IDIAP COMMUNICATION REPORT

PLANNING AND CONTROL OF ROBOT MANIPULATION TASKS

Jérémy Maceiras

Idiap-Com-01-2022

JULY 2022

Centre du Parc, Centre du Parc, Rue Marconi 19, CH - 1920 Martigny
T +41 27 721 77 11 F +41 27 721 77 12 info@idiap.ch www.idiap.ch

IDIAP RESEARCH INSTITUTE

UniDistance.ch

Jérémy Maceiras

Planning and control of robot manipulation tasks

Master Thesis

Idiap Research Institute, Martigny, Switzerland
Distance university of Switzerland

Supervision

Project supervisor: Dr. Sylvain Calinon
Company supervisor: Philip Abbet

June 2020
Copyright (c) 2020 Idiap Research Institute, Martigny - Switzerland,
https://www.idiap.ch/

Student ID: 5-878-895

Preface

> Science does not have a moral dimension. It is like a knife. If you give it to a surgeon or a murderer, each will use it differently
>
> **Wernher von Braun**
> *Aerospace engineer, father of the Apollo program*

This work has been a great personal investment for me, and it is with the greatest pleasure that I submit this report which concludes a great adventure.

This project has been possible by the support of the Idiap Research Institute through their master's degree in artificial intelligence program, and has been performed with the robot learning and interaction group directed by Dr. Sylvain Calinon.

For all questions: maceiras.jeremy@gmail.com

Acknowledgements

All my gratitude goes to the Idiap Research institute for choosing me to be part of this incredible journey that was this master's degree in artificial intelligence. More notably, I want to thank Olivier Bornet, head of the research and development team, and principal manager of the Master AI, whos has always taken care of the well being of students in this master program.

My special thanks go to Dr. Sylvain Calinon, head of the robot learning and interaction group, and supervisor of my project, for his advice and trust all along with this project. He shared his passion for robotics with me and made me work in an always motivating environment. I also want to thank all members of the robot learning and interaction group, in particular Teguh Santoso Lembono, and Emmanuel Pignat, for their advice all along with this project.

I also want to thank all the professors and assistants who followed and instructed me during this master. I also thank Philip Abbet, the company supervisor of my project, for his disponibility.

Other special thanks go to Guillaume Clivaz, research and development engineer, and laboratory roommate, for his advice during the development part of my project.

Finally, my thanks go to all the people around me that have helped me in one way or another during this master. Dear classmates, friends, family, things would have been more difficult without your support!

Contents

Abstract **ix**

Nomenclature **x**

1 Introduction **1**

2 Literature review **4**

2.1 Notions of robotics 4

2.1.1 End-effector representations 4

2.1.2 Robot kinematics 9

2.1.3 Robot dynamics 13

2.2 Linear Quadratic Tracking 15

2.2.1 Batch formulation 16

2.2.2 Dynamic programming formulation 17

2.2.3 Constrained LQT (Quadratic programming solution) 18

2.2.4 Performing LQT in a model predictive control way (MPC) . . 20

2.2.5 Applications of LQT 22

2.3 Riemannian manifold 25

2.3.1 Motivation 25

2.3.2 Definition of $\mathbf{S^d}$ manifold 25

2.3.3 LQT with $\mathbf{S^d}$ manifold 27

3 Proposed approach **32**

3.1 Planning approach 33

3.1.1 Position planning 34

3.1.2 Orientation planning 35

3.2 Tracking approach 37

3.2.1 Position tracking 37

3.2.2 Orientation tracking 39

3.2.3 Merge position and orientation control commands 39

4 Experiments **41**

4.1 Simulator developments 41

4.1.1 The simulator 41

4.1.2 The Python module 42

4.1.3 Developments . . . 45
4.1.4 Conclusion of the simulator experiments . . . 53
4.2 Real example . . . 54
4.2.1 Specifications and first analysis . . . 54
4.2.2 Mechanical developments . . . 56
4.2.3 Temperature sensor development . . . 61
4.2.4 Robotics developments . . . 63
4.2.5 Ongoing developments . . . 71

5 Conclusion **72**
5.1 Discussion . . . 72
5.2 Further works . . . 73
5.3 Personal conclusion . . . 73

A Quaternion algebra **75**
A.1 Addition . . . 75
A.2 Identity quaternion . . . 75
A.3 Conjugate . . . 75
A.4 Multiplication . . . 75
A.5 Inverse . . . 76
A.6 Rotation matrix to quaternion transformation . . . 76

B Proof of QP formulation for LQT **77**

List of Figures

1.1 Example of a task where a robot uses a pre-recorded motion to put caps on top of plastic bottles. As long as the task does not change (i.e., the same models of bottles and caps are used), the robot satisfies the job correctly. Once the mission differs from the original (i.e., if we want to produce bigger bottles), the robot can not fulfill the task, and a problem occurs. 1

2

1.3 Example of task space and joint space representation of a robot. . . . 3

1.4 Example of the nullspace of a task where two joint configurations lead to the same end-effector position. 3

2.1 Representation of the lack of measurability of Euler angles. The computed error path is not the shortest. It is because when calculating the error, we do not take into account that an angle of 0° is the same as an angle of 360° (periodicity of the angle). 5

2.2 Representation of the end-effector pose with respect to the base frame. $\mathbf{p}$ shows the end-effector position and $\{\mathbf{e}_x, \mathbf{e}_y, \mathbf{e}_z\}$ represent the orientation with respect to the base frame. 6

2.3 Representation of the gimbal lock: two gimbals are in the same plane, which provokes the loss of one degree of freedom. 7

2.4 Section view of a three-dimensional sphere, where the two red crosses represent two quaternions. You can see that using the Euclidean distance to measure the error between two quaternions is wrong since the resulting vector will not be part of the sphere. 8

2.5 Example of a 2D robot, d_i represents the distance between joint i and joint $i+1$, it is used to compute $\mathbf{D}_i$. d_0 and $\mathbf{R}_0$ are the position and orientation of the robot in the world, they are used to compute $\mathbf{H}_0^1$. . 10

2.6 Example of revolute and prismatic joints. Since for a revolute joint, the links of the robot are directly connected to a motor, we prefer to talk about torques. 13

2.7 Example of a reproduced end-effector motion made with LQT/MPC. The black curve corresponds to the reproduction. Color curves are partial results obtained if we use all control commands instead of only the first one. It shows that at the beginning of the horizon, it is not absurd to assume that the system matrices are constant. 21

2.8 Generated motion and velocities of the problem. Constraining the motion like this allows the system to correctly goes into the hole. This kind of constrained motion can also be used in a grasping task to ensure a correctly grasp of an object. 23
2.9 In this image, we use a constrained LQT to ask the two agents to meet in the middle of their motion. From this LQT, we can retrieve the corresponding motion for each agent (plotted in red and blue on the figure). The result of this LQT will be given to another regulator who would have to track the motion. 24
2.10 Visualization of the S^2 manifold (a 3d sphere), where we project a point $\mathbf{y}$ lying on the manifold into the tangent space of the point $\mathbf{p}$. . 26
2.11 The easiest example of parallel transport could be a human walking for a (very) long time in a fixed direction on earth (earth can be assumed to be spherical, thus it can act like a S^2 sphere). Sooner or later, he will come back to its starting point, but if we take a look at the evolution of its velocity vector during its journey, we notice that the vector changed all along the way (the vector is presented in green on the figure above), but from his point of view, his velocity never changed. 26
2.12 Example of a planning task on the S^2 manifold (simple integrator system). The tilted blue line corresponds to the solution in the tangent space of the initial point. The green point is a via-point in the middle of the trajectory, and the red cross is the desired final state. . 29
2.13 Same example as figure 2.12 with the LQT/MPC solution in red. As you can see, the LQT/MPC solution provides a better trajectory, which is optimal for the whole motion. 31

3.1 Example of a planning problem, where we know the initial and target state for the position and orientation, and a constraint. We want to retrieve the intermediary states. 33
3.2 Chosen approach to oversample an orientation trajectory. 36

