Executing a Trajectory with the Franka Robot in Gazebo

Petrilli Niccolò [145457], University of Siena

Abstract :

The objective of this project is to make the Franka robot execute a pre-defined trajectory in the Gazebo simulation environment. The project will involve creating the trajectory, executing it using ROS with a suitable controller, collecting data from the robot's execution, and analyzing the results by comparing the desired trajectory with the executed one.

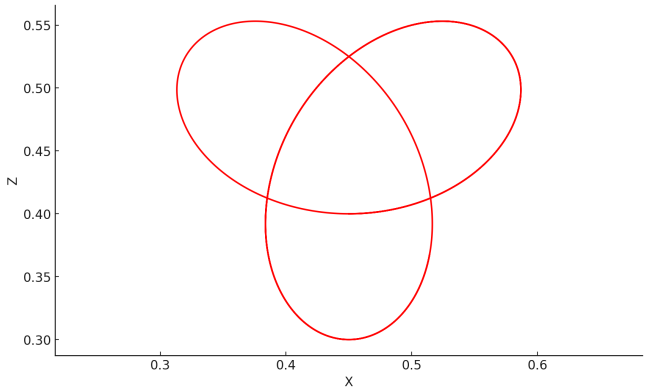

Figure 1. Trefoil Knot trajectory in XZ-Plane

1 Introduction

The aim of this project is the execution of a pre-defined trajectory in the Gazebo simulation environments. It requires to plan different steps to achieve the goal, i.e., write down the parametric equations of the trajectory in such a way that allow the robot to perform it, deciding whether to collect data before execution or calculate the trajectory with it. Then is important to choose a suitable controller offered by Franka_ros package, modifying it properly. After have executed the designed trajectory is important to use a right way of collection data independently from controller to compare those sent with those executed and for subsequent analysis.

The project is organized as follows: Section 2 describes the trajectory equations, Section 3 talks about ROS, the control algorithm and how it was handled. Section 4 describes the methodology used to collect the end-effector data, the graphical elements added, the joint data acquisition and the execution. Section 5 deals with the data analysis. Section 6 discusses the conclusions and possible improvements.

2 Trajectory Design

The required trajectory is the "Trefoil Knot" in XZ-Plane (Figure 1) which will be cycled by the controller. It can be defined as the closed curve obtained from the following parametric equations:

$$\begin{aligned} x(t) &= x_0 + f(sin(\omega t) + 2sin(2\omega t)) \\ y(t) &= y_0 \\ z(t) &= z_0 + fsin(3\omega t) \end{aligned} \tag{1}$$

In addition to the original equations, a point $p_0 = (x_0, y_0, z_0)$ has been added to translate the entire trajectory into a feasible region for the default configuration of the manipulator. Furthermore, a scaling factor f has been introduced to adapt the path to the operating ranges of the manipulator, and finally, there is the angular velocity ω to adjust the speed. The previous parameters were fixed before the trajectory execution through a process of trial and error as follows: $p_0 = (0.45, 0, 0.5)$, $f = 0.05$, $\omega = 0.2$. Therefore, the equations are parameterized by t, which changes over time. The trajectory was implemented inside a python script and executed at runtime as explained in the following sections.

3 Controller Implementation

3.1 ROS and Franka ROS overview

ROS (Robot Operating System) is an open-source framework used to develop robotic applications offering a set of tools and libraries to manage entire systems. The first fundamental unit of ROS is a "node", i.e, a code that can perform a series of operations to accomplish a specific task. Nodes can communicate with each other, mainly through the use of channels called "topics" by exchanging messages. Each topic is identified by a personal name, and nodes can publish (publisher) or subscribe (subscriber) to topics to send or receive messages.
Franka ROS is a library that allows to control the robotic arm "Franka Emika Panda" using ROS. It offers a series of packages to manipulate the robot at low level with specific interfaces, messages and topics.

