1 Task 2: Trajectory Generation and Impedance Control

1.1 Learning Goals

- Master ROS
- Master Gazebo simulator
- Develop a trapezoidal velocity profile
- Create both circular and rectilinear trajectory generators
- Impedance control:
  - Change the robot's inertia matrix based on the tool
  - Track circular trajectories in free motion with performance analysis (desired vs. actual trajectory)
  - Evaluate system behavior with reduced stiffness for more compliant operation

1.2 Tools to Learn

- ROS
- Gazebo simulator
- Franka robot arm

2 Theoretical Background

2.1 Trajectory Planning Problem

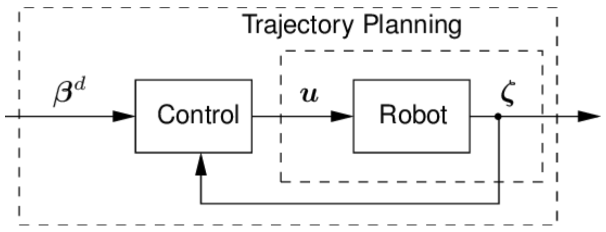


Figure 2.1: Trajectory planning in the closed loop control system

In order to execute a task of cleaning whiteboard we need to come with the algorithms that will generate us the movement of robotic arm. Generally speaking if we have two points we can create a line or an arc that will connect those points.

Having this in mind, the first family of algorithms that will be $point - to - point$ transitional functions. These functions will typically receive as an timestamp, or relative clock t from the start till the end of trajectory generation. Moreover, typically the program is specified by the constraints such as desired velocity q'_max and applied acceleration q'' which define the shape of the path and the type of movement that is planned (fast, slow, linear, etc.). Depending on the application those algorithms might be really complex, for such task as whiteboard cleaning we need the simplest linear/rotational movements.

2.2 Trajectory Profile - Trapezoidal velocity

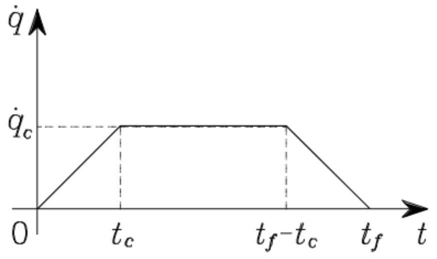


Figure 2.2: Trapezoidal velocity profile

The first task was to make the so called velocity profile. According to the Robotics: Modelling, Planning and Control [1] the easiest solution which enforces constant absolute value of q'' is **trapezoidal velocity profile**. The practicality of this solution comes from the real life hardware constraints, where the robot cannot immediately reach the maximum speed and needs to accelerate to it. Same goes for the deceleration. In the figure Figure 2.2 it is shown explicitly.

The formulae for finding the position using constant* acceleration can be described according to the [1] is:

$$q(t) = \begin{cases} q_i + \frac{1}{2}\ddot{q}_c t^2 & 0 \le t \le t_c \\ q_i + \ddot{q}_c t_c \left(t - \frac{t_c}{2}\right) & t_c \le t \le t_f - t_c \\ q_f - \frac{1}{2}\ddot{q}_c (t_f - t)^2 & t_f - t_c \le t \le t_f \end{cases}$$

where:

1. t is the current time, and t_c is time needed to reach maximum $\dot{q}$. Finally t_f is the overall time needed for executing the program.
2. q_i is the current position which we are updating, q_f is the final position
3. $\ddot{q}_c$ is the absolute value of the constant acceleration

Moreover, we simplify the model, by just considering one degree of freedom. This way we can later, add up multiple velocity profile for the same joint, if other coordinates would need to be changed, instead of just trying to handle multiple cases of different coordinates at the same place in the code.

2.3 Trajectory Generation in 3D

After having a mathematical model for the 1D velocity profile, we can think of how to generate a path of movement. Since, typically, user is defining its desired start and end position, these would be our parameters into the function to generate the point in 3D space. More formally we can say:

$$p = f([\sigma_{start}, \sigma_{end}])$$

where σ_{start} and σ_{end} are the start and end desired positions.

