# Force-torque control of robot under ROS2

Lucie Vajnerová

June 23, 2023

## 1 Introduction

Our goal was to work with the Panda robot from Franka Emika robotics company. This is a force-torque compliant robotic arm with 7 degrees of freedom, allowing for advance torque control. As our middleware we chose the Robotic Operating System (ROS) for it is an open-source framework that provides a collection of software libraries and tools for developing robot applications.

## 2 Software

### 2.1 ROS1 and ROS2

ROS currently has two supported versions: ROS1 and ROS2 [1]. In our work, we chose to work with ROS2 over the older ROS1. There are a few key major differences between these two versions [2]. Firstly, ROS1 operates under a master-slave architecture. It has a ROS Master (a sort of DNS server), that is there to take care of discovery and communication between all other 'Slave' nodes. ROS2 uses a decentralized architecture. Each node is then fully independent and not tied to a global master. ROS2 also provides control with real-time communication. ROS2 uses Data Distribution Service (DDS), thanks to which it can provide support for real-time control. ROS2 is much more modular, with greater control over system dependencies, allowing users to integrate only selected components and libraries compared to ROS1 rigid approach. ROS2 also provides QOL services and better security with encryption and authentication mechanisms. ROS2 supports multiple nodes running in parallel. ROS2 can also be run under different distributions than linux.

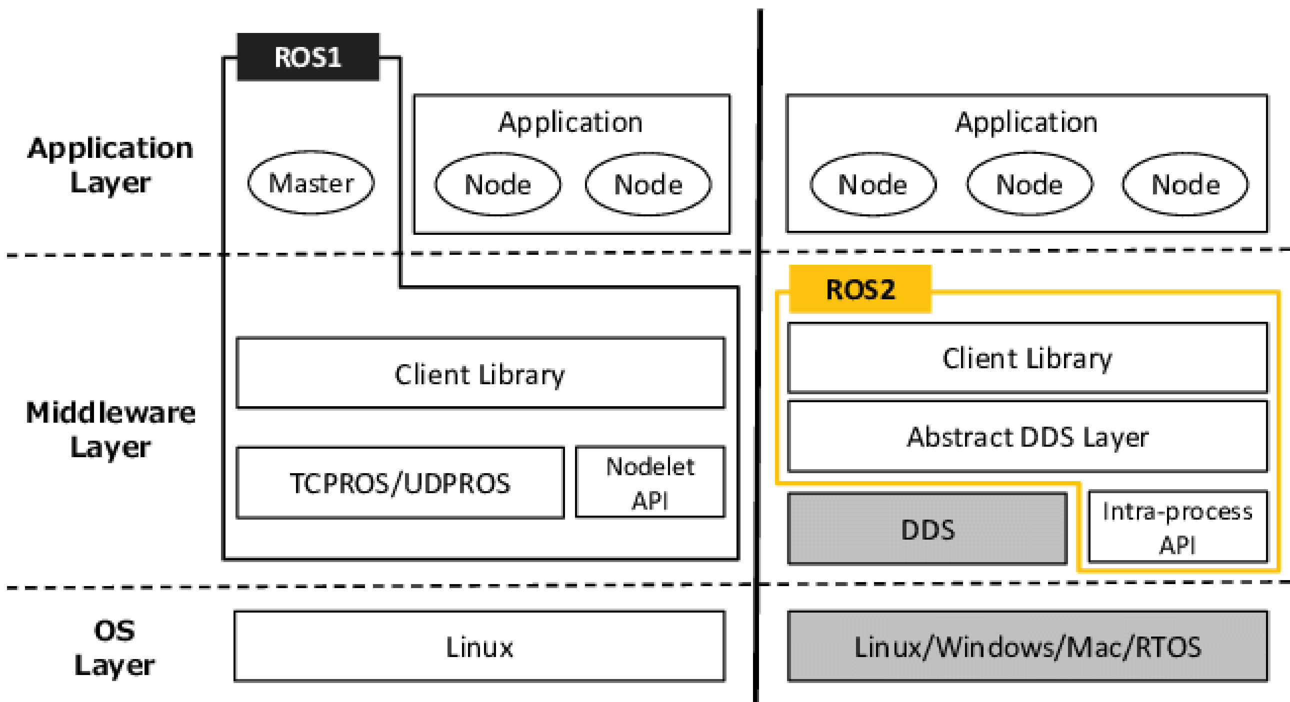



Figure 1: Comparison between ROS1 and ROS2 [3]

