in almost 3 seconds. No visible oscillation.

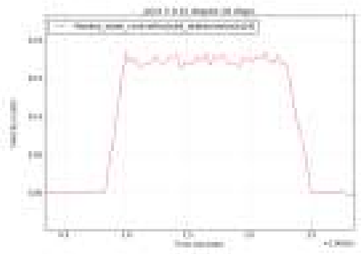


Figure 94: Joint 5: 0-15 degrees with 10 deg/s velocity. Completed in almost 1.75 seconds which is 0.25 seconds longer than expected. No visible oscillation.

Joint 6 had a small oscillation at 5 deg/s but no oscillation is visible at 3 and 10 deg/s (figures 95, 96, 97).

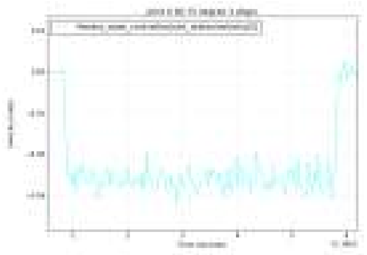


Figure 95: Joint 6: 90-75 degrees with 3 deg/s velocity. Completed in almost 5 seconds. No visible oscillation.

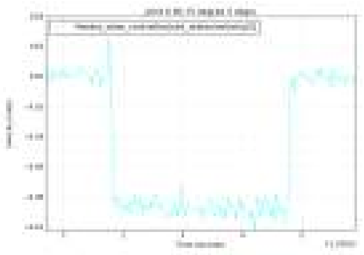


Figure 96: Joint 6: 90-75 degrees with 5 deg/s velocity. Completed in almost 3 seconds. Negligible oscillation is visible on the left side of the chart.

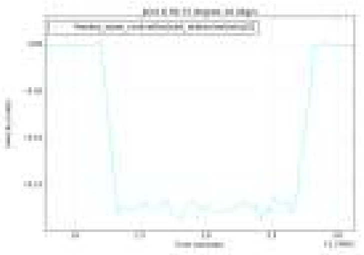


Figure 97: Joint 6: 90-75 degrees with 10 deg/s velocity. Completed in almost 1.5 seconds. No visible oscillation.

Joint 7 has no oscillation at all, works almost perfect (figures 98, 99, 100).

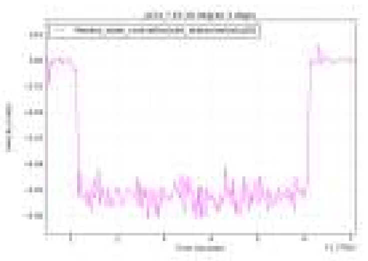


Figure 98: Joint 7: 45-30 degrees with 3 deg/s velocity. Completed in almost 5 seconds. No visible oscillation.

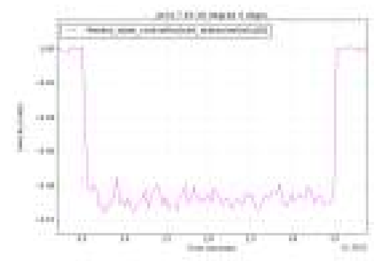


Figure 99: Joint 6: 90-75 degrees with 5 deg/s velocity. Completed in almost 3 seconds. No visible oscillation.

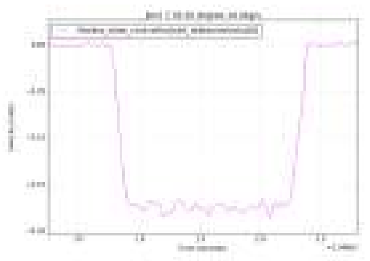


Figure 100: Joint 6: 90-75 degrees with 10 deg/s velocity. Completed in almost 1.5 seconds. No visible oscillation.

28 Conclusion

As we can see from the experiments, Trapezoid Velocity Model is working good with some negligible oscillations (where the oscillation is happening once and for a very short period of time). If more robust controller is needed with no oscillations at all, acceleration divider can be increased even more. But increasing the divider will cause timing accuracy to fall. User has to decide according to their usage of the robot.

29 Video review

I made a review video (figure 125) about joint position control with trapezoid velocity model showing oscillations from previous controller are gone.
You can reach the video from this link: `https://youtu.be/xvkW0rui68A`

Figure 101: A screenshot from review video.

30 Code

Code can be reached from:
`https://github.com/berktepebag/franka_robot_controllers/commits/joint_position_controller`

References

[1] Tepebag, B. (2019, July 6). Hardware Setup for Vision-based Control using Franka Robot

[2] Tepebag, B. (2019, June 26). Single Joint Position Control with Constant Velocity Model for Franka Robot

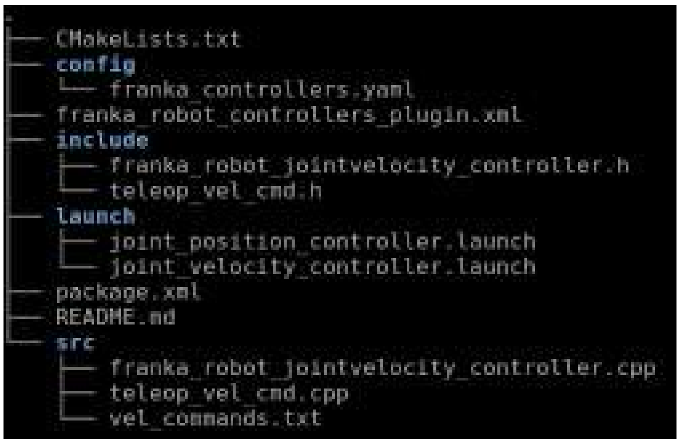
31 Joint Velocity Controller for Franka Robot - Introduction

This report shows how to write a "Joint Velocity Controller" for Franka robot. At the end of this report you will be able to control Franka robot with sending joint velocity commands for a duration.

The rest of the report is organized as follows: Section 32 shows how to write your own joint velocity controller. Section 33 shows how to modify teleop_cmd for velocity controller. Section 34, makes a summation of the controller and finally section 42 shows a demo of the joint velocity controller.

32 Writing your own "Joint Velocity Controller"

Last reports showed theory and implementation of joint position control of the Franka robot. Although we are controlling the velocity with trapezoid velocity model, sometimes it is necessary to control the robot with given velocity for a duration. Here velocity control implementation will be explained. At the end of the report, folder tree will look like figure 102.



```
.
├── CMakeLists.txt
├── config
│   └── franka_controllers.yaml
├── franka_robot_controllers_plugin.xml
├── include
│   ├── franka_robot_jointvelocity_controller.h
│   └── teleop_vel_cmd.h
├── launch
│   ├── joint_position_controller.launch
│   └── joint_velocity_controller.launch
├── package.xml
├── README.md
└── src
    ├── franka_robot_jointvelocity_controller.cpp
    ├── teleop_vel_cmd.cpp
    └── vel_commands.txt
```

Figure 102: Joint velocity controller folder tree.

32.1 Objective, given and knowns of the joint velocity controller

Objective of the joint velocity controller. In this new controller, user will send joint velocities and durations in order to control the Franka robot.

Given values. Joints' goal velocities and command durations will be given to the controller.

Known values. Velocity limits of the joints of the Franka robot can be reached from "Hardware Setup" report, Table 3: Joints' position, velocity, acceleration, jerk, torque, rotatum limits of Franka robot [1]. Seconds passed since the controller started, `seconds_passed` is known and will be used to calculate time to stop joints. Sequence of the message (number of the message sent from the publisher, can be obtained by `getSeq()` function) is known and will be used to move joints if new message is received.

Calculated values. Time to stop joints (in seconds) are calculated and hold in `joint_stop_seconds` vector.

32.2 Theory of the joint velocity control

Theory of the joint velocity controller is relatively easy. Joint position controller only takes in goal velocities and Franka robot tries to reach the given velocities within the hard and soft limits of the Franka robot's joints. Durations of the commands are the responsibility of the user. We have developed a new controller in order to move joints with given velocities during given durations.