4.1 DOM schema of a standard URDF file. A robot is composed of one ore more links and joints. 42
4.2 UML Class diagram of the LAL. 43
4.3 Sequence diagram which shows that an information about the robot state is computed only if the robot state has changed. 44
4.4 The left image represents the orientation of the end-effector at time step 0, and the right image the desired orientation at timestep T, since in this experiment, we only regulate the orientation of the end-effector, its final position may not be the same as the one in the right image. Basically, at the end of the motion, we want to have the end-effector of the robot pointing in the direction of the table. 46

4.5 The left image represents the desired final orientation of the end-effector frame, and right image represents the planning solution to the final orientation of the end-effector frame. As you can see, these two orientations are the same. This is a visual test to see if the planning worked as expected. 47
4.6 The left image represents the resulting final orientation of the tracking, as yon can see, the orientation of the end-effector is similar to the desired one presented in figure 4.4. To validate this first assumption, the right image shows the tracking error evolving over time, with an average error about `1e-4`, we can assume that the tracking performed as expected. 48
4.7 Tracking error over time for the operational space dynamics with an identity matrix solution. As you can see, this trick allows to keep an acceptable tracking error over time (even if it performs worst than the IK solution). By modifying $\mathbf{\Lambda}$ the control command does not respect the dynamics of the robot anymore, thus it adds a sawtooth perturbation to the error signal, that can leads to the instability of the regulator. 49
4.8 Result of the planning part, where a trajectory for each robot is generated. As you can see, the planner chooses a meeting point for the two robots with a small distance to avoid collision. 51
4.9 Left image shows the tracking error for the LQT/MPC approach, and right image shows the tracking error for the operational space dynamics approach. 52
4.10 Visualization of the setup for the project. 55
4.11 Plan of the Panda gripper physical connector. 56
4.12 Image of the fondue bowl, as you can see the handle is not straight, and it could lead to difficulties for correctly grasping it. 57
4.13 3D model of the left gripper. On the left image, you can see that the top part of the gripper aims to grab the fondue bowl, by marrying the shape of the handle. The bottom part of the gripper is a normal gripper to grasp casual objects. 57
4.14 Test with the 3D printed left gripper, it can correctly grasp and lift the bowl. 58
4.15 3D model of the right gripper. This gripper is made to grasp objects horizontally or vertically. 59
4.16 Test with the 3D printed right gripper, where it grasp the wine bottle 59
4.17 Test with the 3D printed right gripper, where it grasps the spatula. In this picture, you can also see the left robot maintaining the fondue bowl. 59
4.18 3D model of the spatula support. 60
4.19 3D printed spatula support with the spatula and the robot that is grasping it. 60
4.20 Schema on how the different parts of the sensor are connected together 61
4.21 Developed temperature sensor 62

4.22 Accuracy of the temperature sensor (average error: 0.63 °C). 62
4.23 Real setup inspired by figure 4.10 63
4.24 Schema representing the architecture of ROS. 63
4.25 Architecture of `libfranka`, source: `https://frankaemika.github.io/docs/libfranka.html`. 64
4.26 Minimalist class diagram of `libfranka`. 65
4.27 Tracking error on a real application. 68
4.28 Developed architecture for the FondueBot project. It is not specified in the diagram, but all classes use Eigen (a linear algebra library) as mathematical library. 69
4.29 Sequence diagram representing the execution of a task 70
4.30 Structure of the constraints of a task. 70
4.31 State machine of the FondueBot project. 71

Abstract

Nowadays, robots are used in a large number of industrial processes. Their use speeds up the fabrication process and drastically decreases the cost of products. These industrial robots are often used with pre-recorded motions, that allow them to perform a task quickly, but adapting these robots for a new task is time-consuming. It is done by recording another motion that respects the new task. If the task changes again, the same time-consuming approach needs to be repeated, even if the task differs only a bit from the programmed one.

The purpose of this project is to solve this problem by allowing robots to understand the goal of their task and not actions that lead to it. To achieve this purpose, the developed approach consists in finding an optimal trajectory that fulfills the task and makes a robot tracks it.

In this project, a task is fully represented by the final position and orientation of a robot end-effector, and by constraints that occur during the motion. These constraints aim to specify a way to achieve the task for a robot. They are particularly useful if we do not want a robot to go through a given position or want to specify the approach to the task.

From this definition of a task, the developed methods, will find a task space trajectory that respects the task, and make a robot tracks it with torque or joint velocities commands.

Keywords: position and orientation planning, position and orientation tracking, operational space control, Linear Quadratic Tracking (LQT), quaternions control, spherical manifold

Nomenclature

Symbols

ϵ	Quaternion in a complex form
$\boldsymbol{\epsilon}$	Quaternion in a vector form
$\mathbf{b}$	Vector
$\mathbf{A}$	Matrix
$\boldsymbol{\tau}$	Torque vector
$\mathbf{w}$	Angular speed
$\boldsymbol{\mathcal{S}}_t$	Desired state of the system at time t
S^d	Spherical manifold of dimension d
$\mathcal{T}_{\mathbf{p}}\mathcal{M}$	Tangent space of point $\mathbf{p}$

Acronyms and Abbreviations

LQT	Linear Quadratic Tracking
LQR	Linear Quadratic Regulator
SDK	Software Development Kit
OS	Operational Space
IK	Inverse Kinematics
QP	Quadratic Programming
DP	Dynamic Programming
LQT/MPC	Linear Quadratic Tracking in a Model Predictive Control way
LAL	Library Abstraction Layer

Chapter 1

Introduction

In the modern world, the use of robots in the industry is entirely democratized. It is not surprising to see that our car [1], washing machine [2], computer [3], ... is built on an assembly line mainly composed of robots. These robots serve a unique task and perform it during their entire lifetime.
Often, these robots are programmed with pre-recorded motions, which allow them to do a unique task quickly and precisely, but they do not offer generalization and adaptivity abilities. As long the assignment of a robot does not change, it will perform well. Once a modification occurs in the task, the manufacturer has to reprogram it entirely to satisfy the modified task.

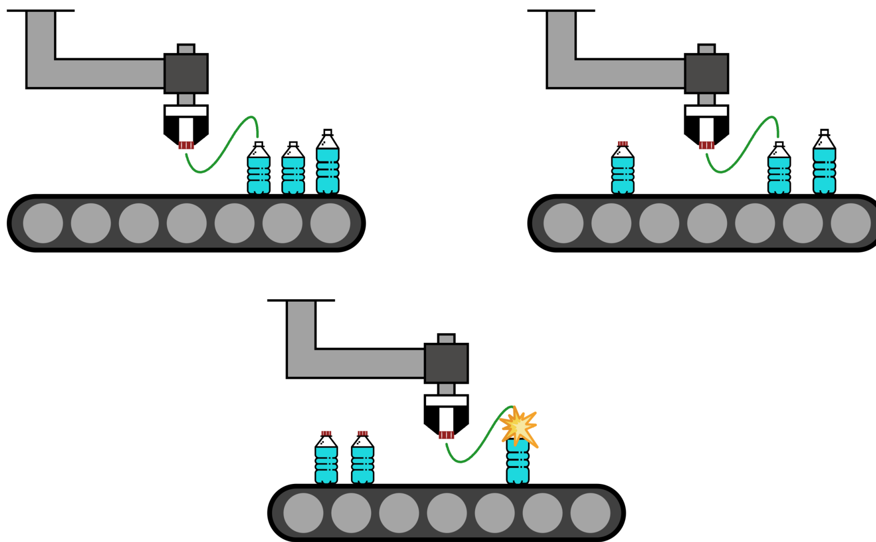


Figure 1.1: Example of a task where a robot uses a pre-recorded motion to put caps on top of plastic bottles. As long as the task does not change (i.e., the same models of bottles and caps are used), the robot satisfies the job correctly. Once the mission differs from the original (i.e., if we want to produce bigger bottles), the robot can not fulfill the task, and a problem occurs.

The purpose of this project is to solve this problem by specifying to a robot the goal

and not actions that lead to the goal. Actions are learned by the robot in function of the task and its constraints.
A constraint is a duty for the robot that occurs at a time t of the motion. In this context, a constraint can take two forms:

- Equality constraint: we want the state of the robot at a time t to be equal to another state. They can be useful to plan a meeting between two agents during the motion.
- Inequality constraint: we want the state of the robot at a time t to be greater or equal to a specific limit. They can be useful if we do not want the robot to be too close to a given object in the workspace.