In connection with ROS2, we had to use libfranka [4], a C++ library that provides low-level control of Franka Emika research robots. To connect this library to our middleware ROS2, we used franka_ros2 [5]. At time of working on this projects, this library was still released only as a beta-version. This provided unique challenges, such as missing integration for computation of Jacobian straight over ROS, as was the case under ROS1.

As of 25. May 2023, there are three current distributions of ROS2 [6]. These are (from oldest to newest): Foxy Fitzroy, Humble Hawksbill and Iron Irwini. During our work on this project, we had experimented with both Foxy and Humble, but most of our code was written with Foxy as our primary goal. Iron was released on 23. May 2023 and as such was not part of our work on this project.

## 2.2 Algorithm Libraries

We had primarily used two libraries for numerical algorithms. Both of these libraries can function on generic legged robots and to use it for a specific robot (e.g. Panda robot) the user must provide its Unified Robotics Description Format, URDF. This is an XML specification used in academia and industry to model multibody systems.

### 2.2.1 Pinocchio

Pinocchio is an open-source software framework that implements rigid body dynamics algorithms and their analytical derivatives. Pinocchio does not only include standard algorithms employed in robotics (e.g., forward and inverse dynamics) but provides additional features essential for the control, the planning and the simulation of robots [7]. The Pinocchio website provides numerous examples of use and was highly educational for our purposes [8].

### 2.2.2 TSID

The Task Space Inverse Dynamics is a control framework [9]. It provides users with inverse-dynamics controllers for legged robots. It operates similarly to Least-Squares Programs, a form of Quadratic Programming. This library uses Pinocchio as its base and together they form a powerful tool for developing controllers on robotic arms (such as our Panda robot).

## 2.3 MoveIt2

MoveIt 2 is a robotic manipulation platform for ROS 2, and incorporates the latest advances in motion planning, manipulation, 3D perception, kinematics, control, and navigation [10]. MoveIt2 has running configurations for both ROS2 Foxy and ROS2 Humble, with differences between their implementations. It provides a MoveIt2 plugin known as MoveIt2 Servo that provides direct control over end effector velocity commands [11].

## 2.4 Low Level Control

The `ros2_control` is a framework for (real-time) control of robots using ROS2. The controllers in the ros2_control framework are based on control theory and compare the reference value with the measured output and, based on this error, calculate a system's input [12]. This framework also provides a simple user interface over command services.

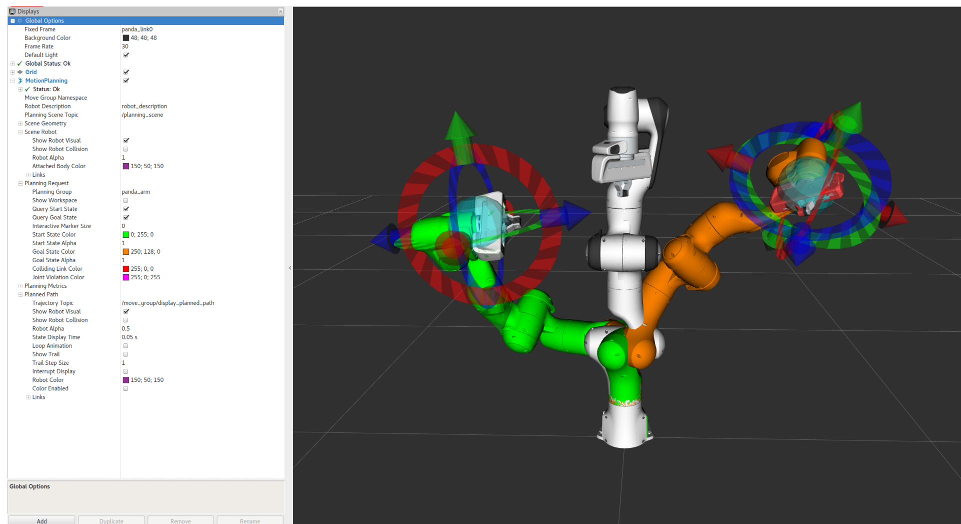
## 2.5 Visualization

There are many options for visualizing not only our robotic arm and running its controllers to observe them in virtual setting, but also to monitor incoming and outgoing data for correct computation and debugging. Our tools included:

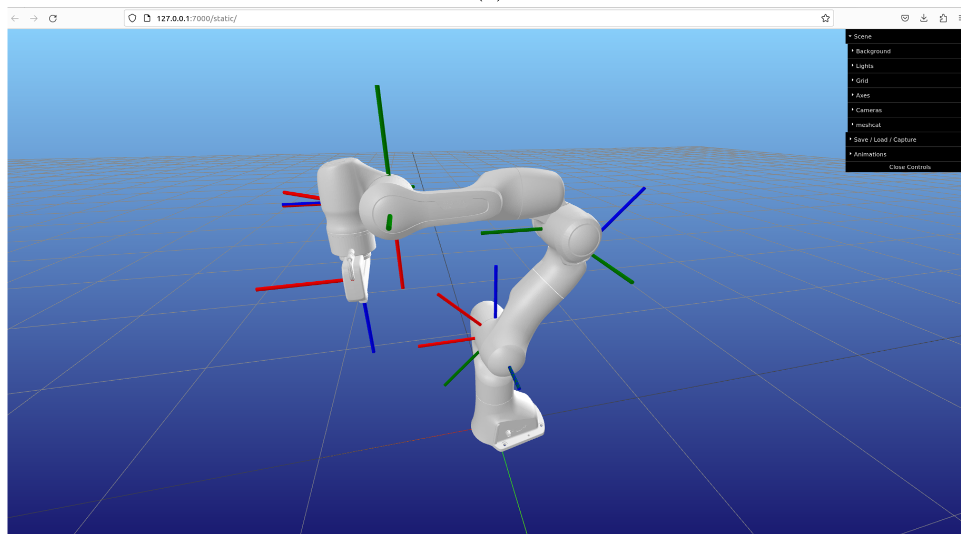
a. Rviz2 - a 3D visualization tool for ROS2. It provides a graphical user interface (GUI) that enables users to visualize sensor data, robot models, trajectories, and various other types of data related to robot operation.

b. Meshcat - a 3D visualization program that provides interactive visualization capabilities for robotics and simulation environments. It is designed to be lightweight, web-based, and highly customizable.

c. PlotJuggler - a data visualization and analysis tool specifically designed for time-series data. It allows users to efficiently visualize, analyze, and compare multiple time-series data streams in a user-friendly and interactive manner.



(a)



(b)

Figure 2: (a) Rviz2 GUI. (b) Meshcat Vizualization.

d. Foxglove Studio - a data visualization and analysis tool designed specifically for robotics and autonomous systems. Its capabilities are similar to PlotJuggler, but sadly seems to be less consistent and robust.

# 3 Custom Controllers Structure

Inside a ROS2 package we can create controller for low level control of robots using ros2_control. We do this by creating C++ file and its header that will contain the implementation of our controller. After we add declarations into header file - most notably declaring the class of file as a ControllerInterface - we can start writing the controller methods into corresponding <controller_name>.cpp file: init, command_interface_configuration, state_interface_configuration, on_configure, on_activate, on_deactivate and update.

a. init - initialize member variables, reserve memory, and most importantly, declare node parameters used by the controller.

b. on_configure - parameters are usually read here, and everything is prepared so that the controller can be started.

c. command_interface_configuration, state_interface_configuration - here the required interfaces are defined (e.g. effort or velocity interfaces).

d. on_activate - checks and potentially sorts the interfaces and assigning members' initial values. This method is part of the real-time loop, so it's good to keep it as short as possible.

e. on_deactivate - does the opposite of on_activate and often this method is empty.

f. update - when this method is called, the state interfaces read the most recent values from the hardware, and new commands for the hardware should be written into command interfaces. The method should be implemented with real-time constraints in mind.