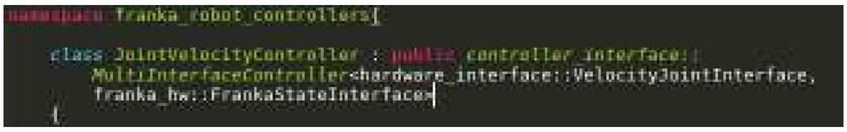
32.3 franka_robot_jointvelocity_controller.h

We can modify franka_robot_jointposition_controller.h since they are sharing most of the code. Make a copy of it and start editing with your favorite text editor.

We only have to add `#include <franka_hw/franka_state_interface.h>` to the imported libraries which is used to read Franka robot's states.

We will keep the same namespace, but change class name and type as depicted in figure 103:

```
   class JointVelocityController :  public controller_interface::
MultiInterfaceController<hardware_interface::VelocityJointInterface, franka_hw::FrankaSt
```

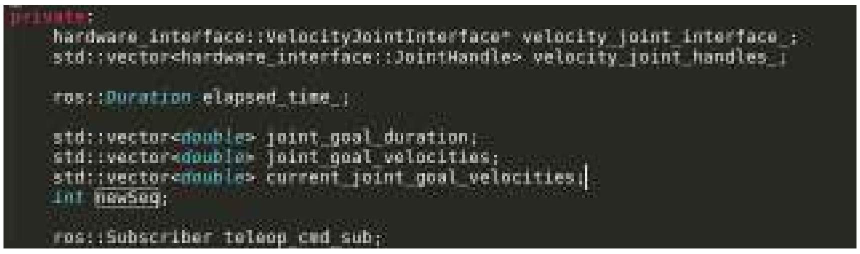


```
namespace franka_robot_controllers{

    class JointVelocityController : public controller_interface::
        MultiInterfaceController<hardware_interface::VelocityJointInterface,
        franka_hw::FrankaStateInterface>
    {
```

Figure 103: Class name and type for joint velocity control.

Since we are changing control method, we have to change hardware interface to the relative interface, which is `VelocityJointInterface`.

Private variables. To control Franka robot with velocity and duration we have to create 3 new vectors: one for duration, one for desired joint velocities and one for current velocity which will act as a placeholder for sending commands as depicted in figure 104. Also we are changing how we are sending and receiving messages from `teleop_cmd_vel`. From now on `teleop_cmd_vel` will only publish once when user enters a new order, this "message number" will be written to `newSeq` variable.

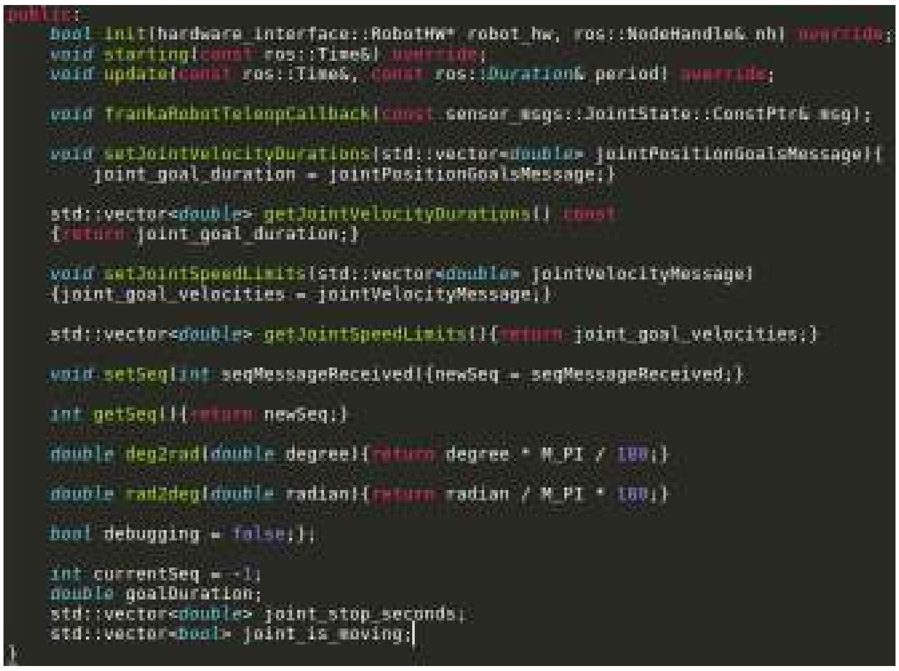


```
private:
    hardware_interface::VelocityJointInterface* velocity_joint_interface_;
    std::vector<hardware_interface::JointHandle> velocity_joint_handles_;

    ros::Duration elapsed_time_;

    std::vector<double> joint_goal_duration;
    std::vector<double> joint_goal_velocities;
    std::vector<double> current_joint_goal_velocities;
    int newSeq;

    ros::Subscriber teleop_cmd_sub;
```

Figure 104: Private variables of the joint velocity class.

Public variables. 3 functions will not change since they are necessary to run plugins: `init`, `starting` and `update`. Also it is not necessary to change `frankaRobotTeleopCallback` since we are not changing the topic we are subscribing (figure 105).

- `setJointVelocityDurations(std::vector<double> jointPositionGoalsMessage)` takes in duration messages from the related topic to set `joint_goal_duration` vector.
- `getJointVelocityDurations()` returns the `joint_goal_duration` vector.
- `setJointSpeedLimits(std::vector<double> jointVelocityMessage)` takes in velocity messages from the related topic to set `joint_goal_velocities` vector.
- `getJointSpeedLimits()` returns the `joint_goal_velocities` vector.
- `setSeq(int seqMessageReceived)` takes in sequence variable from the related topic. As new message is sent to the topic from `teleop_cmd_vel`, `currentSeq` will set equal to the sequence number received from the topic.
- `getSeq()` returns sequence number, `newSeq`.
- `deg2rad(double degree)` converts degree to radian.
- `rad2deg(double radian)` converts radian to degree.
- `int currentSeq` It is the number of last received messages sequence and used to check if new messages arrived.
- `std::vector<double> joint_stop_seconds;` Is used to calculate time to stop joints.



```
public:
    bool init(hardware_interface::RobotHW* robot_hw, ros::NodeHandle& nh) override;
    void starting(const ros::Time&) override;
    void update(const ros::Time&, const ros::Duration& period) override;

    void frankaRobotTeleopCallback(const sensor_msgs::JointState::ConstPtr& msg);

    void setJointVelocityDurations(std::vector<double> jointPositionGoalsMessage){
        joint_goal_duration = jointPositionGoalsMessage;}

    std::vector<double> getJointVelocityDurations() const
    {return joint_goal_duration;}

    void setJointSpeedLimits(std::vector<double> jointVelocityMessage)
    {joint_goal_velocities = jointVelocityMessage;}

    std::vector<double> getJointSpeedLimits(){return joint_goal_velocities;}

    void setSeq(int seqMessageReceived){newSeq = seqMessageReceived;}

    int getSeq(){return newSeq;}

    double deg2rad(double degree){return degree * M_PI / 180;}

    double rad2deg(double radian){return radian / M_PI * 180;}

    bool debugging = false;};

    int currentSeq = -1;
    double goalDuration;
    std::vector<double> joint_stop_seconds;
    std::vector<bool> joint_is_moving;
}
```

Figure 105: Public variables of the joint velocity class.

32.4 franka_robot_jointvelocity_controller.cpp

First thing we have to change is header file path:
`#include <franka_robot_jointvelocity_controller.h>` and add
`#include <hardware_interface/joint_command_interface.h>`

Caution! Since class name has changed do not forget to change every class name from `JointPositionController` to `JointVelocityController`.

32.4.1 JointVelocityController::frankaRobotTeleopCallback()

Use previously defined functions to set information from received messages to the related vectors.