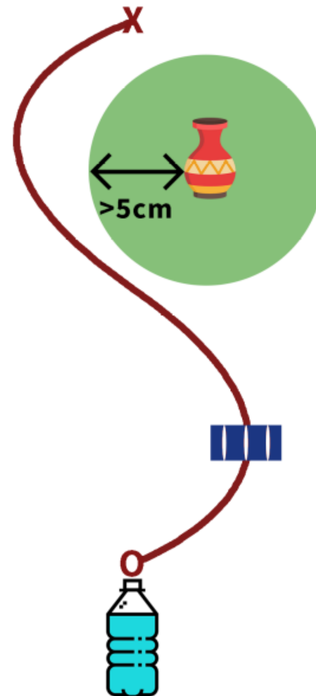


Figure 1.2: Example of a planned motion (in red) where an inequality constraint is set to keep a distance bigger than 5 centimeters between the robot and the vase (to avoid a contact between the robot and the object), an equality constraint is set to be sure that at a given time, the robot is at the position of the cap to grab it. In addition to the constraints, the robot has to reach the desired final state. In this case, at the end of the motion, the robot should be on top of the bottle in order to put the cap.

The states referred previously are a set of quantities, which is enough to represent a robot configuration at a time t. In robotics, states can be referenced in two possible spaces:

- The joint space represents a robot configuration with information about its joints (angular position and angular velocity of each joint).
- The task space represents a robot configuration with information about its end-effector (position and velocity).

For a user, it is more convenient to specify a task in the task space instead of the joint space. With this representation, we can directly determine the desired

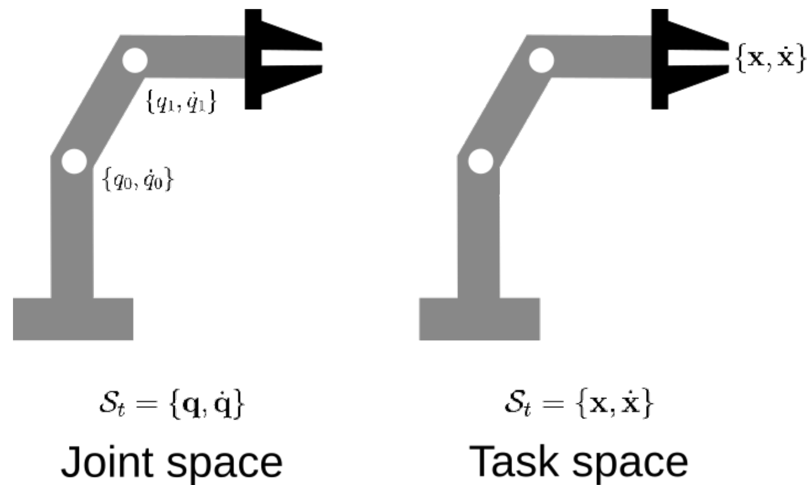


Figure 1.3: Example of task space and joint space representation of a robot.

position of the end-effector at each time step. The problem with this representation is that it does not correspond to a unique robot configuration, whereas the joint space representation yes. When we work with redundant robots, an end-effector position can be achieved with several joint positions. This information can be used to construct the nullspace of the task, which can be used to perform other tasks that do not perturb the execution of the main task.

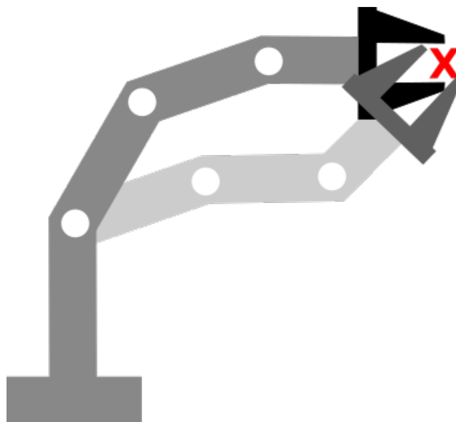


Figure 1.4: Example of the nullspace of a task where two joint configurations lead to the same end-effector position.

The developed approach solves this problem by breaking it into two steps, the planning, and the tracking. The goal of the planning is to determine a motion that satisfies the constraints and the final state. The motion resulting from this step will be given to the tracking part where the robot would have to track precisely the movement.

Chapter 2

Literature review

This section aims to introduce and present the different tools used in the project. These tools rely on the expertise of the laboratory and are widely used in the group. Firstly, some notions of robotics will be briefly presented to fully understand the next of the project. Afterward, the topics of planning and tracking of position and orientation will be explained.

2.1 Notions of robotics[1]

The position of a robot can be fully represented by its joints position. In practice, it is more interesting to represent the state of a robot in function of the position and orientation of its end-effector (figure 1.3). However, when we work with redundant robots (robots with more than six degrees of freedom, three translational and three rotational), the position and orientation of the end-effector can be achieved with different joints configuration (figure 1.4).

2.1.1 End-effector representations

Choosing a good representation for the end-effector has to respect some criterias, to facilitate their planning and tracking:

- **Uniqueness**: the representation of orientation and position should be unique, a representation $\mathbf{x}_i$ should not be the same as a representation $\mathbf{x}_j$ (e.g. an angle expressed in radians is not unique $0 = 2\pi = 4\pi$).
- **Measurability**: the error between two orientations $\mathbf{x}_i$ and $\mathbf{x}_j$, represented in the same way, should be measurable and calculable. Furthermore, the error could be expressed in $\mathbb{R}^3$, since the orientation of a robot is controlled in task space with an angular velocity, acceleration, or wrenches. With an error representation in $\mathbb{R}^3$, each element of this vector should represent a modification to perform on one axis to get the desired orientation.

[1] Everything presented in this section is a summary of the fundamental of robotics, more details can be found in [4]

- **Efficiency**: it should be easy to convert an information given under a certain representation to the other representation.

There is no problem to represent the position; we can use the three-dimensional world coordinates of the end-effector. It is unique, measurable, and efficient. This representation is already used natively in all robotics SDK, whereas representing the orientation accurately is a more challenging problem.

Orientation

Campa & de la Torre [5], provide an interesting survey about the different manners to represent the orientation of an end-effector. Here, only the Euler angles, rotation matrices, and unit quaternions, also called Euler parameters (the one used in the project), are presented.

Euler angles

It is the most natural orientation representation. It consists of a three-dimensional vector representing the amount of rotation around each of the three world axis:

$$\boldsymbol{\phi} = [\alpha, \beta, \gamma]^\top,$$

with α, β, γ representing the rotation around axis x, y, z respectively.
This representation is often present in all robotics SDK. It is efficient (as the representation is given, there is no need to transform it), but it lacks measurability and uniqueness.
Given two orientations represented in Euler angles $\boldsymbol{\phi}_d$ and $\boldsymbol{\phi}_t$, the error

$$\mathbf{e} = \boldsymbol{\phi}_d - \boldsymbol{\phi}_t,$$

is valid only for a local case where we assume that the two elements are relatively close.

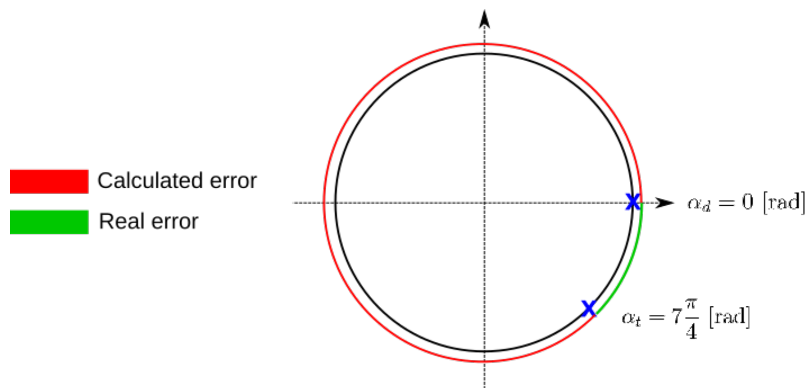


Figure 2.1: Representation of the lack of measurability of Euler angles. The computed error path is not the shortest. It is because when calculating the error, we do not take into account that an angle of 0° is the same as an angle of 360° (periodicity of the angle).

To overcome this error, we have to make a particular transformation before using these angles. For the same reason, the uniqueness of value is not guaranteed.
Due to these representation problems, most robotics SDK prefer to use the rotation matrix presented below.