It's important not to forget at the end of our file (after the namespace is closed) to add the PLUGINLIB_EXPORT_CLASS macro.

Inside the same package, it's necessary to include <controller_name>.xml file and add a definition of the library and controller's class which has to be visible for the pluginlib. After this, the controller can be run after ensuring all necessary steps for CMakeLists.txt and package.xml inclusion.

A more detailed version of this procedure can be found in ros2_controllers documentation [13].

# 4 Example Controllers Implementation

Below are some example controllers we have tested during our work on this project [14]. In each, we will list their main computation method used as part of the update method and other necessary things. Panda robotic arm has only one command interface - the effort command interface - and as such each controller below presumes we need to send desired torque values for each robot joint (7 in total) to the effort command interface.

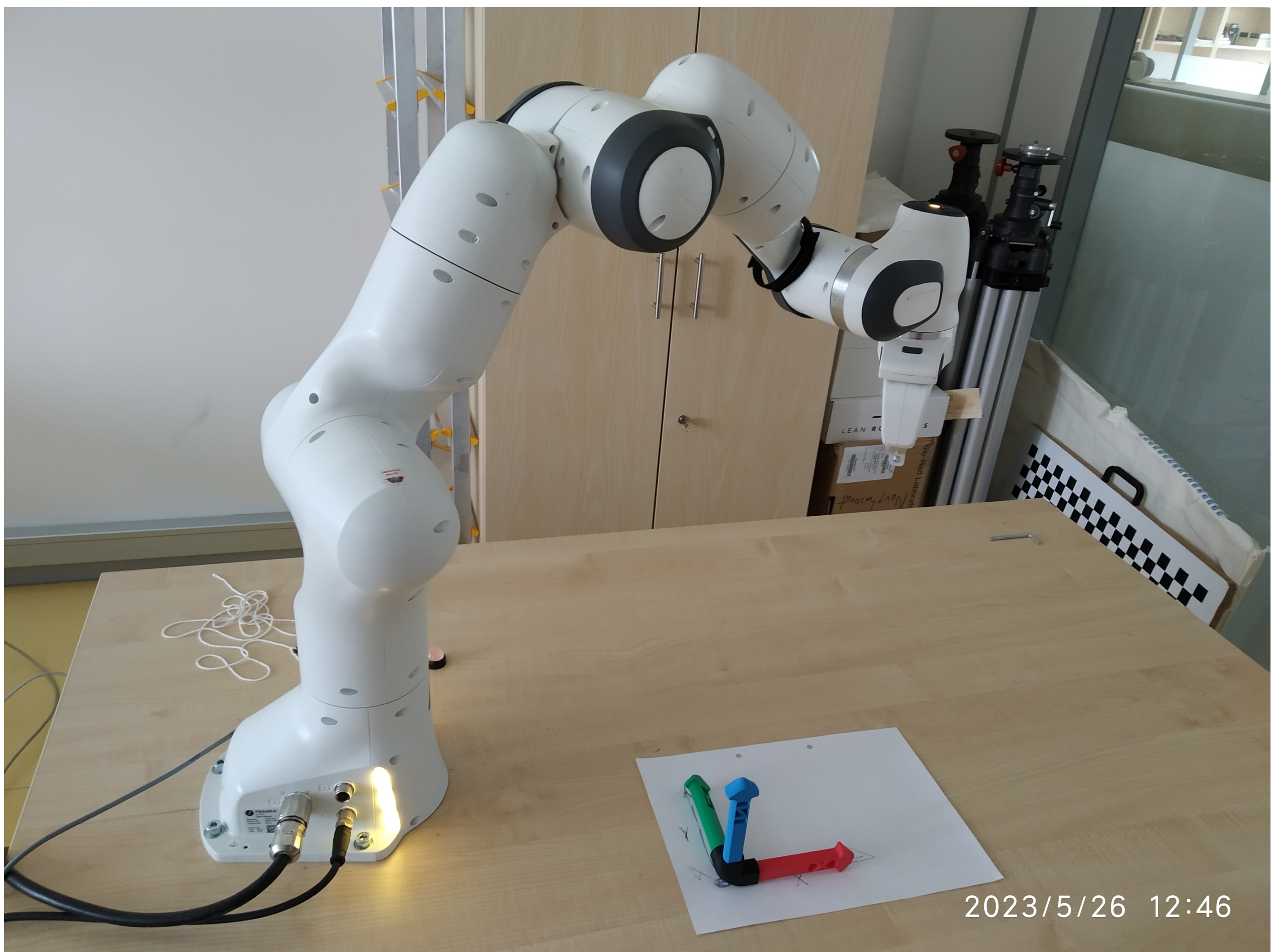



Figure 3: The Panda robotic arm during run of example controllers

Some controllers we implemented used the Pinocchio library. There are a few important functions we used:

a. rnea(model, data, q, v, a): The Recursive Newton-Euler algorithm. It computes the inverse dynamics, aka the joint torques according to the current state of the system, the desired joint accelerations and the external forces. Its inputs are: *model* - the model structure of the rigid body system, *data* - the data

structure of the rigid body system, $q$ - the joint configuration vector, $v$ - the joint velocity vector, $a$ - the joint acceleration vector.

b. `computeGeneralizedGravity (model, data, q)`: Computes the generalized gravity contribution $g(q)$ of the Lagrangian dynamics. Its parameters are: *model* - the model structure of the rigid body system, *data* - the data structure of the rigid body system, $q$ - the joint configuration vector.

c. `forwardKinematics(model,data,q)`: Update the joint placements according to the current joint configuration. Its parameters are: *model* - the model structure of the rigid body system, *data* - the data structure of the rigid body system, $q$ - the joint configuration vector.