3.2 Control algorithm

For this project it has been used a particular package named "Franka Gazebo", i.e, the only way to use the Panda robot in simulation through Gazebo and RViz software. This package offers a series of default controllers that can be used to move the robot. Among these are the cartesian impedance controller, the force controller, the joint position and velocity controller. Unfortunately these are the only controllers that can be executed in simulation despite the Franka ROS library providing other types that can only be used with the real robot. For this reason, the choice of controller is limited to those available in simulation.
After having carefully analyzed their structure, the choice fell on the use of the joint velocity controller that allows a direct and low control on the robotic arm's movements making them smooth and continuous. Unfortunately the default script only provides an example of how to send joint velocities to the manager without giving the possibility to execute a desired trajectory, for this reason, it is necessary to implement a control law that allows to translate the trajectory into joint's velocities and send them to the robot.

From the control's theory, is known that the differential kinematics equation represents a linear mapping between the joint velocity space and the operational velocity space through the usage of jacobian matrix:

$$v_e = \begin{bmatrix} \dot{p}_e \\ \dot{\omega}_e \end{bmatrix} = \dot{x}_e = J(q)\dot{q} \tag{2}$$

This fact suggests the possibility to utilize this equation to compute the inverse kinematics problem: assuming to have an assigned trajectory to the end-effector, in terms of its velocity, we can determine a trajectory in joint space $(q(t), \dot{q}(t))$ that reproduces the desired behavior:

$$\dot{q}(t_k) = J^{-1}(t_k)v_e(t_k) \tag{3}$$

The joint velocities are obtained by using the inverse of the Jacobian evaluated with the joint variables at the previous instant of

time. However this implementation is affect by drift phenomena such that the end-effector pose corresponding to the computed joint variables differs from the desired one. This problem could be overcome introducing in the solution the operational space error between the desired and the actual end-effector pose, defined as:

$$e(t_k) = x_d(t_k) - x_e(t_k) \tag{4}$$

which, considering the time derivative and the differential kinematics in (2), can be written as:

$$\dot{e} = \dot{x}_d - J(q)\dot{q} \tag{5}$$

For this equation to lead to an inverse kinematics algorithm, is necessary to choose a relationship between $\dot{q}$ and e such that (5) gives a differential equation describing error evolution over time. The choice is as follows:

$$\dot{q}(t_k) = J^{\dagger}(q(t_k))[\dot{x}_d(t_k) + Ke(t_k)] \tag{6}$$

where $\dot{x}_d$ contains the dynamics of the desired linear and angular velocity, the general expression is:

$$\dot{x}_d(t_k) = \frac{x_d(t_{k+1}) - x_d(t_k)}{Ts} \tag{7}$$

While linear velocity fully reflects this equation, to handle the quaternions expressing orientation it was necessary to convert the latter into a minimal notation such as Euler angles in terms of three variables ϕ_e allowing angular velocity to be easily expressed.

The matrix K is positive definite such that the system that describer the error's dynamic is asymptotically stable. The convergence rate of error depends on the eigenvalues of K matrix, the larger they are, the faster the convergence.

In order to use this methodology, some additions to the default script were necessary. First from "Franka hardware" package has been added the model of robotic arm to read the Jacobian matrix made available by the library, then the robot state was integrated to receive real-time information on the end-effector pose. These components were subsequently started correctly in the "init" and "starting" methods of the controller. All this allowed to be able to implement the previous equations inside the "update" method. A first test involving only the error on the pose allowed to verify the correctness of the method, even though a small initial drift was observed due to simulation.

3.3 Control hierarchy

In the previous section it was explained how the ROS system works, for this reason it was chosen to continue with a control according to the publisher/subscriber communication. The first step was to build a specific script called "controller_official.py" (node) to send directly the trajectory to the joint velocity controller. This is a real time approach and it allows to compute the trajectory during the controller execution without pre-saving data, avoiding further loading and possible incorrect readings. This node publishes data on the topic called "/trajectory_loader" in PoseStamped message format. This particular topic is created by the controller and it is subscribed on that to receive messages as soon as they are published. The hierarchy is showed in Figure (2).