- `this->setSeq(msg->header.seq);`
- `this->setJointVelocityDurations(msg->position);`
- `this->setJointSpeedLimits(msg->velocity);`

32.4.2 JointVelocityController::init() function

`position_joint_interface_` from previous controller modified to `velocity_joint_interface_`, change all the related variables and instantiate it with:

```
velocity_joint_interface_
= robot_hw->get<hardware_interface::VelocityJointInterface>();
```

State interface is used to get sensor readings from Franka robot.

```
auto state_interface = robot_hw->get<franka_hw::FrankaStateInterface>(); if (state_inter
== nullptr) { ROS_ERROR("JointVelocityController: Could not get state interface
from hardware!"); return false; }
```

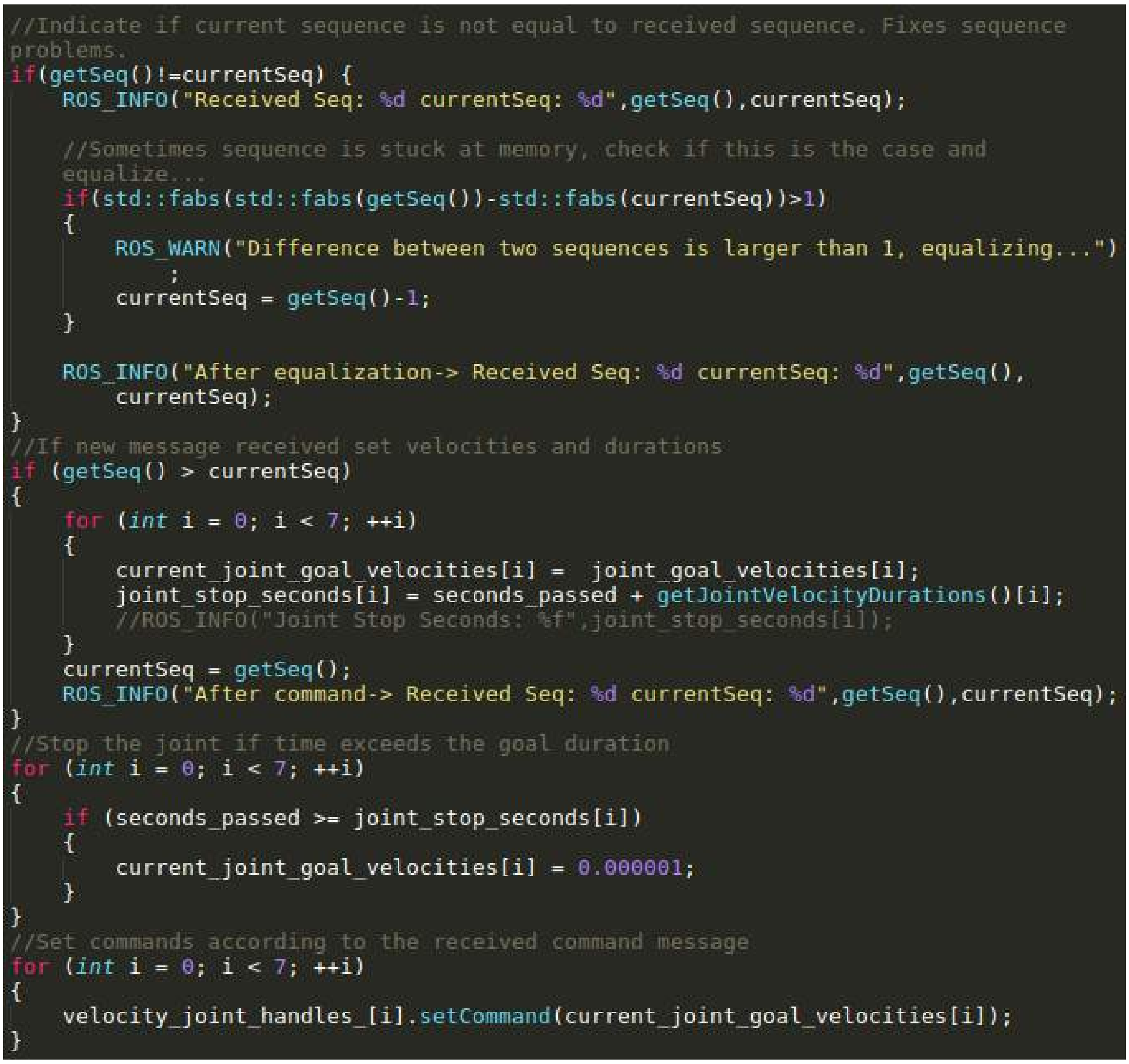
It checks if velocity interface is reachable, if not throws an error.

32.4.3 JointVelocityController::starting() function

While starting controller, `joint_goal_velocities,joint_goal_duration` and `current_joint_goal_velocities` are set to zero and `currentSeq` to -1;

32.4.4 JointVelocityController::update() function

In the update function, if sequence number received from topic message (obtained with `getSeq()` function) is higher than current sequence, `currentSeq`, a new message has arrived and joint will move.
`current_joint_goal_velocity` is set to received velocity. `goalDuration` is set to second which joint will stop moving. It is calculated by adding the goal duration received to the seconds passed since cotnroller started (`seconds_passed`) (figure 106).



```
//Indicate if current sequence is not equal to received sequence. Fixes sequence
problems.
if(getSeq()!=currentSeq) {
    ROS_INFO("Received Seq: %d currentSeq: %d",getSeq(),currentSeq);

    //Sometimes sequence is stuck at memory, check if this is the case and
    equalize...
    if(std::fabs(std::fabs(getSeq())-std::fabs(currentSeq))>1)
    {
        ROS_WARN("Difference between two sequences is larger than 1, equalizing...")
            ;
        currentSeq = getSeq()-1;
    }

    ROS_INFO("After equalization-> Received Seq: %d currentSeq: %d",getSeq(),
        currentSeq);
}
//If new message received set velocities and durations
if (getSeq() > currentSeq)
{
    for (int i = 0; i < 7; ++i)
    {
        current_joint_goal_velocities[i] =  joint_goal_velocities[i];
        joint_stop_seconds[i] = seconds_passed + getJointVelocityDurations()[i];
        //ROS_INFO("Joint Stop Seconds: %f",joint_stop_seconds[i]);
    }
    currentSeq = getSeq();
    ROS_INFO("After command-> Received Seq: %d currentSeq: %d",getSeq(),currentSeq);
}
//Stop the joint if time exceeds the goal duration
for (int i = 0; i < 7; ++i)
{
    if (seconds_passed >= joint_stop_seconds[i])
    {
        current_joint_goal_velocities[i] = 0.000001;
    }
}
//Set commands according to the received command message
for (int i = 0; i < 7; ++i)
{
    velocity_joint_handles_[i].setCommand(current_joint_goal_velocities[i]);
}
```

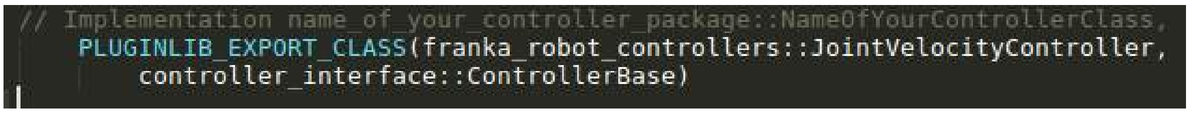
Figure 106: Update() function of the joint velocity class.

To prevent same message trigger the joint movement more than once, set `currentSeq` to last received sequence number.
`seconds_passed` is the total seconds passed since the controller started. If it is higher than the `goalDuration`, then controller should stop sending velocity commands. Because of a known error, setting joint velocity to zero causes connection error to be thrown. Instead very small number is set to stop the joint.
Finally, with a for loop each joint is set with the new command.

32.4.5 PLUGINLIB_EXPORT_CLASS.

Since we have created a new plugin, we have to declare it as depicted in figure 107.

```
    PLUGINLIB_EXPORT_CLASS(franka_robot_controllers
::JointVelocityController, controller_interface::ControllerBase)
```



```
// Implementation name_of_your_controller_package::NameOfYourControllerClass,
    PLUGINLIB_EXPORT_CLASS(franka_robot_controllers::JointVelocityController,
        controller_interface::ControllerBase)
```

Figure 107: PLUGINLIB_EXPORT_CLASS, changed name according to the class name.

32.5 Controller Configuration (yaml file)

Open franka_controllers.yaml file and add:

```
   joint_velocity_controller_pi:
type:  franka_robot_controllers/JointVelocityController
joint_names:
- panda_joint1
- panda_joint2
- panda_joint3
- panda_joint4
- panda_joint5
- panda_joint6
- panda_joint7
```

32.6 plugin Description File

Open franka_robot_controllers_plugin.xml file and add:

```
   <class name="franka_robot_controllers/JointVelocityController"
type="franka_robot_controllers::JointVelocityController"
base_class_type="controller_interface::ControllerBase">
<description>
Franka robot joint velocity controller, Pascal Institute
</description>
</class>
```

32.7 CMakeList.txt and Package.xml

32.7.1 Modifications in Package.xml

No modification done to Package.xml.