## 4.1 Gravity Controller

The aim of this controller is to allow free movement of the robotic arm with respect to operator's movements. The arm moves without resistance if operator tries to move the arm around. This is achieved by sending the desired torque values as equal to zero. This means the robot only has to compensate for its weight, but nothing more.

$$\vec{\tau} = 0 \tag{1}$$

## 4.2 Snake Controller

This controller tries to implement numerous control versions based on similar algorithms created by Mederic Fourmy [15] with the help of the Pinocchio library. These methods are based on Lagrangian dynamics

$$\vec{\tau} = M(\theta) \cdot \ddot{\theta} + c(\theta, \dot{\theta}) + g(\theta) + J^T \cdot f, \tag{2}$$

where $M$ is the mass matrix, $c$ is the velocity-product (Coriolis and centripetal) term and $g$ is the gravity term. Next, $\theta$ is the joint positions, $\dot{\theta}$ is the joint velocities and $\ddot{\theta}$ is the joint accelerations. The last term $J^T \cdot f$ is composed of the Jacobian transpose and the wrench that the end-effector applies to the environment. For this controller we will presume the wrench is equal to zero and thus get

$$\vec{\tau} = M(\theta) \cdot \ddot{\theta} + c(\theta, \dot{\theta}) + g(\theta). \tag{3}$$

There are four control variants for computing desired torque as part of this controller. These variants are: IDControl, IDControlSimplified, PDGravity and PureGravity. For each control version we can read current joint positions, their velocities and the effort on those joints (from state interfaces). The requested joint positions, velocities and accelerations are computed from two options of trajectories: sinus trajectory and homing trajectory. Sinus reference trajectory moves all joints positions along a sinusoid path. The homing trajectory takes as input a desired end configuration of joint positions and computes joint positions, velocities and acceleration within specified time reference.

### 4.2.1 IDControl

This inverse dynamics control variant first computes acceleration level feedback law

$$\ddot{\theta_d} = \ddot{\theta_r} - K_p \cdot (\theta_m - \theta_r) - K_d \cdot \Delta e, \tag{4}$$

where $\ddot{\theta_r}$ is the desired acceleration, $K_p$ and $K_d$ are constant parameters for ID control. Next, $\theta_m$ are measured joint positions, $\theta_r$ are requested joint positions. Lastly, $\Delta e$ is the derivative joint trajectory error.