Rotation matrices

A rotation matrix represents the end-effector orientation by expressing its frame in the world base frame. The three axes are stacked together to form a 3×3 matrix. It is also common to vectorize this matrix for a better representation.

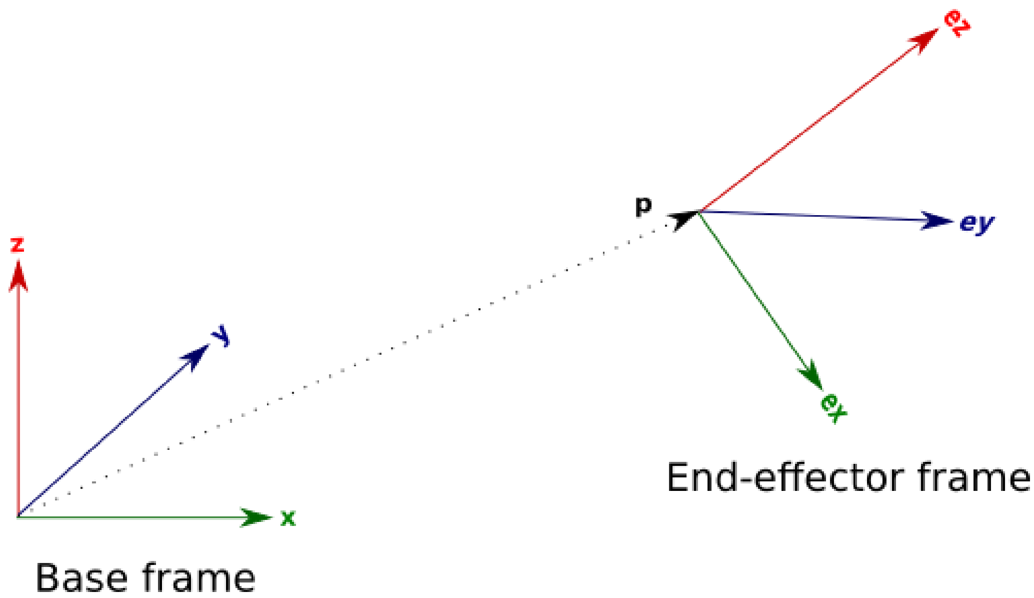


Figure 2.2: Representation of the end-effector pose with respect to the base frame. $\mathbf{p}$ shows the end-effector position and $\{\mathbf{e}_x, \mathbf{e}_y, \mathbf{e}_z\}$ represent the orientation with respect to the base frame.

$$\mathbf{R} = \begin{bmatrix} \mathbf{e}_x & \mathbf{e}_y & \mathbf{e}_z \end{bmatrix}$$

In robotics SDK, the rotation matrix is often accessible through the homogeneous transformation matrix (or pose matrix) of the end-effector, which combines the position and orientation in one matrix,

$$\mathbf{H}_{\mathbf{ee}} = \begin{bmatrix} \mathbf{R} & \mathbf{p} \\ \mathbf{0} & 1 \end{bmatrix}.$$

The error between two rotation matrices $\mathbf{R}_d$ and $\mathbf{R}_t$ is given by:

$$\widetilde{\mathbf{R}} = \mathbf{R}_d \mathbf{R}_t^\top,$$

where $\widetilde{\mathbf{R}} \in \mathbb{R}^{3\times3}$ is the error rotation matrix.

To get the error represented in $\mathbb{R}^3$, [6] proposed the following definition:

$$\mathbf{e} = \frac{1}{2}\begin{bmatrix} \widetilde{r}_{32} - \widetilde{r}_{23} \\ \widetilde{r}_{13} - \widetilde{r}_{31} \\ \widetilde{r}_{21} - \widetilde{r}_{12} \end{bmatrix},$$

where $\widetilde{r}_{ij}$ stands for the element in the i-th row and j-th column of $\widetilde{\mathbf{R}}$.
Rotation matrices are serious candidates to represent the orientation of the end-effector as they respect all the criteria presented above. But for this project, it has been chosen to use a representation based on unit quaternions. Section 2.3 will motivate this choice.

Unit quaternions

Quaternions can be described as higher-dimensional complex numbers. They are composed of a real and imaginary part:

$$\epsilon = \underbrace{\epsilon_0}_{\text{Real part}} + \overbrace{\epsilon_1 i + \epsilon_2 j + \epsilon_3 k}^{\text{Imaginary part}},$$

where i,j,k are imaginary units respecting:

$$\begin{cases} i^2 = j^2 = k^2 = -1, \\ ij = k, jk = i, ki = j, \\ ji = -k, kj = -i, ik = -j. \end{cases}$$

Quaternions are commonly used in computer science to represent spatial rotations, they are more compact and quicker to compute than representations by matrices, and unlike Euler angles, quaternions are not subject to "gimbal lock". This is why they are widely used in computer graphics.

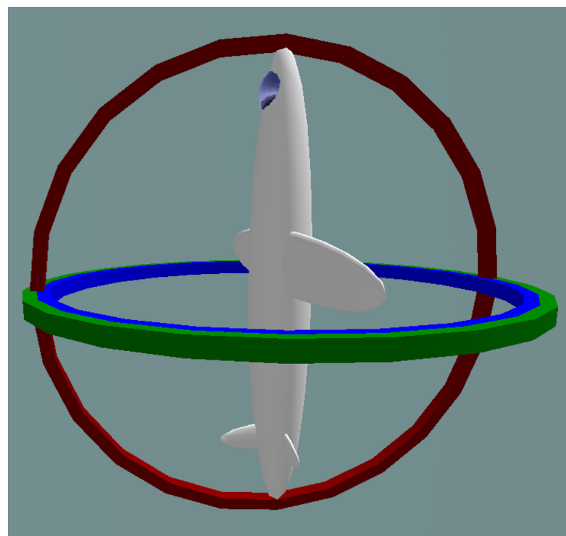


Figure 2.3: Representation of the gimbal lock: two gimbals are in the same plane, which provokes the loss of one degree of freedom.

To facilitate the notation, it is common to represent a quaternion as a vector $\boldsymbol{\epsilon} \in \mathbb{R}^4$, with

$$\boldsymbol{\epsilon} = [\epsilon_0, \epsilon_1, \epsilon_2, \epsilon_3]^\top.$$

A unit quaternion is a quaternion that satisfies a unit norm condition (i.e: $\boldsymbol{\epsilon}^\top \boldsymbol{\epsilon} = 1$). Thus, it can be seen as a point lying on the hypersphere $S^3 \subset \mathbb{R}^4$, where the real part mainly influences the angle of rotation, and the imaginary part mainly influences the axis of rotation:

$$\epsilon_0 = cos(\frac{\theta}{2}),\ \mathbb{I}\text{m}(\boldsymbol{\epsilon}) = sin(\frac{\theta}{2})\mathbf{u},$$

where θ is the angle of rotation, and $\mathbf{u}$ is the unit rotation axis.
A problem that occurs with a representation of orientation with quaternions is that the quaternions $-\epsilon$ and ϵ represent the same orientation.
Quaternions have their own algebra. Thus, we have to be prudent when we want to manipulate them. [7] provides a good introduction to quaternion algebra, where the main operations with their equivalence by considering a quaternion as a vector are listed in appendix A.
Since a quaternion is a point lying on the spherical manifold $S^3 \subset \mathbb{R}^4$, representing the error between ϵ_1 and ϵ_2 may be a difficult challenge for the human mind, mainly because it is difficult/impossible for a human to represent a four-dimensional sphere. This is why, for a better understanding, we may prefer to represent a quaternion as a point lying on the 3-dimensional unit sphere and adapt the algorithm to work on S^3.
The first idea that may come is to use the Euclidean distance between the two quaternions as error. Even if this solution may work if the two are relatively close, it is not accurate since the resulted error will not be part of the sphere.