32.7.2 Modifications in CMakeList.txt

Inside the `add_library`, add newly created cpp files name.

```
   add_library(${PROJECT_NAME}
src/franka_robot_jointvelocity_controller.cpp )
```

32.8 joint_velocity_controller.launch file

Let's modify the joint_position_controller.launch and name it joint_velocity_controller.launch. And change:

```
   <node name="jointvelocity_controller_spawner"
pkg="controller_manager" type="spawner" respawn="false"
output="screen" args="joint_velocity_controller_pi"/>
```

32.9 Making the project

Last step is to make file:

- `cd ~/catkin_ws/`
- `catkin_make`
- `source devel/setup.bash`

And launch the controller, if you set `safety_check=true` do not forget to bring Franka robot to starting position.

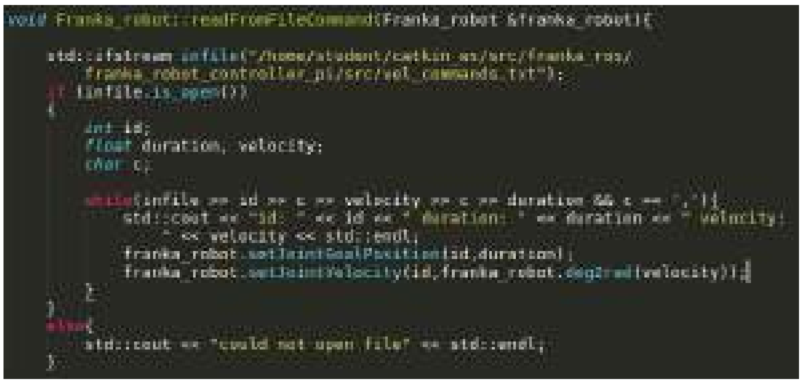
-
  ```
  roslaunch franka_robot_controller_pi
  joint_velocity_controller.launch
  ```

33 Teleop Command

33.1 teleop_cmd_vel.cpp

Start by making copies of the `teleop_cmd .cpp` and `.h`. Then rename them `teleop_cmd_vel .cpp` and `.h`. Only modification we did at `teleop_cmd` is sending duration using `sensor_msgs::JointState` message type's position message. Since it takes any numeric value, it is not necessary to create our own message type (figure 108).

- `franka_robot.setJointGoalPosition(id,duration);`
- `franka_robot.setJointVelocity(id,franka_robot.deg2rad(velocity));`



```
void Franka_robot::readFromFileCommand(Franka_robot &franka_robot){

    std::ifstream infile("/home/student/catkin_ws/src/franka_ros/
        franka_robot_controller_pi/src/vel_commands.txt");
    if (infile.is_open())
    {
        int id;
        float duration, velocity;
        char c;

        while(infile >> id >> c >> velocity >> c >> duration && c == ','){
            std::cout << "id: " << id << " duration: " << duration << " velocity:
                " << velocity << std::endl;
            franka_robot.setJointGoalPosition(id,duration);
            franka_robot.setJointVelocity(id,franka_robot.deg2rad(velocity));
        }
    }
    else{
        std::cout << "could not open file" << std::endl;
    }
```

Figure 108: Reading from txt file to command the Franka robot with velocity and duration.

34 Conclusion

Joint velocity control is used when joints desired to be controlled with user given velocity and duration. Franka robot's `VelocityJointInterface` provides joint velocity control but it does not provide control for duration. So we have implemented a duration control in our joint velocity controller. Usage of the controller can be seen at video review, in the next section.

35 Video review

I made a review video (figure 125) about joint velocity control for Franka robot. It shows the usage of the controller. You can reach the video from this link: `https://youtu.be/_6n2tGYREDs`

Figure 109: A screenshot from review video.

36 Code

Code can be reached from:
`https://github.com/berktepebag/franka_robot_controllers/tree/velocity_controller`

References

[1] Tepebag, B. (2019, July 6). Hardware Setup for Vision-based Control using Franka Robot

37 Cartesian Pose Control with MoveIt! for Franka Robot - Introduction

Franka Emika provides MoveIt! out-of-the-box [1] so that user can easily move Franka robot using RVIZ. But with version 0.6.0 of franka_ros, MoveIt! was not showing handles to move Franka robot in RVIZ. This report shows how to fix the "not showing handles in RVIZ problem" and how to write code to control real Franka robot with calculating joint positions for the desired pose by using Inverse Kinematic (IK) calculation functions from MoveIt! and sending commands with previously written Joint Position Control with Trapezoid Velocity Model.

Note: Since this problem is already been known by Franka Emika, problems mentioned here can be fixed with later releases. If you are doing a fresh installation, please try to run MoveIt! without applying fixes mentioned below. If it does not work check if this problems persists and then apply fixes.

38 Known Problems and Fixes

38.1 Key problems

Due to some security concerns, ROS is not updating with the old keys [2]. Old key has to be changed with the new key.

- old key: 421C365BD9FF1F717815A3895523BAEEB01FA116
- new key: C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654

Steps to change keys:

1. Remove the old key:
   `sudo apt-key del 421C365BD9FF1F717815A3895523BAEEB01FA116`
2. Import the new key:
   `sudo -E apt-key adv -keyserver 'hkp://keyserver.ubuntu.com:80' -recv-key C1CF6E31E6BADE8868B1`
   (If you cannot copy all at once first copy to a text file and then paste it to the terminal, or follow the link for original post [2].
3. Run update: `sudo apt-get update`

38.2 MoveIt! handles not visible

Going back to working commit

With the latest update (March, 2019) MoveIt! is not working out of the box. Handles for moving the Franka robot in RVIZ is not appearing due to some changes done probably in the URDF files. We can fix it with turning back to the working commit:

1. First go to the directory of the panda_moveit_config with:
   `roscd panda_moveit_config`
2. And checkout to the working commit:
   `git checkout 03bb4`
3. Go to catkin_ws
   `cd ˜\catkin_ws`
4. Compile:
   `catkin_make`
5. Source:
   `source devel\setup.bash`

Make sure, all the terminals are closed and open new terminals. Do not forget to `source` every new terminal you opened (If you added necessary shortcuts at `.bashrc` file, you do not have to source every new terminal).

38.3 Missing JointTrajectoryController

If "Joint Trajectory Controller" is not installed with any other package or project it will be missing and throw error:
`Could not load controller '...' because controller type 'position_controllers\JointTrajectoryController' does not exist.` To install JointTrajectoryController:

1. `sudo apt-get install ros-kinetic-joint-trajectory-controller`

39 Running MoveIt!

In order to start running MoveIt!:

```
roslaunch panda_moveit_config panda_control_moveit_rviz.launch
robot_ip:=172.16.0.2
```

To add MotionPlanning: Click Add->moveit_ros_visualization -> MotionPlanning -> OK to import Motion Planning (figure 110).

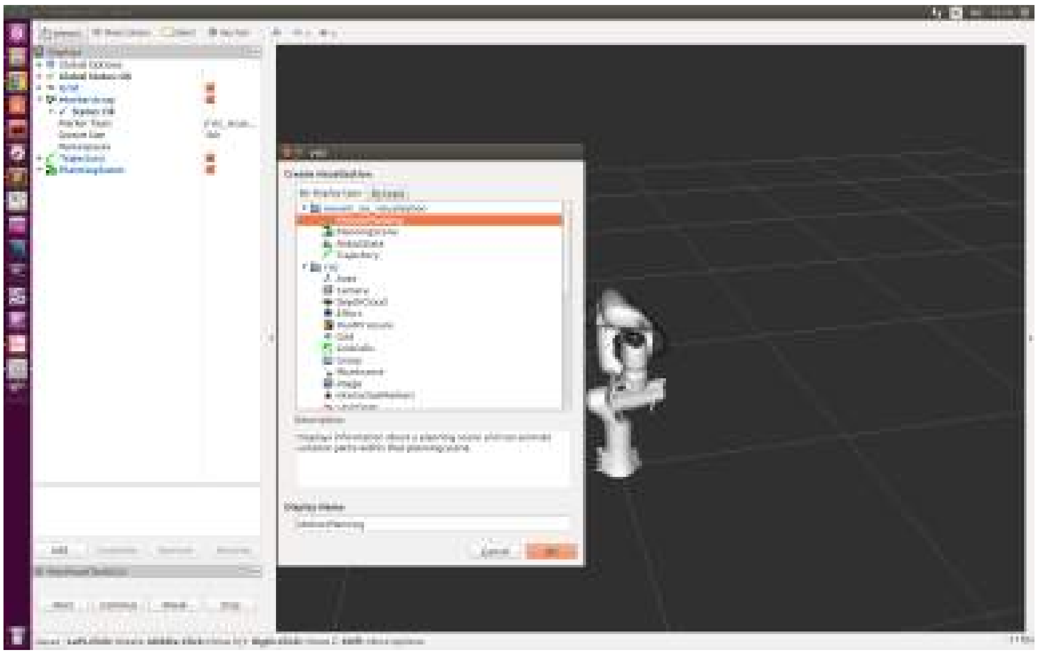


Figure 110: Importing Motion Planning.