To find the necessary $\vec{\tau}$, we use the Pinocchio library. More specifically we use its `rnea()` function that computes the inverse dynamics, aka the joint torques according to the current state of the system and the desired joint accelerations. This is also known as the Recursive Newton-Euler algorithm. However, for Pinocchio we need to subtract the gravity term from $\tau_{rnea}$ to get only centrifugal and Coriolis forces. Thus, we use Pinocchio library again to compute generalized gravity currently acting on the robot (function `computeGeneralizedGravity()`) - note: both functions require correct setup of Panda robot model and its URDF.

Finally, we compute our desired torque commands as

$$\vec{\tau} = \vec{\tau_{rnea}} - \vec{g}. \tag{5}$$

### 4.2.2 IDControlSimplified

In this inverse dynamics control simplified variant we skip over the computation of $\ddot{\theta_d}$ and use the Recursive Newton-Euler algorithm with our requested joint accelerations instead. We find $\tau_{feedback}$ from position error and its derivative with gain arrays product wise multiplication to 'weight' these gains:

$$\vec{\tau_{feedback}} = -\vec{k_p} \cdot (\theta_m - \theta_r) - \vec{k_d} \cdot \Delta e. \tag{6}$$

We compute generalized gravity and then our desired torque commands as

$$\vec{\tau_d} = \vec{\tau_{rnea}} + \vec{\tau_{feedback}} - \vec{g}. \tag{7}$$

This Controller works as intended.

### 4.2.3 PDGravity

In this controller we rely on the gravity compensation implemented in Panda and the feedback torque from our state interface:

$$\vec{\tau_d} = -\vec{k_p} \cdot (\theta_m - \theta_r) - \vec{k_d} \cdot \Delta e. \tag{8}$$

### 4.2.4 PureGravity

This version of control is identical to the Gravity Controller and as such

$$\vec{\tau_d} = \vec{0}. \tag{9}$$

## 4.3 Cartesian Controller

This controller tries to control the robot with the use of the TSID library. It uses measured data and Pinocchio library functions to first compute forward kinematics and differential forward kinematics. Then it creates a configuration for TSID containing description of the given task for inverse dynamics from a user-defined configuration parameter dataset and current robot's model and URDF. This object computes the task space inverse dynamics problem to solve for $\vec{\tau_d}$.

This controller is based on a python example code of using the TSID library for Cartesian control of a robotic arm [16]. Sadly, during our work on this project we could not get this controller to work correctly.

## 4.4 Cartesian Impedance Controller

This controller attempts the same as previous Cartesian Controller, but does not make use of TSID library and its optimization functions. Instead, it is based on paper Linear Model Predictive Control in SE(3) for online trajectory planning in dynamic workspaces [17] and its demo code Cartesian Impedance [18].

After reading the current robot state and computing forward kinematics, we use Pinocchio function `updateFramePlacements()` to get updated robot frames. From this we compute quaternions for our rotation. We also find errors for end effector position and rotation and from this we compute the desired force on the EE, here called $F_{ee}$. Then we do

$$\vec{\tau}_{Coriolis} = C_M \cdot \dot{q}_m, \tag{10}$$

$$\vec{\tau}_{task} = J^T \cdot F_{ee}, \tag{11}$$

$$\vec{\tau}_{nullspace} = (I - J^T \cdot (J^T)^{-1}) \cdot (K_{NS} \cdot q_{null} - q_m) - 2\sqrt{K_{NS}} \cdot \dot{q}_m, \tag{12}$$

where $C_M$ is the Coriolis matrix found thanks to Pinocchio library. $K_{NS}$ is a Cartesian stiffness.

Finally, we get our final $\vec{\tau}$ as

$$\vec{\tau} = \vec{\tau}_{Coriolis} + \vec{\tau}_{task} + \vec{\tau}_{nullspace}. \tag{13}$$

This controller works as desired, being able to move in Cartesian coordinates. It allows to dynamically set the desired target position incl. rotation and its Cartesian stiffness.

### 4.5 Colab Controller

This controller is based on the demo of Kryštof Teissing called the Coworker Controller on the KUKA robotic arm [19]. We tried to implement similar logical structure on our Panda robotic arm in ROS2.