There are two primitive ideas to generate a path from one point to the other, which are

- Linear movement
- Circular movement

From these primitive movements we can potentially create more complex patterns of robot locomotion.

2.3.1 Rectilinear movement

Even though the textbook of rectilinear movement describes more sophisticated patterns that this type can create, our case is a simple linear path generation from a to b. This is why the formula applied is the same, the parameters can be strictly defined and are described below

$$\mathbf{p} = \mathbf{p}_i + s * \frac{\mathbf{p}_f - \mathbf{p}_i}{\left\| \mathbf{p}_f - \mathbf{p}_i \right\|}$$

Let us explore this formula. It essentially tells us which direction we will move by specifying the difference between current($\mathbf{p}_i$) and end points ($\mathbf{p}_f$). We divide this vector of direction with the norm and multiply by the rate of change s that we have of our trajectory. Since, the trajectory for the linear movement doesn't change such s can be a predefined as the difference between start and end position, resulting in:

$$\mathbf{p} = \mathbf{p}_0 + 2 * \frac{\mathbf{p}_f - \mathbf{p}_0}{\left\| \mathbf{p}_f - \mathbf{p}_i \right\|}$$

where $\mathbf{p}_0$ is the starting point of the trajectory. This way simplified the generalized solution for our purpose.

2.3.2 Circular path

When simulating circular motion, it is possible to interpolate the previous rectilinear trajectory. However, a much nicer solution is use the notion of unit circle and angles between the start and endpoint. Even though in the original textbook [1] there is a more expressive solution, we have decided to stick to the simpler formula.

$$\mathbf{p}(s) = \begin{pmatrix} \rho \cos\left(\frac{s}{\rho}\right) \\ \rho \sin\left(\frac{s}{\rho}\right) \\ 0 \end{pmatrix}$$

Essentially what we are doing is assigning to the x and y coordinates their position relative to the unit circle. To grasp all circles, the notion of radius is added ρ. Also x is defined though the cosine of the angle, same for the y, just we are using sine instead.

3 User Documentation

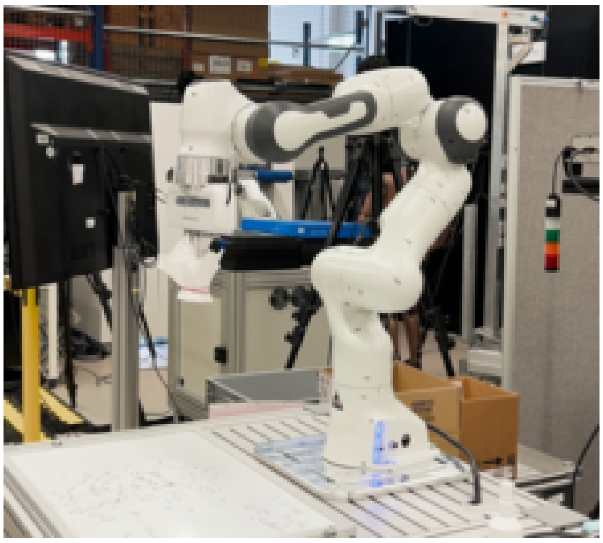


Figure 3.1: Franka Emika Panda Robot

In this chapter it will be explained how to build and execute the program with the real Franka Robot. Keep in mind that without physical source, you would need to address Developer Documentation to run it using simulator. You can see the example of Franka robot Figure 3.1

3.1 Installation and Usage

The following installation was partially taken from our supervisor ©Alessandro Melone alessandro.melone@tum.de

3.1.1 Cloning

First clone the repo to your local machine

```
git clone https://gitlab.lrz.de/tum-impl-ss24/assignment2-group1
git checkout main
```

3.1.2 Setup

To use properly the docker you need modify the `.env` file replacing

- `<username>` with your username (i.e., the output of `whoami` in your terminal) in `USERNAME=<username>`
- `<username>` with your username in `ROS_WS=/home/<username>/ros_ws`, (`ROS_WS` will be the path of your ROS2 workspace in the container)

  Reason: these variables are used to setup the container user with same credential of the host (by default the container would run as superuser). Among several benefits, this implies that all the files generated by the container have the same ownership of the one generated by the host. For further info, refer to this guide.