To change Planning Group: Click MotionPlanning -> Planning Request -> Planning Group -> choose from drop down list panda_arm_hand (figure 111).

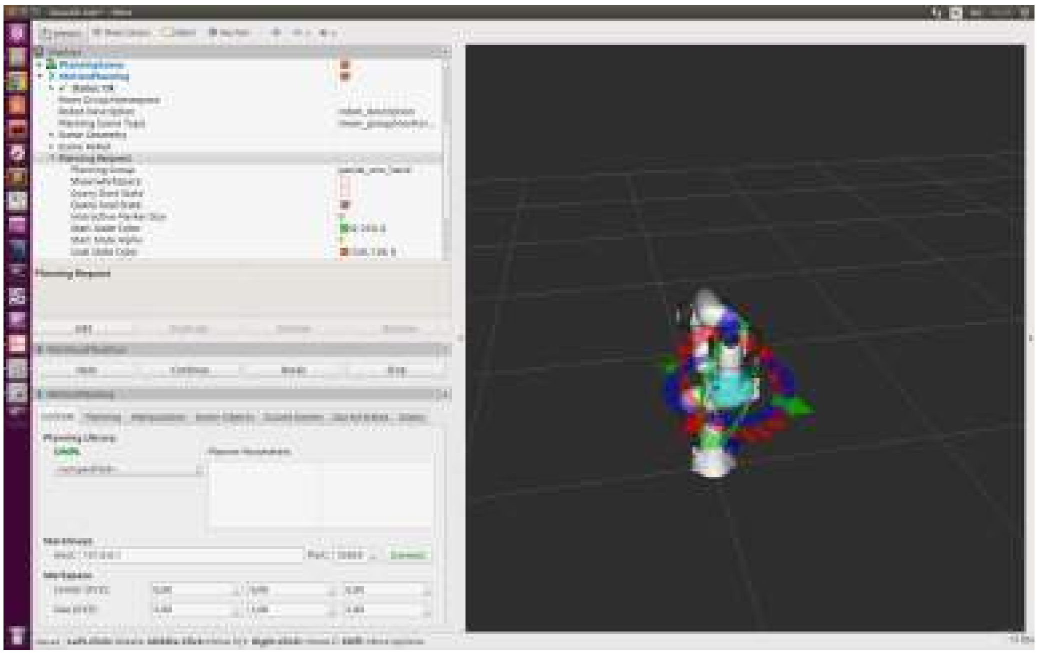


Figure 111: Changing Planning Group.

To plan and execute new pose: Move handles on the Franka robot to desired pose-> MotionPlanning -> Planning -> Plan and Execute (or first plan then execute) (figure 112).

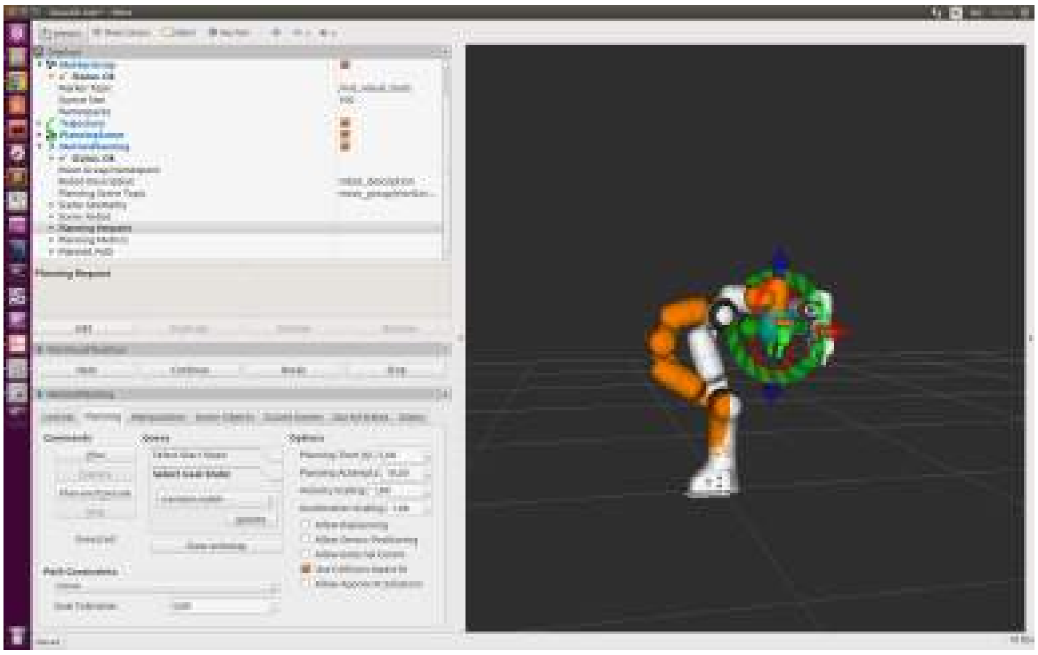
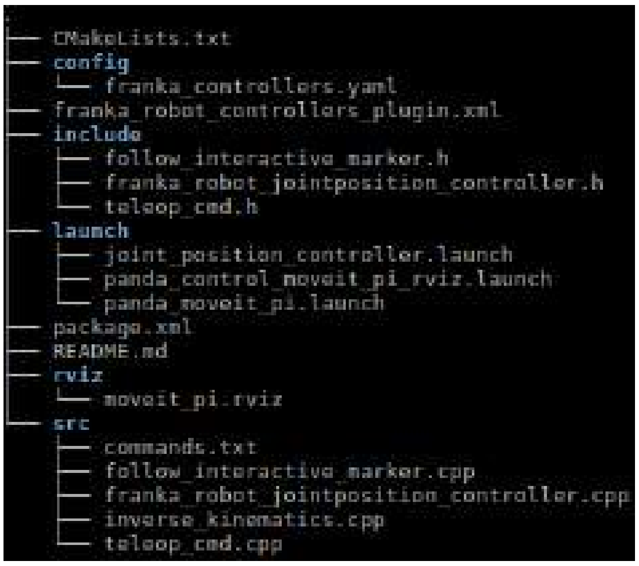


Figure 112: Planning new pose.

40 Implementing Joint Position Trapezoid Velocity Controller

After modifying files and creating new ones our folder directory will look like figure 113.



```
.
├── CMakeLists.txt
├── config
│   └── franka_controllers.yaml
├── franka_robot_controllers_plugin.xml
├── include
│   ├── follow_interactive_marker.h
│   ├── franka_robot_jointposition_controller.h
│   └── teleop_cmd.h
├── launch
│   ├── joint_position_controller.launch
│   ├── panda_control_moveit_pi_rviz.launch
│   └── panda_moveit_pi.launch
├── package.xml
├── README.md
├── rviz
│   └── moveit_pi.rviz
└── src
    ├── commands.txt
    ├── follow_interactive_marker.cpp
    ├── franka_robot_jointposition_controller.cpp
    ├── inverse_kinematics.cpp
    └── teleop_cmd.cpp
```

Figure 113: Folder tree.