First, we compute the Jacobians with the help of Pinocchio library. Then we do the Moore-Penrose pseudoinverse

$$J^{-1} = (J \cdot J^T)^{-1} \tag{14}$$

and extract the left ($J^{-1} \cdot J$) and right ($J^T \cdot J^{-1}$) side Jacobian inverses. Next we get the Cartesian forces/moments as

$$\vec{F} = J^{-1}_{left} \cdot \vec{\tau_m}. \tag{15}$$

From this force $\vec{F}$ we get the twist as

$$\upsilon = \begin{pmatrix} F_0 \cdot K_p \\ 0 \\ F_2 \cdot (-K_p) \\ 0 \\ 0 \\ 0 \end{pmatrix}, \tag{16}$$

where $K_p$ is a constant parameter found using the formula

$$K_p = \frac{\upsilon_{max} \cdot T}{\tau_m ax}. \tag{17}$$

As can be seen, this parameter is found using the information about the robot's limits in velocity and torques. $T$ is the period with which the command is being send (in our case around 650 Hz).

Then we find $\dot{\theta}_r$ with

$$\dot{\theta}_r = J^{-1}_{right} \cdot \upsilon \tag{18}$$

From here we find the trajectory joint reference with simple calculation of joint positions and acceleration found using derivative and integral of our found $\dot{\theta}_r$. To compute final desired torque we once again make use of the Pinocchio library to compute `rnea()` and generalized gravity to find $\vec{\tau_d}$ using approach identical to IDControlSimplified.

This controller does not yet work as intended. Possible reasons are wrong usage of Pinocchio library and uploading the robot model.

### 4.6 Playback Controller

This controller is able to replay a learned trajectory of operator-guided manipulation. This controller makes use of a path recorded by `rosbag2`, transforming its contents into .csv file and then extracting said trajectory to use as requested $\theta_r$, $\dot{\theta}_r$ and $\ddot{\theta}_r$.

The Playback Controller uses a simple PD+ torque computation to find desired torques from

$$\vec{\tau_d} = \tau_{ff} - K_p \cdot (\theta_m - \theta_r) - K_d \cdot (\dot{\theta_m} - \dot{\theta}_r), \tag{19}$$

where $\tau_{ff}$ is the recorded torque. We then find $\ddot{\theta}_r$ over Pinocchio `rnea()` function, get the generalized gravity and finally get

$$\vec{\tau_d} = \vec{\tau_d} - \vec{g}. \tag{20}$$

This Controller works as intended.

# 5 Conclusion

In our work, we tried various approaches used for controlling real-life Franka Emika Panda robot. We placed our focus on working on real Panda robot as much as possible and we did not spend much time on software simulations such as Gazebo, Webots or PyBullet. We worked primarily on ROS2 Foxy distribution; there was newer distribution ROS2 Humble at time of working on this demo, but we preferred the Foxy installation since this is the only installation that works side-by-side with older ROS1 distribution Noetic. However, the differences between Foxy and Humble were only subtle with regards to implementing low-level controller. In the world of MoveIt2, the differences are more pronounced (and Humble version is better supported), but we did not focus primarily on MoveIt2 so this was not a big issue for us.

During our work on this project we have achieved many of our goals, but struggled with implementing some of the controllers and behaviors. Below is a table summarizing our general results with each library or control variant. Under `Install` are listed those we managed to successfully install in ROS2 Foxy. Under `Implemented` are the variants that were successfully used inside our codes. The `Test` category means that the libraries have been tested on real life Panda robotic arm.

| Name | Install | Implemented | Test |
|---|---|---|---|
| MoveIt2 | Yes | No | No |
| MoveIt Servo | Software Only | Did not use | Did not use |
| Gravity PD Controllers | Yes | Yes | Yes |
| Pinocchio - Joint Space Controllers | Yes | Yes | Yes |
| Cartesian Controllers Package [20] | Yes | No | No |
| TSID | Yes | Partial | No |
| Cartesian Impedance Controller | Yes | Yes | Yes |

The MoveIt2 platform offers high level control features like trajectory planning in collision space. However, we wished to use more low-level control approach instead and opted to not pursue this option. MoveIt Servo plugin employs a lower level of control than MoveIt2, since it can be driven by TwistStamped messages, but we could not get it to work on real life robot yet (only in pure software environment). The Gravity PD and Joint Space Controllers both worked well. These controllers were implemented under the Gravity, Snake and Playback Controllers. TSID was implemented under the Cartesian Controller, but due to time constraints we did not manage to get it working fully. The Cartesian Controllers Package is a library that works well with ros2_control. However, it only works for command interfaces of velocity and position, which at this point in time franka_ros2 does not give access to (only effort command interface exists). Finally, we implemted basic Cartesian control in the Cartesian Impedance Controller.