4 Execution and and Collection

4.1 Methodology for collecting the end-effector pose

To check whether the robotic arm is executing the desired trajectory, it is necessary to analytically acquire the pose of the

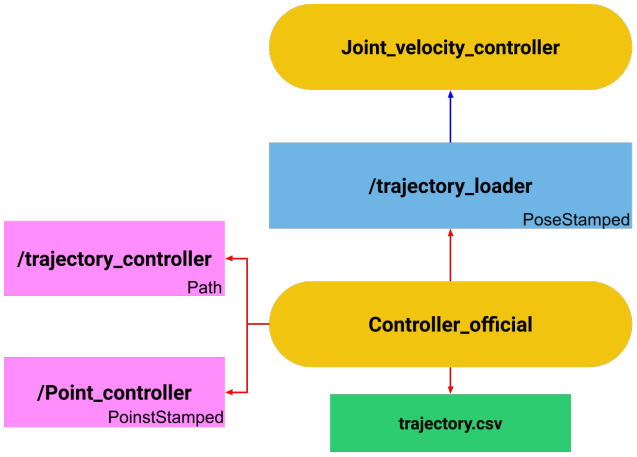

Figure 2. The node "Controller_official" publishes the computed trajectory on "/trajectory_loader" topic in PoseStamped message format, at the same time it collects data on "trajectory.csv" file that is saved after the execution. The joint_velocity_controller is subscribed to the topic in order to receive the pose for end-effector that it has to execute. Finally it publishes information on some topics (pink boxes) for the graphic part.

end-effector. For a correct analysis the method must be independent of the controller and from the data that it acquires. A first possibility is to read the data provided by the "tf" package. The package allows to keep track of different coordinate frames of the links, maintaining the relative relationships in a tree structure over time. In this way it is possible to trace the frame of the end-effector and recover its pose. The package also offers instant frame viewing via a plugin called "TF" in the RViz software, allowing the user to view different poses.

Despite the existence of the previous approach, within the scope of this project, it was decided to carry out a low-level end-effector pose acquisition, applying a direct kinematics algorithm. The goal of this approach is to determine the end-effector's pose respect to the base frame through a series of local transformation knowing the joint configuration, i.e, it consists to compute $T^b_{ee}(q)$. This total transformation can be expressed by the product of the homogeneous transformation matrices $T^{j-1}_j(q_j)$, where each represents the transformation of the j-th link with respect to the j-1-th link as a function of the single joint variable:

$$T^b_{ee}(\mathbf{q}) = T^b_0 \prod_{j=1}^{n} T^{j-1}_j(q_j) T^n_{ee} \tag{8}$$

In this context, the Denavit-Hartenberg (DH) parameters are used to define each local transformation. These parameters consist of four elements $(a_j, \alpha_j, d_j, \theta_j)$ providing a standardized way to describe the position and orientation between two consecutive joints. The parameters in question are made available in Franka's documentation and are shown in Table 1 with the addition of the transformation of the gripper to the robot flange.

Note also that the DH parameters are given respect the Craig convention, so it is necessary to use the transformation matrix given in (9) unlike the one most commonly known.

$$T^{j-1}_j = \begin{bmatrix} \cos(q_j) & -\sin(q_j) & 0 & a_j \\ \sin(q_j)\cos(\alpha_j) & \cos(q_j)\cos(\alpha_j) & -\sin(\alpha_j) & -\sin(\alpha_j)d_j \\ \sin(q_j)\sin(\alpha_j) & \cos(q_j)\sin(\alpha_j) & \cos(\alpha_j) & \cos(\alpha_j)d_j \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

Table 1
The Denavit–Hartenberg parameters for the Franka panda kinematic chain derived following Craig's convention with the addition of gripper.

a	d	α	θ
0	0.333	0	θ_0
0	0	$-\frac{\pi}{2}$	θ_1
0	0.316	$\frac{\pi}{2}$	θ_2
0.0825	0	$\frac{\pi}{2}$	θ_3
-0.0825	0.384	$-\frac{\pi}{2}$	θ_4
0	0	$\frac{\pi}{2}$	θ_5
0.088	0	$\frac{\pi}{2}$	θ_6
0	0.107	0	0
0	0.1034	0	$-\frac{\pi}{4}$

Respecting the idea of the project this algorithm has been implemented inside the node "dk_ee.py" which is subscribed to the topic "/joint_states" retrieving their position, the estimated end-effector pose is published on a new topic called "/estimation-ee". Acquisition hierarchy is shown in the Figure (3).