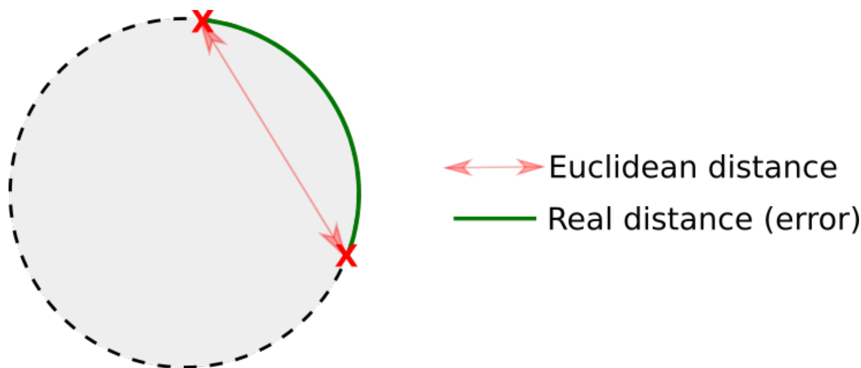


Figure 2.4: Section view of a three-dimensional sphere, where the two red crosses represent two quaternions. You can see that using the Euclidean distance to measure the error between two quaternions is wrong since the resulting vector will not be part of the sphere.

Instead of the Euclidean distance, we prefer to use the geodesic distance between two quaternions (presented in green on figure 2.4). Figure 2.4 also shows that if we bring closer the two points, the Euclidean distance will become more and more similar to the geodesic distance.

Measuring this geodesic distance is not trivial, and the topic will be presented in detail when the Riemannian manifold part will be presented. The manifold will present some transformation that we can apply to quaternions to make them unique, measurable, and efficient.

2.1.2 Robot kinematics

In the introduction of this thesis, the two most used state spaces in robotics were briefly presented: the joint space and the task space. Robot kinematics is a set of tools to connect these two spaces. We refer to forward kinematics, the task of transforming a joint space configuration into a task space configuration. Similarly, inverse kinematics is the action to transform a task space information into its corresponding joint space information. The transformation between these two spaces is fundamental. Since, in practice, we usually prefer to define a task in the task space, whereas it is more convenient to control a robot in joint space.

Forward kinematics

Given the joint position $\mathbf{q}$, and velocities $\dot{\mathbf{q}}$ in $\mathbb{R}^n$ (n is the total number of joints of the robot), we are interested into finding the corresponding end-effector position $\mathbf{x}$, and velocities $\dot{\mathbf{x}}$. Finding the end-effector position is quite straightforward, we multiply each forward transformation matrix of each joint and we end up with the homogeneous transformation matrix of the end-effector, which as mentioned above, informs us about the position and orientation of the end-effector (see the rotation matrix part of chapter 2.1.1).

$$\mathbf{H_{ee}} = \mathbf{H}_0^1 \mathbf{H}_1^2 \dots \mathbf{H}_{n-1}^n,$$

with

$$\mathbf{H}_i^{i+1} = \begin{bmatrix} \mathbf{R}_i & \mathbf{D}_i \\ \mathbf{0} & 1 \end{bmatrix},$$

where:

- $\mathbf{R}_i$, the rotation matrix of link i defined by the position of joint q_i.
- $\mathbf{D}_i$, the position of joint $i+1$ view from the frame of joint i (= view from the coordinate system of joint i).
- $\mathbf{H}_i^{i+1}$, the forward transformation matrix to go from frame i to frame $i+1$. Frame 0 is the base frame, since no joints are attached to it, it is constant and is used to know the position and orientation of the robot in the environment.

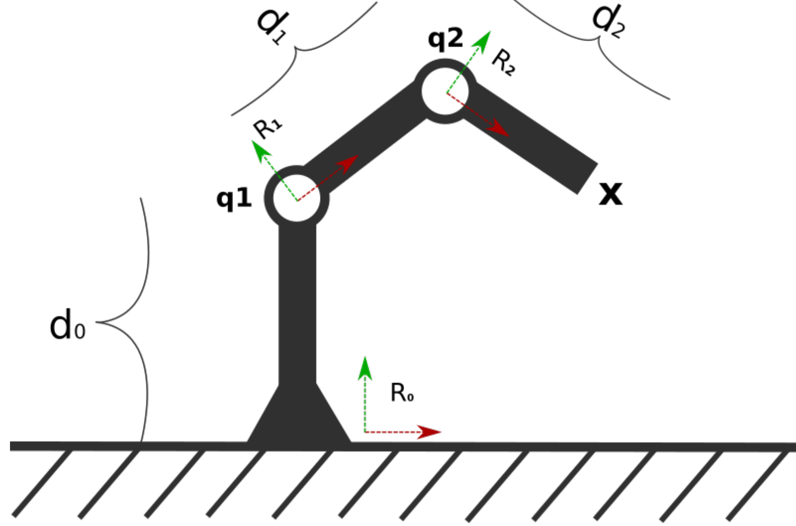


Figure 2.5: Example of a 2D robot, d_i represents the distance between joint i and joint $i+1$, it is used to compute $\mathbf{D}_i$. d_0 and $\mathbf{R}_0$ are the position and orientation of the robot in the world, they are used to compute $\mathbf{H}_0^1$.

This is how a joint state information $\mathbf{q} \in \mathbb{R}^n$ can be transformed to represent the position and orientation of the end-effector. The problem of finding the end-effector position and orientation corresponding to the joint positions of the robot can be seen as a mapping function f, mapping the joint space to the task space:

$$\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^6,$$

$$\mathbf{x} = \mathbf{f}(\mathbf{q}).$$

The task space is in $\mathbb{R}^6$ because, here, we assume that we use Euler angles to represent the orientation of the end-effector, even if as said before, it is not the best way to represent the orientation, their time-derivatives are probably the best way to describe a change in the orientation, and we are still missing a tool for transforming the joint velocities $\dot{\mathbf{q}} \in \mathbb{R}^n$ into the task space velocities $\dot{\mathbf{x}} \in \mathbb{R}^6$ (three translational velocities and three rotational velocities). Deriving the equation above with respect to time by using the chain rule, give us a way to link the first order derivatives of the two spaces together:

$$\frac{d}{dt}\mathbf{x} = \frac{d}{dt}\mathbf{f}(\mathbf{q}).$$

$$\dot{\mathbf{x}} = \frac{\partial \mathbf{f}}{\partial \mathbf{q}} \frac{\partial \mathbf{q}}{\partial t} \text{ (chain rule)},$$

where $\frac{\partial \mathbf{f}}{\partial \mathbf{q}}$ is the Jacobian matrix of the system, which aims to relate the first order partial derivatives of the input to the first order partial derivatives of the output and is defined as:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial q_1} & \dots & \frac{\partial \mathbf{f}}{\partial q_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial x_1}{\partial q_1} & \cdots & \frac{\partial x_1}{\partial q_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial x_6}{\partial q_1} & \cdots & \frac{\partial x_6}{\partial q_n} \end{bmatrix} \in \mathbb{R}^{6 \times n}.$$

One important thing to notice is that the Jacobian changes in function of $\mathbf{q}$ (the robotic arm's current state). Indeed, a joint angles velocity command will not have the same effect on the end-effector's velocity according to the robot's current state. We will use $\mathbf{J}$ for notation convenience instead of $\mathbf{J}(\mathbf{q})$.
This Jacobian matrix can be computed exactly, but due to the increasing complexity of robots nowadays, we prefer to compute it approximately (with an autograd tool or by observing the changes in the task space position and orientation when applying a small perturbation dq for each joint).
With the Jacobian we can relate a joint space velocity to a task space velocity with:

$$\mathbf{J}\dot{\mathbf{q}} = \begin{bmatrix} \dot{\mathbf{x}} \\ \mathbf{w} \end{bmatrix},$$

where $\dot{\mathbf{x}}$ denotes the linear velocity, and $\mathbf{w}$ denotes the angular velocity, both in $\mathbb{R}^3$. Before, we used $\dot{\mathbf{x}}$ to denote all the velocities (translation + angular), but from now, we will separate them. Because we are not always interested in the full task space velocities (sometimes we only want to deal with position or orientation). That's why we can shrink the Jacobian in two:

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_T \\ \mathbf{J}_R \end{bmatrix},$$

where $\mathbf{J}_T$ is the part of the Jacobian, which influences only the position (translational Jacobian), and $\mathbf{J}_R$ influences only the orientation (rotational Jacobian).