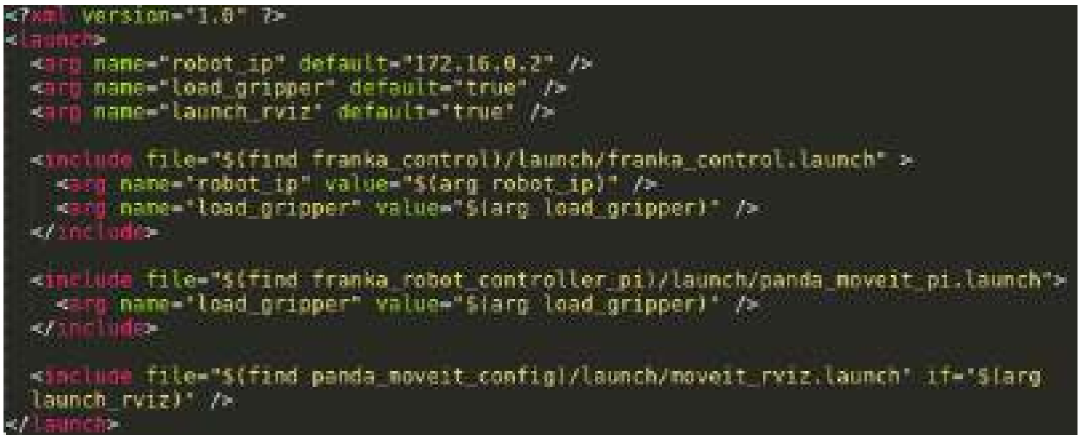
Until now we have used Franka Emika version of the MoveIt. We can use MoveIt's capabilities through GUI but if we want more control over it we have to start modifying the code.

40.1 Modifying launch files

40.1.1 roslaunch panda_moveit_config panda_control_moveit_rviz.launch

This launch file is used to call necessary controllers and config files in order to connect RVIZ, MoveIt and Franka robot.

Let's copy it under launch folder and rename:
`panda_moveit_config panda_control_moveit_pi_rviz.launch` so we can differ it from original file.



```
<?xml version="1.0" ?>
<launch>
  <arg name="robot_ip" default="172.16.0.2" />
  <arg name="load_gripper" default="true" />
  <arg name="launch_rviz" default="true" />

  <include file="$(find franka_control)/launch/franka_control.launch" >
    <arg name="robot_ip" value="$(arg robot_ip)" />
    <arg name="load_gripper" value="$(arg load_gripper)" />
  </include>

  <include file="$(find franka_robot_controller_pi)/launch/panda_moveit_pi.launch">
    <arg name="load_gripper" value="$(arg load_gripper)" />
  </include>

  <include file="$(find panda_moveit_config)/launch/moveit_rviz.launch" if="$(arg
  launch_rviz)" />
</launch>
```

Figure 114: panda_moveit_config panda_control_moveit_pi_rviz.launch file.

Let's give `robot_ip` a default value since we already arranged Control's IP: "172.16.0.2" as depicted at third line of the figure 114. Next two lines will stay same, we want gripper and RVIZ to launch when this launcher is called.

There are 3 included launch files:

1. `franka_control.launch`
2. `panda_moveit.launch`

3. `moveit_rviz.launch`

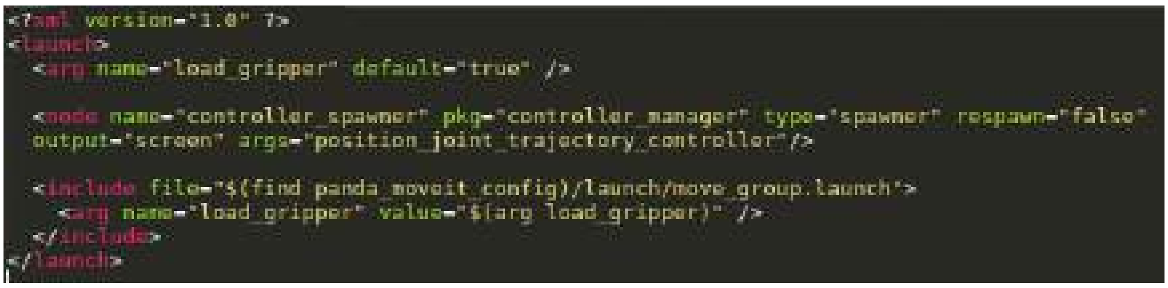
Let's check what they are responsible:

franka_control.launch It is located under `/opt/ros/kinetic/share/franka_control` folder and responsible from launching Franka robot related controllers, libraries etc.

panda_moveit.launch It is located under `/home/student/catkin_ws/src/panda_moveit_config/launch`, make a copy of it under our controllers launch folder and name it `panda_moveit_pi.launch`. This launcher calls

1. Controller manager which calls `position_joint_trajectory_controller`, a controller which is activated when pressed "Execute" at RVIZ that follows the calculated path.
2. `move_group.launch` which is also under `/home/student/catkin_ws/src/panda_moveit_config/launch`, that calls services like IK calculators (figure 115).

Since we already developed a joint position with trapezoid velocity controller we can use that to send selected position from MoveIt inside RVIZ.

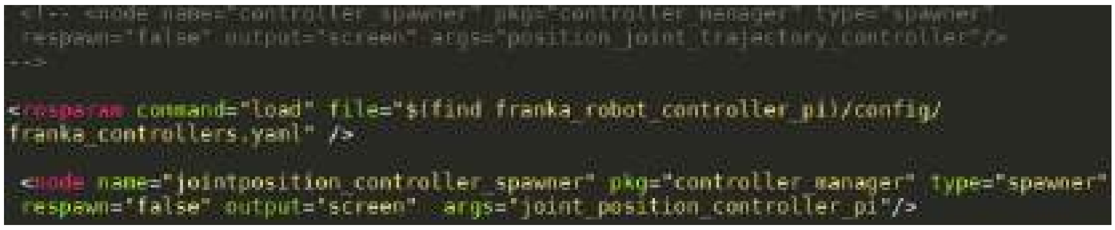


```
<?xml version="1.0" ?>
<launch>
  <arg name="load_gripper" default="true" />

  <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
  output="screen" args="position_joint_trajectory_controller"/>

  <include file="$(find panda_moveit_config)/launch/move_group.launch">
    <arg name="load_gripper" value="$(arg load_gripper)" />
  </include>
</launch>
```

Figure 115: panda_moveit.launch file.

Let's change controller with our controller as depicted in figure 116:

```
<rosparam command="load" file="$(find franka_robot_controller_pi)
/config/franka_controllers.yaml" />
<node name="jointposition_controller_spawner"
pkg="controller_manager" type="spawner"
respawn="false" output="screen" args="joint_position_controller_pi"/>
```



```
<!-- <node name="controller_spawner" pkg="controller_manager" type="spawner"
 respawn="false" output="screen" args="position_joint_trajectory_controller"/>
-->

<rosparam command="load" file="$(find franka_robot_controller_pi)/config/
franka_controllers.yaml" />

 <node name="jointposition_controller_spawner" pkg="controller_manager" type="spawner"
 respawn="false" output="screen"  args="joint_position_controller_pi"/>
```

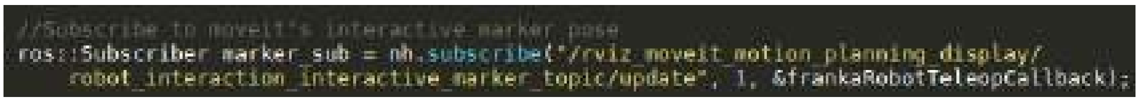
Figure 116: panda_moveit_pi.launch file.