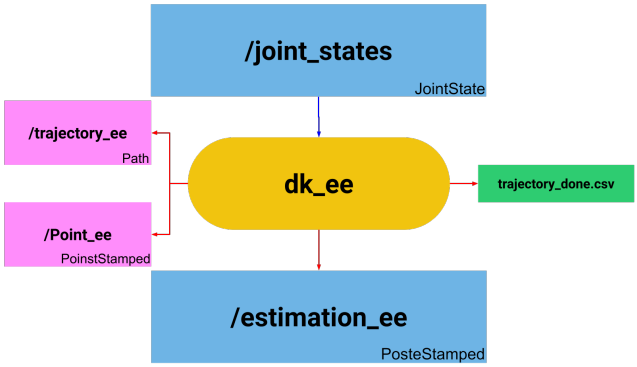

Figure 3. The node "dk_ee" that gets the information from the joints topic, publishes the results in "/estimation_ee" and on the trajectory_done.csv file and finally on the two topics to manage interface on RViz (pink).

4.2 Graphic interface in RViz

In order to be able to verify the results during the simulation execution, some graphic elements have been added. This is possible because RViz offers plugins to add elements that read the data from some topics.
Since that to compare the trajectories, the controller node publishes in real time on the topic "/trajectory_controller", the trajectory that it is sending. On the other hand, the node that estimates the position of the end-effector traces the actual trajectory by publishing it in the topic "trajectory_ee". For both trajectories, two spheres representing the desired pose and the pose performed over time were also added to check whether the second one follows the first one. This allows to compare the data immediately and to be able to intervene promptly during the execution. It is possible to see the result in Figure (4).

4.3 Joint status data collection

Rosbag files are a particular format used in ROS to record data present in topics in a sequence of messages. For this reason, they represent a good solution to capture information on the position and velocity of the joints during the execution of the robotic arm. Subsequently, a "DataReader.m" script was written to read and analyze the data offline. Both information are available in the topic "/joint_states".

4.4 Execution phase

To run the project, it is necessary to launch the "panda.launch" file of the Franka Gazebo package. In order to synchronize the startup, control and data acquisition phases, the "robot.launch" file (which starts the nodes in the system) has been modified by adding those created for this project and the recording of a rosbag file for the other data is started. These phases follow each other with a small delay to ensure perfect synchronization. The additional advantage is the possibility of being able to finish and save the data, all at once by stopping the simulation.
Once the simulation is started, it is possible to view the behavior of the arm in real time in the RViz software, as can be seen in Figure (4). Initially the end-effector is in a predefined equilibrium pose, when the controller starts sending the pose, the EE tries to follow the desired trajectory.

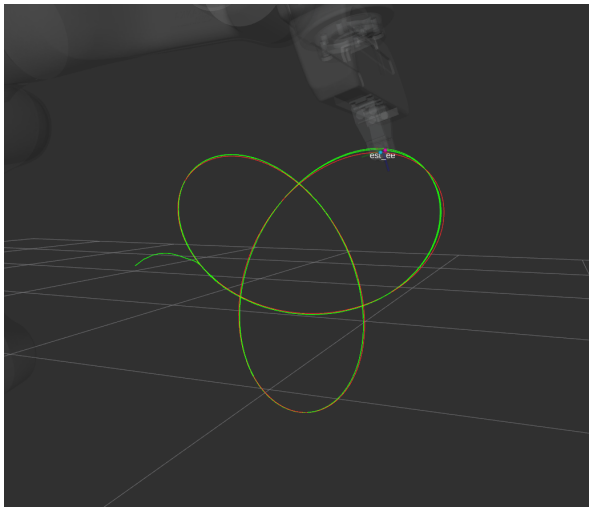

Figure 4. The figure shows the simulation, where the robotic arm executes (green) the desired trajectory (red), the behavior is visible in real time thanks to the graphic elements introduced in RViz.