3.1.3 Usage

At the first usage, build the docker image using the following command in the project folder:

```
./setup_docker.bash
```

After, open a terminal in the project folder and run:

```
docker compose up
```

Now a docker container is running on your machine. To access the container, open another terminal and run:

```
docker compose exec -it gazebo_franka_ros1-ros1_node-1 bash
```

> this is the equivalent of opening a terminal on your Ubuntu installation but on the Ubuntu installation running in the container.

where `gazebo_franka_ros1-ros1_node-1` is the name of the container you want to access. To see the names of all the containers running on your machine, you can use `docker ps`.

3.1.4 First Usage and Build

Access a running container and use the following command to initialize your `catkin_ws`:

1. Inside container of **terminal1** run

   ```
   catkin_make
   ```

2. Open **terminal2** and run

   ```
   roslaunch trajectory_gen cartesian_impedance_example_controller.launch
   \\ robot:=panda robot_ip:=192.168.3.8
   ```

   This will connect your code from franka-ros to the remote Franka robot

3. Open new **terminal3** and remove other publishers, if there are any

   ```
   rosnode kill /interactive_marker
   ```

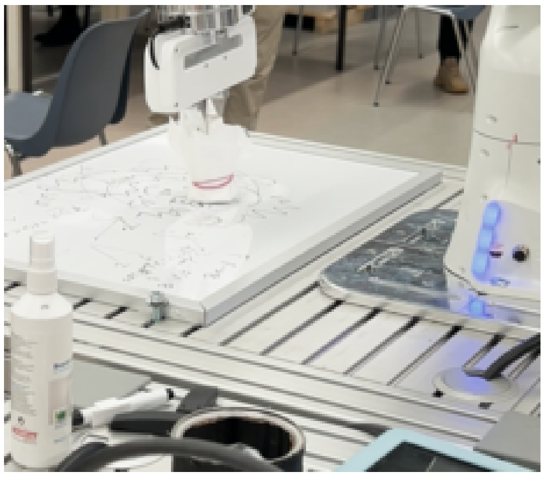
4. Go back to the **terminal1** and run

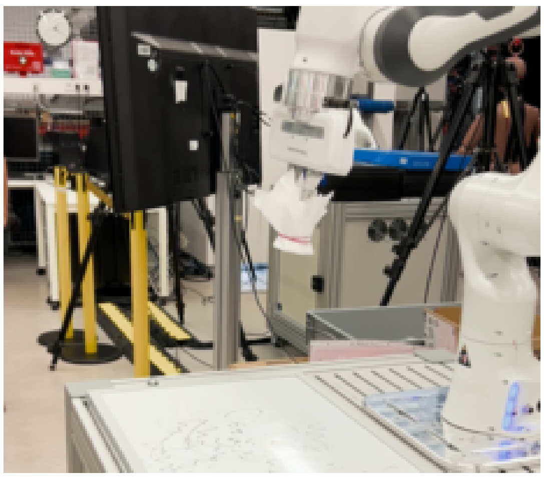
   ```
   rosrun trajectory_gen trajectory_gen_node
   ```

3.2 Operation Demo

Here are some images showcasing how the robot moves

(a) Linear movement towards the surface


(b) Rotational movement on the surface


(c) Going up to initial position

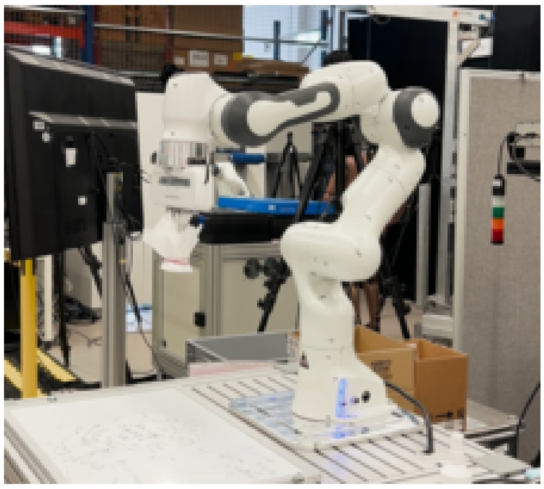

(d) Return to initial robot position

Figure 3.2: Different states of Franka movement during program execution

4 Developer Documentation

In this chapter it will be presented how the project is structured and what are the core principles and ideas behind it.