Now instead of trajectory controller, our joint position with trapezoid velocity controller is called.

moveit_rviz.launch It is located under `/home/student/catkin_ws/src/panda_moveit_config/launch` and calls RVIZ related functions. Keep it as it is. (figure 116)

40.2 follow_interactive_marker.cpp

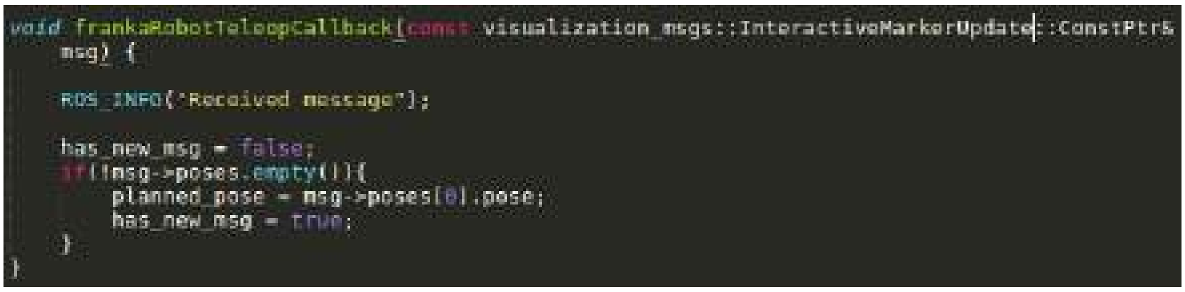
In order to control Franka robot using interactive markers from RVIZ and MoveIt!, first we need to receive pose of the end effector, solve with inverse kinematic solver and send newly found joint goal positions to our trapezoid velocity controller. Last part is not different than using `teleop_cmd`, we will send desired joint velocities and positions to the controller with
`follow_interactive_marker.cpp`. Since we changed our launch file to call our controller instead of trajectory controller, as soon as we start publishing commands Franka robot should start moving. We can get end effector pose from `/rviz_moveit_motion_planning_display`
`/robot_interaction_interactive_marker_topic/update` topic as depicted in figure 117. This topic is returning `visualization_msgs::InteractiveMarkerUpdate` type messages which is an array of poses that first one gives the last pose of the end effector.



```
//Subscribe to moveit's interactive marker pose
ros::Subscriber marker_sub = nh.subscribe("/rviz_moveit_motion_planning_display/
    robot_interaction_interactive_marker_topic/update", 1, &frankaRobotTeleopCallback);
```

Figure 117: Subscribing to the poses of the end effector.

When the message is received as depicted in figure 118, it is checked if the message is empty or not. If not empty, get the pose of the end effector and set `has_new_msg` true.

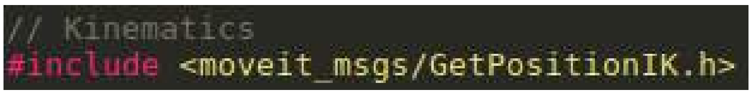


```
void frankaRobotTeleopCallback(const visualization_msgs::InteractiveMarkerUpdate::ConstPtr&
    msg) {

    ROS_INFO("Received message");

    has_new_msg = false;
    if(!msg->poses.empty()){
        planned_pose = msg->poses[0].pose;
        has_new_msg = true;
    }
}
```

Figure 118: Callback function to get poses from the topic.

40.2.1 Inverse Kinematic Solver

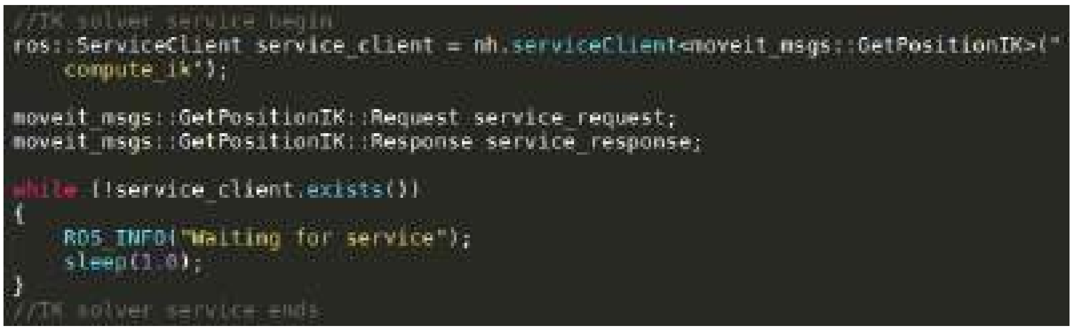
Since we are receiving end effector poses we have to convert them into joint space so that we can use our joint position controller with trapezoid velocity.
Add `#include <moveit_msgs/GetPositionIK.h>` header in order to be able to call service (figure 119).



```
// Kinematics
#include <moveit_msgs/GetPositionIK.h>
```

Figure 119: Inverse kinematic header.

Instantiate service, using `serviceClient`. Create two messages, request and response. If service is not available wait for a second till it is (figure 120).

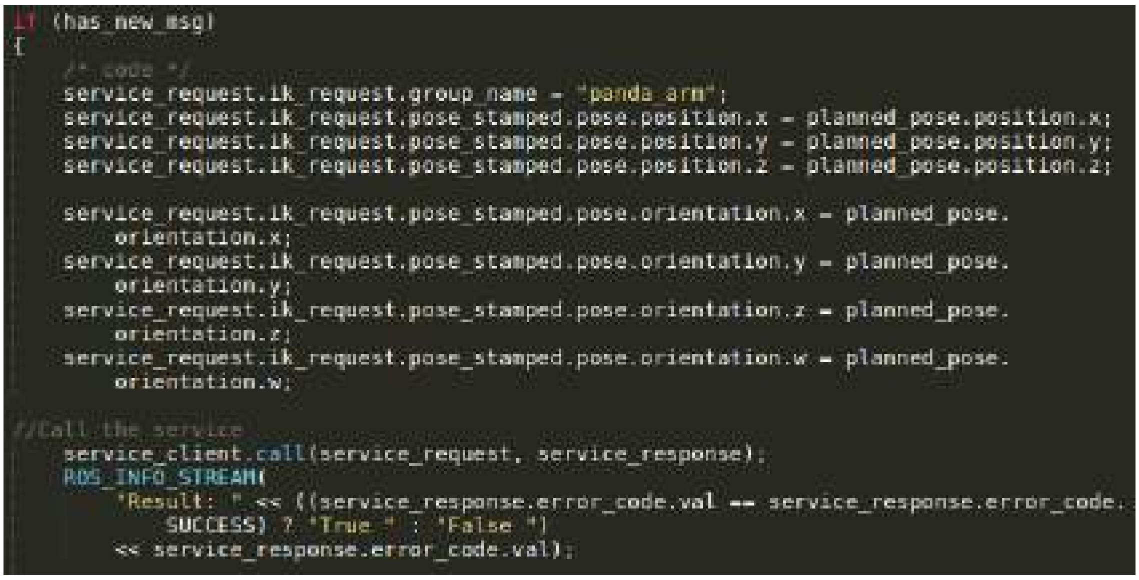


```
//IK solver service begin
ros::ServiceClient service_client = nh.serviceClient<moveit_msgs::GetPositionIK>("
    compute_ik");

moveit_msgs::GetPositionIK::Request service_request;
moveit_msgs::GetPositionIK::Response service_response;

while (!service_client.exists())
{
    ROS_INFO("Waiting for service");
    sleep(1.0);
}
//IK solver service ends
```

Figure 120: Inverse kinematic solver service.

40.2.2 While ROS is running

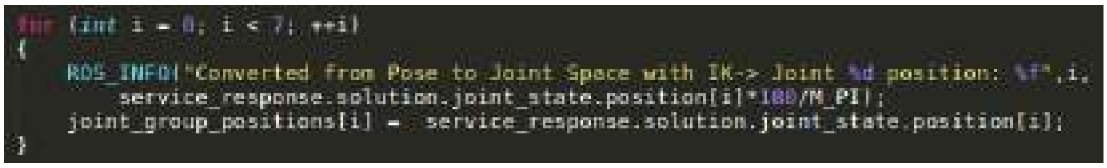
If ROS is running and we received a message from topic, we put it into to the request message (figure 121).



```
if (has_new_msg)
{
    /* code */
    service_request.ik_request.group_name = "panda_arm";
    service_request.ik_request.pose_stamped.pose.position.x = planned_pose.position.x;
    service_request.ik_request.pose_stamped.pose.position.y = planned_pose.position.y;
    service_request.ik_request.pose_stamped.pose.position.z = planned_pose.position.z;

    service_request.ik_request.pose_stamped.pose.orientation.x = planned_pose.
        orientation.x;
    service_request.ik_request.pose_stamped.pose.orientation.y = planned_pose.
        orientation.y;
    service_request.ik_request.pose_stamped.pose.orientation.z = planned_pose.
        orientation.z;
    service_request.ik_request.pose_stamped.pose.orientation.w = planned_pose.
        orientation.w;

//Call the service
    service_client.call(service_request, service_response);
    ROS_INFO_STREAM(
        "Result: " << ((service_response.error_code.val == service_response.error_code.
            SUCCESS) ? "True " : "False ")
        << service_response.error_code.val);
```

Figure 121: If received new message from the topic, make an IK solver request.