Inverse kinematics

Forward kinematics is used to represent a robot in the task space, which is more convenient to define action on the robot. Once we analyzed the robot configuration in the task space, we may want to return to the robot a control command to correct/improve the actual robot configuration. The problem is that robots only understand commands expressed in joint space since they are directly related to their motors. That is where inverse kinematics comes, when we need to transform a control command in task space into its corresponding joint space command.
From the forward kinematics equations

$$\mathbf{J}\dot{\mathbf{q}} = \begin{bmatrix} \dot{\mathbf{x}} \\ \mathbf{w} \end{bmatrix},$$

we want to isolate $\dot{\mathbf{q}}$, such that

$$\dot{\mathbf{q}} = \mathbf{J}^{-1} \begin{bmatrix} \dot{\mathbf{x}} \\ \mathbf{w} \end{bmatrix}.$$

The problem is that the Jacobian matrix is not always invertible, its invertibility depends on the number and configuration of the joints of the robot. Since for this project we are working with redundant robots (number of degrees of freedom greater than six), we can say that for our specific case, the Jacobian is never invertible, and here, all the challenge of inverse kinematics is to find a good pseudo-inverse for the Jacobian such that:

$$\dot{\mathbf{q}} \approx \mathbf{J}^{\dagger} \begin{bmatrix} \dot{\mathbf{x}} \\ \mathbf{w} \end{bmatrix},$$

where $\mathbf{J}^{\dagger}$ denotes the Jacobian pseudo-inverse.
In [8], the author proposes several approaches for computing the pseudo-inverse, but in this project, only the weighted pseudo-inverse of the Jacobian will be used:

$$\mathbf{J}^{\dagger} = \mathbf{W}^{-1}\mathbf{J}^{\top}(\mathbf{J}\mathbf{W}^{-1}\mathbf{J}^{\top})^{-1},$$

with $\mathbf{W} \in \mathbb{R}^{n \times n}$ the weight matrix. For this project, the mass inertia matrix $\mathbf{M}$ will be used as weight matrix. This matrix will be presented more in details when we will speak about the dynamics of the robot.

Second order kinematics

It is also possible to link the second order derivatives of $\mathbf{q}$ (joint space acceleration) and $\mathbf{x}$ (task space acceleration) by deriving the Jacobian equation:

$$\frac{d}{dt}(\mathbf{J}\dot{\mathbf{q}}) = \frac{d}{dt}(\begin{bmatrix} \dot{\mathbf{x}} \\ \mathbf{w} \end{bmatrix}),$$

$$\dot{\mathbf{J}}\dot{\mathbf{q}} + \mathbf{J}\ddot{\mathbf{q}} = \begin{bmatrix} \ddot{\mathbf{x}} \\ \dot{\mathbf{w}} \end{bmatrix}.$$

And the corresponding second order inverse kinematics:

$$\ddot{\mathbf{q}} = \mathbf{J}^{\dagger}(\ddot{\mathbf{x}} - \dot{\mathbf{J}}\dot{\mathbf{q}}),$$

where $\dot{\mathbf{J}}$ denotes the time-derivative of the Jacobian, [9] demonstrates a way to compute it accurately by using the chain rule. It is the method used in the project.

2.1.3 Robot dynamics

Dynamics relate to the study of the forces and torques, which are responsible for the motion, whereas, kinematics correspond to the study of the motion without consideration of its causes. In this chapter, we will investigate how a dynamic command (torque or force) applied to a joint will interfere in the motion.
In function of the kind of joint that we want to control, we will rather speak about forces or torques. Indeed, torque would be more meaningful for a revolute joint, and a force would be more accurate for a prismatic joint (basically, an electrical cylinder). In modern robots, prismatic joints tend to disappear; thus when talking about dynamic control of a robot, we will prefer to talk about torques instead of forces.

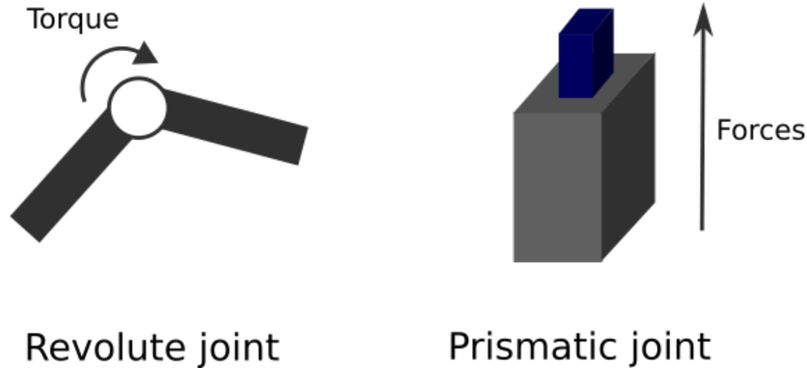


Figure 2.6: Example of revolute and prismatic joints. Since for a revolute joint, the links of the robot are directly connected to a motor, we prefer to talk about torques.

The joint space formulation of the robot dynamics can be described by using the dynamic equation:

$$\boldsymbol{\tau} + \boldsymbol{\tau}_{ext} = \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{g}(\mathbf{q}),$$

where:

- $\boldsymbol{\tau} \in \mathbb{R}^n$ is the vector of torques to execute to get a joint acceleration of $\ddot{\mathbf{q}}$.
- $\boldsymbol{\tau}_{ext} \in \mathbb{R}^n$ is the vector of external torques measured by the robot (due to a perturbation).
- $\mathbf{M}(\mathbf{q}) \in \mathbb{R}^{n \times n}$ is a symmetric positive-definite mass inertia matrix.
- $\mathbf{C} \in \mathbb{R}^{n \times n}$ is the matrix containing information about the centripetal and Coriolis forces.

- $\mathbf{g}(\mathbf{q}) \in \mathbb{R}^n$ is the gravity vector.

Often, we do not have to consider the full equation to control our robot in torque. Some terms are of greater importance than others. For example, the Coriolis and centripetal term is close to zero for small joint velocities, thus it is not absurd to ignore it. Furthermore, we may encounter robots that automatically compensate the gravity,thus $\mathbf{g}$ becomes null and it simplifies the dynamics. Sometimes, the robot SDK does not give all the information needed to implement this equation, thus we have no choices than do without these terms (it is often the case for $\boldsymbol{\tau}_{ext}$, where the cheapest robots do not have a torque sensor on each joint).

Operational space dynamics

Similarly to what has been made in the kinematics part, it is possible to translate the dynamics equation from the joint space to the task space:

$$\boldsymbol{\mathcal{F}} = \boldsymbol{\Lambda} \begin{bmatrix} \ddot{\mathbf{x}} \\ \dot{\mathbf{w}} \end{bmatrix} + \boldsymbol{\mu} + \mathbf{p},$$

with:

$$\boldsymbol{\Lambda} = (\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^\top)^{-1},$$
$$\boldsymbol{\mu} = \boldsymbol{\Lambda}(\mathbf{J}\mathbf{M}^{-1}\mathbf{C}\dot{\mathbf{q}} - \dot{\mathbf{J}}\dot{\mathbf{q}}),$$
$$\mathbf{p} = \boldsymbol{\Lambda}\mathbf{J}\mathbf{M}^{-1}\mathbf{g},$$

where:

- $\boldsymbol{\mathcal{F}} \in \mathbb{R}^6$ is the vector of end-effector forces and wrenches to apply to get the desired acceleration.
- $\boldsymbol{\Lambda} \in \mathbb{R}^{6\times6}$ is the mass inertia matrix reported to the end-effector.
- $\boldsymbol{\mu} \in \mathbb{R}^6$ is the Coriolis and centripetal term reported to the end-effector.
- $\mathbf{p} \in \mathbb{R}^6$ is gravity term reported to the end-effector.
- $\ddot{\mathbf{x}} \in \mathbb{R}^3$ is the linear acceleration of the end-effector.
- $\dot{\mathbf{w}} \in \mathbb{R}^3$ is the angular acceleration of the end-effector.