Some important notes:

1. The entiry file is `trajectory\_gen\_node.cpp` file
2. Trajetory generator files are `trajectory\_generator\_circular` (hpp and cpp) and `trajectory\_generator\linear` (hpp and cpp)
3. Project separates header and implementation, and also applies OOP principle such as polymorphism

4.1 Gazebo simulator

In order to check the behaviour of the trajectory generator we have to see the result by our own eyes. To do so the docker environment was already preinstalled with Gazebo simulator.

To run it use this command in the opened terminal inside container

```
roslaunch franka_gazebo panda.launch x:=-0.5 world:=$(rospack find demo_pkg)/world/pick_place
rviz:=false
```

4.2 Main program

As mentioned before, main program is located in the `trajectory\_gen\_node.cpp` and is connecting rosnode subscribers information to the trajectory classes as well as deciding on the next robots frame position.

Here is the full functionality of the main program

1. Subscribe to the "/franka_state_controller/franka_states" to retrieve the data about the initial position and current force.
2. Close gripper (grab the object) before starting main loop
3. Assign the initial state of the program. There are three states which are running though the main loop
   a) DOWN_LINEAR state - apply linear trajectory until reaching a force threshold

b) DOWN_CIRCULAR state - once reaching destination (desk) start making circular movements to clean the board

c) UP_LINEAR state - go to the initial state specified by the program, generally a good practice, not to have force disturbance in the beginning from the previous actions on the robot, and finishing complying to the standard of "pick and place" operation.

As a side note this implementation could be extended to so called "Behaviour tree", where each action can lead to multitude of action defined by the decision of the robots cognition.

4. Once state UP_LINEAR successfully transition into final state, finish the program

4.3 Trajectory generators and helper functions

4.3.1 VelocityProfile class

Velocity profile is one of the simpler classes, however the most important one since it will be used by the both trajectory generators.
Using knowledge from 2 this class takes as an input the following

- p_start - Start point of movement
- p_end - End point of movement
- q_dot_max - maximum speed the robot tries to reach
- q_double_dot_max - constant absolute value of acceleration

The start point and the end point provide the total distance of the movement. The maximal velocity and the constant acceleration together determine the time for both acceleration and deceleration periods

$$t_c = \frac{q_{dotmax}}{q_{doubledotmax}}$$

Since the total distance is already given, the distance for each period (acceleration, deceleration, or constant velocity) and the total movement time t_f can be calculated

$$t_f = 2t_c + (d_2 - q_{doubledot} * tc^2)/q_{dotmax}$$

4.3.2 Trajectory class

Trajectory class is a small interface which provides decent enough abstraction for its child classes, so that they could be used together in sequential and similar task assignments.

```
class Trajectory {
   public:
      virtual ~Trajectory() {}
      virtual void update(double dt, double force = 0) = 0;
      virtual bool isEnded() = 0;
      virtual geometry_msgs::Point getPoint() = 0;
      virtual std::string getName() = 0;
   };
```

4.3.3 Linear Trajectory class

Linear Trajectory is a child class inherited from Trajectory thus applying its methods. Linear trajectory algorithms are heavily inspired by the 2 theoretical background, by using relative positions, norms for the direction, etc..
Using knowledge from 2 this class takes as an input the following

- c - triple vector from initial robot position to the end effector (chosen frame). Provide the notion of relative movement
- normal_vector_hat - unit normal vector
- p_start - starting point of movement
- p_end - end point of movement
- q_dot_max - maximum speed of the robot
- q_double_dot_max - constant absolute value of acceleration
- force_threshold - force which is acceptable, until the threshold is reached

The c vector points from the origin of the base to the origin of the new coordination system. The unit normal vector stands for the orientation of the plane of the new coordination system (in the same direction of z axis). Furthermore, cross product of the unit normal vector and the c vector (after normalization) gives the unit vector y, cross product of y and unit normal vector yields unit vector z. Subsequently, all these vectors x, y, z build the rotation matrix, which with the c vector are able to transform the coordinates in the new CoS back to the base coordination system. This method offers us applicability even to the coordinate transformations.