5 Data Analysys

After the simulation is run, data analysis is required. To do this, a MATLAB script was created to read the pose of the end-effector sent and executed, extracting the coordinates and transforming the orientation from the quaternion to Euler angles.

Data from two experiments were analyzed, the first using only the equations presented in Section 2 and the second adding a rotation to verify correct tracking even with a desired orientation.

5.1 First experiment

Figure (5) shows a trefoil knot in the XZ plane, correctly reflecting the requested trajectory. The executed curve (blue) is very close to the requested one (red), meaning that the robot has executed the trajectory with good global accuracy. In the initial portion, a slight difference can be noticed due to the fact that the initial pose of the robot is not on the desired trajectory. In fact, the robot makes a movement to start following it, the convergence speed can be varied by changing the eigenvalues of the K matrix in equation (6).
In Figure (6) the components over time are shown, also in this case the overlap can be noted for all three components. The RPY trends remain very stable, with minimal variations. The joint positions shown in Figure (7) are stable and show regular oscillation with smooth control. It can be seen that only joints 2, 4, 6 take part in the movement while the remaining ones maintain a constant position,

performing a simple overall motion. The velocities show regular and smooth oscillations with low noise.

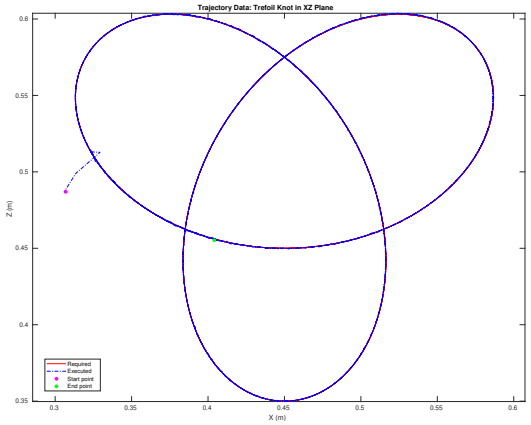

Figure 5. Trefoil Know trajectory in XZ plane in simulation

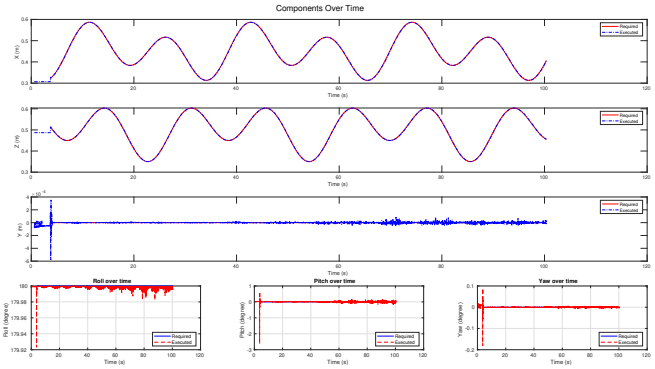

Figure 6. Position and orientation components over time

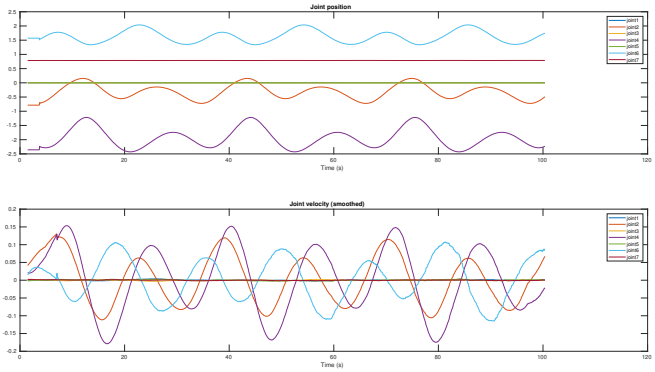

Figure 7. Joint positions and velocities data over time

5.2 Second experiment

In this case, a desired orientation was added in such a way as not to compromise the execution of the original trajectory, but allowing the verification of the orientation tracking. The choice fell on the XYZ rotation of Euler angles, with a 180-degree rotation relative to the X axis and applying a sinusoidal motion on the yaw (local Z axis) in order to engage joint 7 