## 5.1 Future Work

The possible future work and expansion of this demo shall move in the following direction:

- Finish implementation of Cartesian control by using the TSID library.
- Test the whole stack under ROS2 Humble and make MoveIt2 Servo work.
- Finish Colab Controller.
- Polish and clean-up the code.

## References


[1] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[2] Arun Venkatadri. *ROS 1 vs ROS 2 What are the Biggest Differences?* 2023. URL: https://www.model-prime.com/blog/ros-1-vs-ros-2-what-are-the-biggest-differences.

[3] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. "Exploring the performance of ROS2". In: Oct. 2016, pp. 1–10. DOI: 10.1145/2968478.2968502.

[4] Franka Emika. *Libfranka Github page.* 2022. URL: https://github.com/frankaemika/libfranka.

[5] Franka Emika. *Franka ros2 Github page.* 2022. URL: https://github.com/frankaemika/franka_ros2.

[6] Open Robotics organization. *ROS2 releases.* 2023. URL: https://docs.ros.org/en/rolling/Releases.html.

[7] Justin Carpentier et al. "The Pinocchio C++ library – A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives". In: *IEEE International Symposium on System Integrations (SII).* 2019.

[8] Justin Carpentier, Florian Valenza, Nicolas Mansard, et al. *Pinocchio: fast forward and inverse dynamics for poly-articulated systems.* https://stack-of-tasks.github.io/pinocchio. 2015–2021.

[9] Nicolas Mansard Andrea Del Prete et al. "Implementing Torque Control with High-Ratio Gear Boxes and without Joint-Torque Sensors". In: *Int. Journal of Humanoid Robotics.* 2016, p. 1550044. URL: https://hal.archives-ouvertes.fr/hal-01136936/document.

[10] Ioan A. Sucan and Sachin Chitta. *MoveIt2 Github page.* 2023. URL: https://github.com/ros-planning/moveit2.

[11] Adam Pettinger. *Introducing MoveIt Servo in ROS 2.* 2020. URL: https://moveit.ros.org/moveit/ros2/servo/jog/2020/09/09/moveit2-servo.html.

[12] Open Robotics organization. *Control 2 ROS.* 2023. URL: https://control.ros.org/master/doc/ros2_control/doc/index.html.

[13] Denis Štogl. *Writing a new controller.* 2023. URL: https://control.ros.org/master/doc/ros2_controllers/doc/writing_new_controller.html.

[14] *Colab Panda package.* URL: https://gitlab.ciirc.cvut.cz/vajneluc/colab-panda.

[15] Mederic Fourmy. *panda_torque_mpc Package Github page.* 2023. URL: https://github.com/MedericFourmy/panda_torque_mpc.

[16] *TSID Manipulator.* URL: https://github.com/stack-of-tasks/tsid/blob/master/exercizes/tsid_manipulator.py.

[17] Nicolas Torres Alberto et al. "Linear Model Predictive Control in SE(3) for online trajectory planning in dynamic workspaces". working paper or preprint. Sept. 2022. URL: https://hal.science/hal-03790059.

[18] *franka_example_controllers package and its demo code Cartesian Impedance Example Controller.* URL: https://github.com/frankaemika/franka_ros/blob/develop/franka_example_controllers/src/cartesian_impedance_example_controller.cpp.

[19] Kryštof Teissing. *Coworker Controller Package Github page.* 2020. URL: https://gitlab.ciirc.cvut.cz/capek/coworker-controller.

[20] S. Scherzinger, A. Roennau, and R. Dillmann. "Forward Dynamics Compliance Control (FDCC): A new approach to cartesian compliance for robotic manipulators". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* 2017, pp. 4568–4575. DOI: 10.1109/IROS.2017.8206325.