In addition, start point and end point (in the new CoS) define the direction of movement, and where the linear movement starts and ends, while maximal velocity, and constant acceleration serve for the inputs of velocity profile function, in order to create trapezoidal velocity profile. Force threshold should be set by the user, and once the detected force exceeds this threshold, the robot will stop. This is useful for real cases when robot touches the surface of the table and stops to draw a circle on it. The current position value (double) from the velocity profile function times unit vector in the moving direction plus start point should give the current position in vector form.

4.3.4 Circular Trajectory class

Circular Trajectory is a child class inherited from Trajectory thus applying its methods. Circular trajectory algorithm is heavily inspired by the 2 theoretical background, by using relative positions to define the position according to the trigonometric rules of cosine and sine.

Using knowledge from 2 this class takes as an input the following

- c - triple vector from initial robot position to the end effector (chosen frame). Provide the notion of relative movement
- normal_vector_hat - unit normal vector
- radius - radius of the circle
- q_dot_max - maximum speed of the robot
- q_double_dot_max - constant absolute value of acceleration

For the circular trajectory function, inputs are c vector, unit normal vector, radius(double), maximal velocity(double), and constant acceleration(double). The c vector and unit normal vector are the same as in the linear trajectory function, a rotation matrix is obtained based on them and coordinate transformations can thus be realized, which has already been described in detail previously. The radius means the circumference of the circle, which is also the total path needs to be travelled. The movement starts from the origin of the new CoS, it first moves to the (radius, 0, 0), to perform the circle drawing. Similarly, the last two inputs: maximal velocity and constant acceleration together with 0 (for start point) and circumference (for end point) will be input to the velocity profile function to generate trapezoidal velocity. Using the current position value acquired from the function, the ratio of the travelled path to the radius can also be gained. Futhermore, this ratio is of great importance, which gives the current position in vector form in the new CoS: (cos(ratio) * radius, radius * sin(ratio), 0). Finally, the movement should stop if the travelled path is larger than or equal to the circumference(2 * Pi * radius).

5 Results

To present our results we would like to refer to two videos created as part of the assignment. The first 5.1 was done in the Gazebo environment and it performs smoothly and in expected manner. The same could be seen in the video made in the real life setting 5.2 by applying the same trajectory generators.

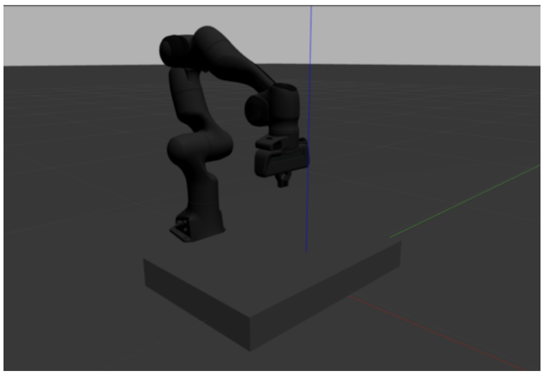


Figure 5.1: Simulation video (lrz link)

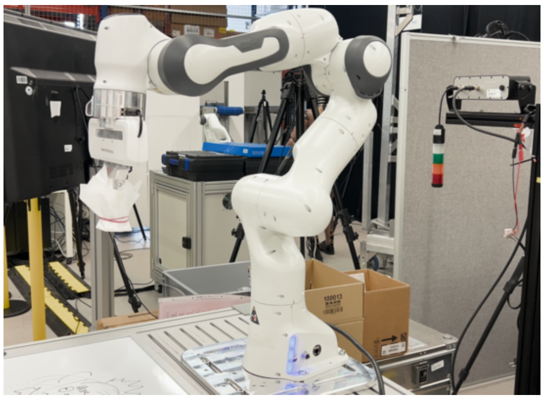
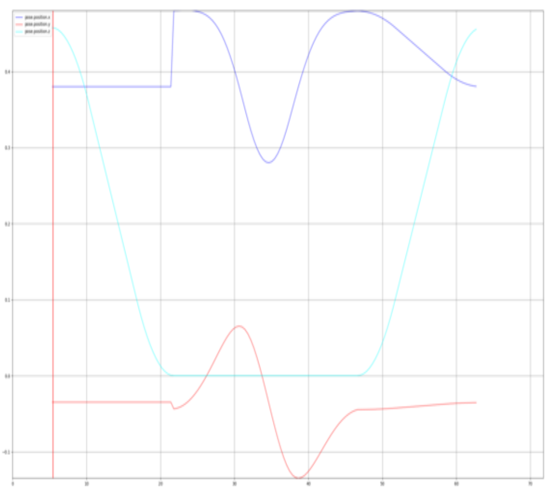