This equation allows us to perform everything in the task space, but as stated before, a robot only understands commands expressed in joints space. We can transform $\boldsymbol{\mathcal{F}}$ into its corresponding formulation in the joint space by multiplying it by the Jacobian transpose:

$$\boldsymbol{\tau} = \mathbf{J}^\top\boldsymbol{\mathcal{F}}.$$

2.2 Linear Quadratic Tracking

The linear quadratic tracker is an extension of an LQR (Linear Quadratic Regulator) problem, where we want to make an object modelized by a linear system track a trajectory instead of one point. The base of this method is that our system can be modelized as a linear system (in this case, discrete-time):

$$\boldsymbol{\mathcal{S}}_{t+1} = \mathbf{A}\boldsymbol{\mathcal{S}}_t + \mathbf{B}\mathbf{u}_t,$$

where:

- $\boldsymbol{\mathcal{S}}_t \in \mathbb{R}^{\mathcal{I}}$ is the state vector at time t.
- $\mathbf{A} \in \mathbb{R}^{\mathcal{I}\times\mathcal{I}}$ is the system matrix.
- $\mathbf{B} \in \mathbb{R}^{\mathcal{I}\times\mathcal{O}}$ is the input matrix.
- $\mathbf{u}_t \in \mathbb{R}^{\mathcal{O}}$ is the control command at time t.

Given a desired trajectory $\boldsymbol{\mu} \in \mathbb{R}^{\mathcal{I}T}$ composed of all desired states $\{\boldsymbol{\mathcal{S}}_i\}_{i=1}^{i=T}$ vectorized, the purpose of LQT is to find the optimal sequence of control command $\mathbf{u} \in \mathbb{R}^{\mathcal{O}(T-1)}$ (vectorized), which minimizes the cost function:

$$J = ||\boldsymbol{\mu} - \mathbf{x}||^2_{\mathbf{Q}} + ||\mathbf{u}||^2_{\mathbf{R}},$$

where:

- $\mathbf{Q} \in \mathbb{R}^{\mathcal{I}T\times\mathcal{I}T}$ is the precision matrix.
- $\mathbf{R} \in \mathbb{R}^{\mathcal{O}(T-1)\times\mathcal{O}(T-1)}$ is the control weight matrix, which penalizes high values of $\mathbf{u}$.
- $\mathbf{x}$ is the trajectory generated by $\mathbf{u}$, starting at $\boldsymbol{\mathcal{S}}_1$ (vectorized).

This optimization problem can be solve in several ways that will be explored in this project:

- With a batch formulation.
- A dynamic programming solution.
- A quadratic programming solution (if we want to constrain the cost function).

2.2.1 Batch formulation

The main idea behind this formulation, is to solve the LQT as a least square problem. For this, we need a way to formulate $\mathbf{x}$ in function of $\mathbf{u}$ and $\boldsymbol{\mathcal{S}}_1$:

$$\mathbf{x} = f(\boldsymbol{\mathcal{S}}_1, \mathbf{u}).$$

For this, we need to expand our discrete-time linear system equation:

$$\begin{aligned}
\mathbf{x}_1 &= \boldsymbol{\mathcal{S}}_1,\\
\mathbf{x}_2 &= \mathbf{A}\mathbf{x}_1 + \mathbf{B}\mathbf{u}_1 = \mathbf{A}\boldsymbol{\mathcal{S}}_1 + \mathbf{B}\mathbf{u}_1,\\
\mathbf{x}_3 &= \mathbf{A}\mathbf{x}_2 + \mathbf{B}\mathbf{u}_2 = \mathbf{A}^2\boldsymbol{\mathcal{S}}_1 + \mathbf{A}\mathbf{B}\mathbf{u}_1 + \mathbf{B}\mathbf{u}_2,\\
\mathbf{x}_4 &= \mathbf{A}\mathbf{x}_3 + \mathbf{B}\mathbf{u}_3 = \mathbf{A}^3\boldsymbol{\mathcal{S}}_1 + \mathbf{A}^2\mathbf{B}\mathbf{u}_1 + \mathbf{A}\mathbf{B}\mathbf{u}_2 + \mathbf{B}\mathbf{u}_3,\\
&\vdots\\
\mathbf{x}_T &= \mathbf{A}\mathbf{x}_{T-1} + \mathbf{B}\mathbf{u}_{T-1} = \mathbf{A}^{T-1}\boldsymbol{\mathcal{S}}_1 + \mathbf{A}^{T-2}\mathbf{B}\mathbf{u}_1 + \mathbf{A}^{T-3}\mathbf{B}\mathbf{u}_2 + \cdots + \mathbf{B}\mathbf{u}_{T-1},
\end{aligned}$$

where $\mathbf{x}_i$ is the prediction for $\boldsymbol{\mathcal{S}}_i$. It shows that a sequence of states can be entirely defined by its initial state and a list of control commands. Instead of computing each $\mathbf{x}_i$ iteratively as presented above, the set of equations can be vectorized to get the predicted sequence of states in once:

$$\begin{bmatrix}\mathbf{x}_1\\ \mathbf{x}_2\\ \mathbf{x}_3\\ \mathbf{x}_4\\ \vdots\\ \mathbf{x}_T\end{bmatrix} = \underbrace{\begin{bmatrix}\mathbf{I}\\ \mathbf{A}\\ \mathbf{A}^2\\ \mathbf{A}^3\\ \vdots\\ \mathbf{A}^{T-1}\end{bmatrix}}_{\mathbf{S}^{\mathbf{x}}}\boldsymbol{\mathcal{S}}_1 + \underbrace{\begin{bmatrix}\mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0}\\ \mathbf{B} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0}\\ \mathbf{A}\mathbf{B} & \mathbf{B} & \mathbf{0} & \dots & \mathbf{0}\\ \mathbf{A}^2\mathbf{B} & \mathbf{A}\mathbf{B} & \mathbf{B} & \dots & \mathbf{0}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ \mathbf{A}^{T-2}\mathbf{B} & \mathbf{A}^{T-3}\mathbf{B} & \mathbf{A}^{T-4}\mathbf{B} & \dots & \mathbf{B}\end{bmatrix}}_{\mathbf{S}^{\mathbf{u}}}\begin{bmatrix}\mathbf{u}_1\\ \mathbf{u}_2\\ \mathbf{u}_3\\ \mathbf{u}_4\\ \vdots\\ \mathbf{u}_{T-1}\end{bmatrix}.$$

Since this notation is a bit heavy, we usually prefer to write it like this:

$$\mathbf{x} = \mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1 + \mathbf{S}^{\mathbf{u}}\mathbf{u}.$$

We can insert this new expression for $\mathbf{x}$ in the cost function defined above:

$$\begin{aligned}
J &= ||\boldsymbol{\mu} - \mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1 - \mathbf{S}^{\mathbf{u}}\mathbf{u}||^2_{\mathbf{Q}} + ||\mathbf{u}||^2_{\mathbf{R}},\\
J &= (\boldsymbol{\mu} - \mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1 - \mathbf{S}^{\mathbf{u}}\mathbf{u})^{\top}\mathbf{Q}(\boldsymbol{\mu} - \mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1 - \mathbf{S}^{\mathbf{u}}\mathbf{u}) + \mathbf{u}^{\top}\mathbf{R}\mathbf{u}.
\end{aligned}$$

If we solve for $\mathbf{u}$ and target a minimal cost ($J = 0$), we end up with:

$$\hat{\mathbf{u}} = ({\mathbf{S}^{\mathbf{u}}}^{\top}\mathbf{Q}\mathbf{S}^{\mathbf{u}} + \mathbf{R})^{-1}{\mathbf{S}^{\mathbf{u}}}^{\top}\mathbf{Q}(\boldsymbol{\mu} - \mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1),$$

the least square solution for a LQT problem. This formulation has a strong pedagogical potential. Indeed, looking at this formulation, it is easier to understand that

a LQT is nothing more than a weighted Ridge regression, where $\mathbf{R}$ plays the role of the Tikhonov matrix used in the Ridge regression. Also, with this formulation, we can better understand the purpose of the weight matrix $\mathbf{Q}$, which tells our algorithm the desired precision for each state. We can also see the advantage of using a vectorized notation. Without vectorization, $\mathbf{Q}$ would only indicate the desired precision of each state, but having everything vectorized allows us to not only specify the desired precision for one state, but for all the variables of each state.