If IK solver solves the pose and returns joint angles successfully print information about it.

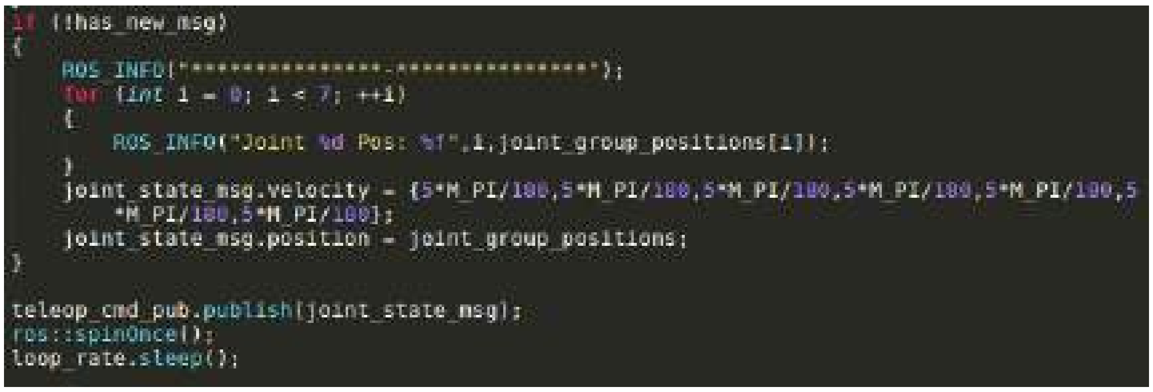


```
for (int i = 0; i < 7; ++i)
{
    ROS_INFO("Converted from Pose to Joint Space with IK-> Joint %d position: %f",i,
        service_response.solution.joint_state.position[i]*180/M_PI);
    joint_group_positions[i] =  service_response.solution.joint_state.position[i];
}
```

Figure 122: If received joint angles from IK solver put them into the joint group positions vector.

If received joint angles from IK solver put them into the joint group positions vector (figure 122).

When moving interactive marker in RVIZ, it publishes to the `/rviz_moveit_motion_planning_display/robot_interaction _interactive_marker_topic/update` topic. If we feed every received message into the controller, controller will have hard time to reach to the joint goals and start oscillating. To prevent this we are checking if user stopped moving the interactive marker, this way we are only receiving the final position of the interactive marker (figure 123).

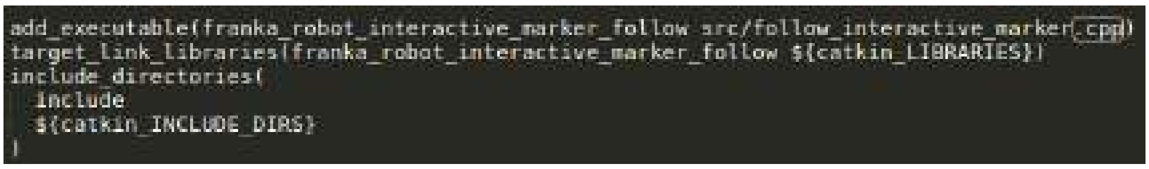


```
if (!has_new_msg)
{
    ROS_INFO("***************.***************");
    for (int i = 0; i < 7; ++i)
    {
        ROS_INFO("Joint %d Pos: %f",i,joint_group_positions[i]);
    }
    joint_state_msg.velocity = {5*M_PI/180,5*M_PI/180,5*M_PI/180,5*M_PI/180,5*M_PI/180,5
        *M_PI/180,5*M_PI/180};
    joint_state_msg.position = joint_group_positions;
}

teleop_cmd_pub.publish(joint_state_msg);
ros::spinOnce();
loop_rate.sleep();
```

Figure 123: If interactive marker stopped, start moving the Franka robot's joints.

40.3 CMakeLists.txt

We have to add our new .cpp file to the CMakeLists.txt as depicted in figure 124.



```
add_executable(franka_robot_interactive_marker_follow src/follow_interactive_marker.cpp)
target_link_libraries(franka_robot_interactive_marker_follow ${catkin_LIBRARIES})
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)
```

Figure 124: Add our .cpp file as executable in order to make it callable as rosrun file.

41 Conclusion

Franka Emika claimed that they provide working MoveIt! but last changes during spring of 2019 broke the package. We showed methods to fix the related problems. Later we modified given launch files to use our controller instead of trajectory controller. Also we had to add a inverse kinematic solver service to convert end effector pose into joint position in order to use our joint position with trapezoid velocity controller. Then subscribed to the related topic and used mentioned solver to get joint positions and send our controller. We did not have to make any changes at our controller since it was suitable for this kind of control. User has to be careful not to give new orders before the previous order is completed. A video review is showing how to use the new method to control Franka robot.

42 Video review

I made a review video (figure 125) about using MoveIt! interactive markers in RVIZ to control the Franka robot with our joint position control with trapezoid velocity controller.
You can reach the video from this link: `https://www.youtube.com/watch?v=jdClWZl2xdY`

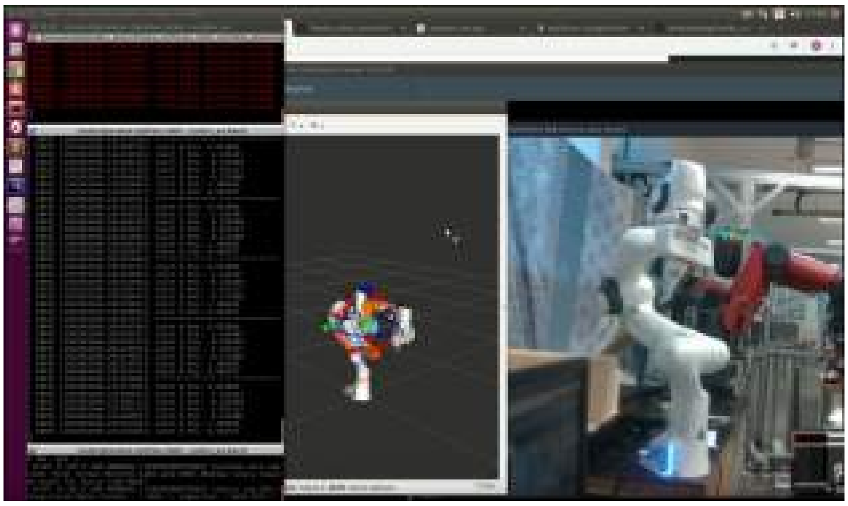


Figure 125: A screenshot from review video.

43 Code

Code can be reached from:
`https://github.com/berktepebag/franka_robot_controllers/tree/moveit_controller`

References

[1] Franka Emika. (2019, July 9). panda_moveit_config. Retrieved from https://frankaemika.github.io/docs/franka_ros.html#panda-moveit-config

[2] ROS Org. (2019, July 9). apt update fails / cannot install pkgs: key not working? https://answers.ros.org/question/325039/apt-update-fails-cannot-install-pkgs-key-not-working/

44 Final Conclusion

Increased competition in the robotics market brought new models and brands available to the users while being more affordable. This increase in the number of robots enforced the robotics companies to develop a common robotic operating system (ROS) which avoids the need for learning each of every brand's robotic software. This also made possible to transfer knowledge from different brand or models that makes easier to work with robots for the end-user. We had used Franka-Emika's 7 DOF Panda robotic arm to develop a new controller, 'joint position controller with trapezoid velocity model', which is a combination of the joint position controller and joint velocity controller. libfranka and franka_ros, libraries, that are provided by the Franka-Emika, has been used to develop novel controller. In the final step, we had combined our novel controller with the MoveIt! library to send commands from RVIZ simulator which provides an easy to use GUI and inverse kinematics solver. Our system is now ready to start developing machine-learning algorithms to find unknown deformable materials properties. To do so, deep-learning methods such as 'convolutional neural networks' and 'recurrent neural networks' or their combinations can be used. Also using reinforced deep-learning methods within the simulators can drastically reduce the time and money needed to train the networks unlike using real robots.