more. From the results obtained, it is noted that the trajectory in the XZ plane turns out to be very similar to the result obtained in the previous case as in Figure 5. From the new data shown in Figure (8)-(9) it can be seen that the X and Z components are stable with a trend similar to the previous one, while the Y component has a slightly higher deviation, but without significant changes. For the rotation components we can see that the yaw follows the desired sinusoidal motion while the remaining components maintain a stable behavior without major deviations.

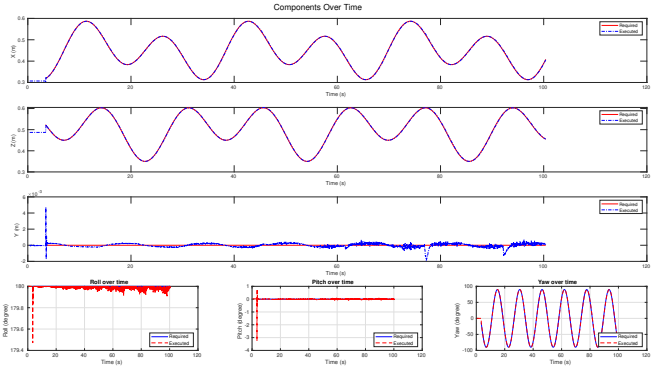

Figure 8. Position and orientation components over time with orientation

As for the joints, less regular positions can be observed with an initial transient phase before the movement begins to stabilize. In this case, it is clearly seen that several joints are activated and working simultaneously, with joint 7 which is the one that must handle more the orientation. Consequently, higher and less uniform speeds are observed due to the need to manage the most complicated pose.

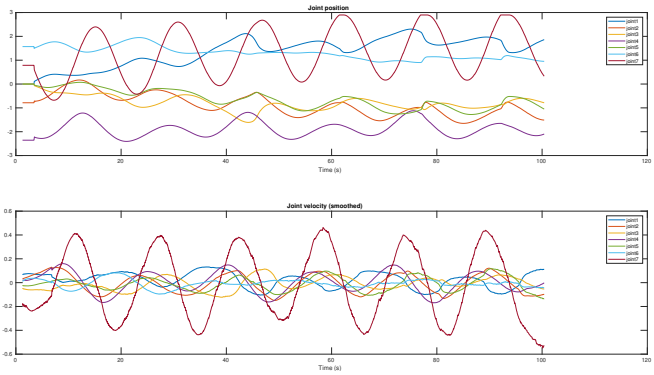

Figure 9. Joint positions and velocities data over time with orientation

In conclusion, to ensure a level of tracking similar to the previous case with the desired orientation, the robot must activate its joints more, with higher speeds, larger and less fluid movements, requiring more effort and resulting in a more complex overall motion.

6 Conclusion

The project successfully demonstrated the execution of a predefined trajectory using the Franka robot in Gazebo, achieving satisfactory accuracy in tracking both trajectory and orientation. The joint velocity controller performed smoothly, with only minor initial drift that was compensated during execution.

Key challenges included selecting an appropriate controller for the simulation, and managing the end-effector's orientation, which required careful tuning of the K matrix for stability.

Future improvements could involve advanced control algorithms, such as adaptive control, to reduce drift and improve rotation handling. Enhancing synchronization in simulation execution would also facilitate future deployment on a real robot.