Figure 5.2: Linear movement towards the surface

Moreover, we have also conducted several experiments which involved changing impedance controller stiffness. It is interesting to notice how stiffness changes sensitivity of the controller.


(a) Position with linear stiffness = 10

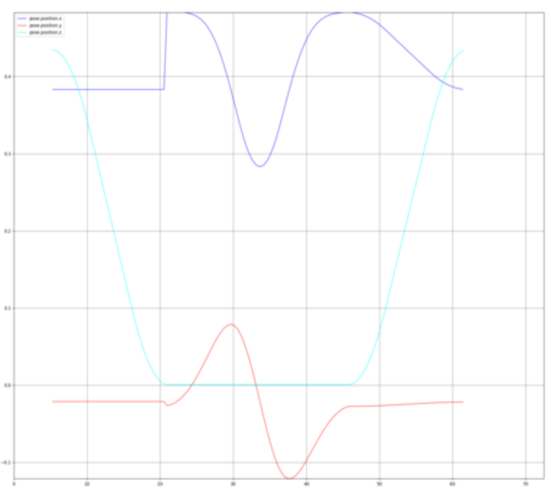
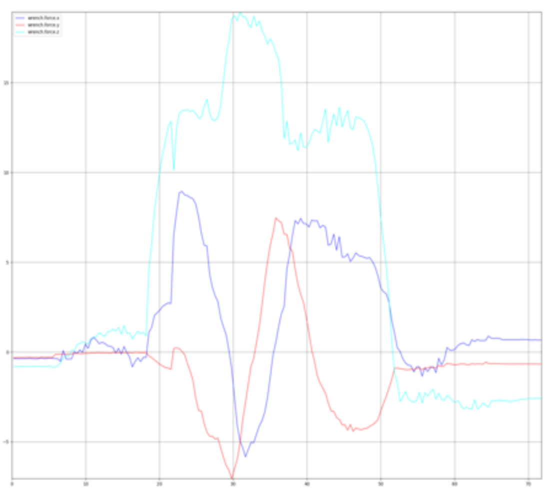

(b) Position with linear stiffness = 13

Figure 5.3: Position (x,y,z) changes through time

From the position change graph we can conclude that controller behaves little better with bigger stiffness, however, does not provide that much of the difference for the z coordinate which we are most interested in.


(a) Force change with linear stiffness = 10

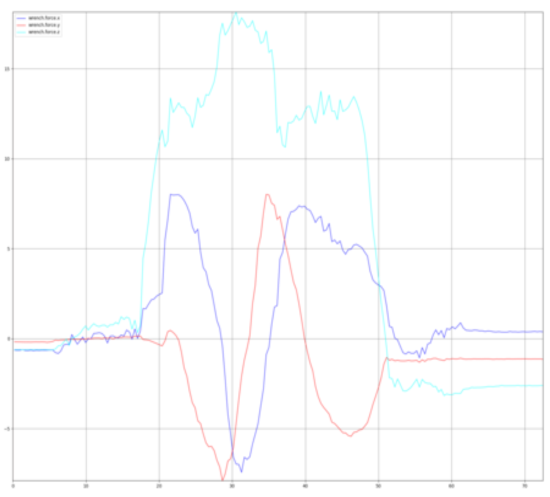

(b) Force change with linear stiffness = 13

Figure 5.4: Force changes through time

We can see the same relation in the force change graph. One last important thing is to notice how position change forms a trapezoid form, exactly as we have programmed it to do.