2.2.2 Dynamic programming formulation

Even if the batch formulation is pedagogically interesting, it involves multiplication of large matrices, which can take an eternity to compute. The solution to overcome this problem is to solve a LQT by using a dynamic programming approach. The approach is described in [10] and consists in solving iteratively the LQT backward in time:

$$\mathbf{P}_t = \mathbf{Q}_t - \mathbf{A}^\top(\mathbf{P}_{t+1}\mathbf{B}(\mathbf{B}^\top\mathbf{P}_{t+1}\mathbf{B} + \mathbf{R}_t)^{-1}\mathbf{B}^\top\mathbf{P}_{t+1} - \mathbf{P}_{t+1})\mathbf{A},$$
$$\mathbf{d}_t = (\mathbf{A}^\top - \mathbf{A}^\top\mathbf{P}_{t+1}\mathbf{B}(\mathbf{B}^\top\mathbf{P}_{t+1}\mathbf{B} + \mathbf{R}_t)^{-1}\mathbf{B}^\top)(\mathbf{P}_{t+1}(\mathbf{A}\boldsymbol{\mathcal{S}}_t - \boldsymbol{\mathcal{S}}_{t+1}) + \mathbf{d}_{t+1}),$$

with the terminal conditions:

$$\mathbf{P}_T = \mathbf{Q}_T \qquad \& \qquad \mathbf{d}_T = \mathbf{0},$$

where $\mathbf{Q}_i \in \mathbb{R}^{\mathcal{I}\times\mathcal{I}}$, and $\mathbf{R}_i \in \mathbb{R}^{\mathcal{O}\times\mathcal{O}}$ are respectively the precision and control weight matrix for state $\boldsymbol{\mathcal{S}_i}$ and control command $\mathbf{u}_i$. The step of computing $\mathbf{P}_t$ (the solution of a discrete-time Ricatti equation), and $\mathbf{d}_t$ (the solution to a linear differential equation) can be retrieved in the LQR problem. It shows us that the dynamic programming solution of LQT consists in computing a discrete-time finite horizon LQR between each state (see [11]).
From $\mathbf{P}_t$, and $\mathbf{d}_t$ we can commpute a feedback gain $\mathbf{K}_t$, and a feedforward term $\mathbf{f}_t$, that will be used to retrieve the control command $\mathbf{u}_t$ for each timestep:

$$\mathbf{K}_t = (\mathbf{B}^\top\mathbf{P}_t\mathbf{B} + \mathbf{R}_t)^{-1}\mathbf{B}^\top\mathbf{P}_t\mathbf{A},$$
$$\mathbf{f}_t = -(\mathbf{B}^\top\mathbf{P}_t\mathbf{B} + \mathbf{R}_t)^{-1}\mathbf{B}^\top(\mathbf{P}_t(\mathbf{A}\boldsymbol{\mathcal{S}}_t - \boldsymbol{\mathcal{S}}_t) + \mathbf{d}_t).$$

Thus, the control command can be computed with:

$$\mathbf{u}_t = \mathbf{K}_t(\boldsymbol{\mathcal{S}}_t - \mathbf{x}_t) + \mathbf{f}_t,$$

where $\mathbf{x}_t$ is the current state of the robot.
Using the dynamic programming formulation instead of batch, reduces the complexity of the problem from $\boldsymbol{\mathcal{O}}(T^3\mathcal{I}\mathcal{O}^2)$ (worst case scenario) to $\boldsymbol{\mathcal{O}}(T\mathcal{I}^3)$. Another advantage of using the dynamic formulation is that the system will compensate for its tracking error (since the control command at a time step is computed in function of the current tracking error).

2.2.3 Constrained LQT (Quadratic programming solution)

The quadratic formulation of the cost, allows us to solve the LQT as a quadratic programming problem. Quadratic programming refers to the process of minimizing a quadratic cost function subject to linear constraints on the optimized variable. It consists of minimizing a subset of a cost function where the constraints are respected. The problem is stated as follow:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}||\mathbf{u}||^2_{\mathbf{P}} + \mathbf{q}^\top \mathbf{u}, \\ \text{subject to} \quad & \mathbf{G}\mathbf{u} \leq \mathbf{h}, \quad \text{(inequality constraint)} \\ \text{and/or} \quad & \mathbf{A}\mathbf{u} = \mathbf{b}, \quad \text{(equality constraint)} \end{aligned}$$

where the constraints can take two forms (equality or inequality). The challenge here, is to transform the cost function of the LQT:

$$J = (\boldsymbol{\mu} - \mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1 - \mathbf{S}^{\mathbf{u}}\mathbf{u})^\top \mathbf{Q}(\boldsymbol{\mu} - \mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1 - \mathbf{S}^{\mathbf{u}}\mathbf{u}) + \mathbf{u}^\top \mathbf{R}\mathbf{u},$$

into something similar to the cost function presented above. Furthermore, it would be interesting to be able to constrain over a state and not a control command (proof presented in appendix B):

$$\begin{aligned} \text{minimize} \quad & \mathbf{u}^\top \underbrace{(\mathbf{S}^{\mathbf{u}\top}\mathbf{Q}\mathbf{S}^{\mathbf{u}} + \mathbf{R})}_{\mathbf{P}} \mathbf{u} + \underbrace{(\boldsymbol{\mathcal{S}}_1^\top \mathbf{S}^{\mathbf{x}\top}\mathbf{Q}\mathbf{S}^{\mathbf{u}} - \boldsymbol{\mu}^\top \mathbf{Q}\mathbf{S}^{\mathbf{u}})}_{\mathbf{q}^\top} \mathbf{u}, \\ \text{subject to} \quad & \mathbf{A}_{\mathcal{S}}\mathbf{S}^{\mathbf{u}}\mathbf{u} \leq \mathbf{b}_{\mathcal{S}} - \mathbf{A}_{\mathcal{S}}\mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1, \end{aligned}$$

where $\mathbf{A}_{\mathcal{S}}$, and $\mathbf{b}_{\mathcal{S}}$ are the constraints exprimed in state space. The corresponding $\mathbf{P}$,$\mathbf{q}$,$\mathbf{A}/\mathbf{G}$, and $\mathbf{b}/\mathbf{h}$ are given to a QP solver to get the solution to our problem.
Using the constrained formulation of LQT, can be particularly interesting to solve the planning problem. With this formulation, we can plan a trajectory that avoids a specific state or forces the motion to pass through a via point at a given time. Its time complexity makes it inefficient for a tracking problem (similar or worst than the batch formulation).

Defining a constraint

Even if having a constraint defined in the command space could be interesting to avoid our system to move too fast. For defining a task, it is more interesting to constrain it in the state space (see chapter 1). In this chapter, the way to build the corresponding $\mathbf{A}_{\mathcal{S}}$ matrix, and $\mathbf{b}_{\mathcal{S}}$ vector will be presented. The construction of the constraint matrix and vector is the same for an inequality or equality constraint, but here we will build them in the case of an equality constraint.
To understand how to create a constraint, lets say that we have:

$$\boldsymbol{\mu} = [\boldsymbol{\mathcal{S}}_1^\top \quad \boldsymbol{\mathcal{S}}_2^\top \quad \boldsymbol{\mathcal{S}}_3^\top]^\top \in \mathbb{R}^{3\mathcal{I}},$$

and we want $\boldsymbol{\mathcal{S}}_2$ to be equal to an arbitrary state $\boldsymbol{\mathcal{S}}_d \in \mathbb{R}^{\mathcal{I}}$. If we formulate our constraint in the state space, we have:

$$\mathbf{A}_{\mathcal{S}}\mathbf{x} = \mathbf{b}_{\mathcal{S}},$$

with $\mathbf{x}$ the vectorized predictions for each state, in function of $\mathbf{u}$, and $\boldsymbol{\mathcal{S}}_1$. The final purpose of this constraint is to express the equality between $\mathbf{x}_2$, and $\boldsymbol{\mathcal{S}}_d$. So, we have to build the constraint matrices with respect to the final condition:

$$\mathbf{x}_2 = \boldsymbol{\mathcal{S}}_d.$$

It can be done by constructing the matrices like:

$$\mathbf{A}_{\mathcal{S}} = \begin{bmatrix} \mathbf{0} & \mathbf{I} & \mathbf{0} \end{bmatrix}, \mathbf{b}_{\mathcal{S}} = \boldsymbol{\mathcal{S}}_d,$$

so we have:

$$\begin{bmatrix} \mathbf{0} & \mathbf{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} = \boldsymbol{\mathcal{S}}_d,$$

which gives:

$$\mathbf{x}_2 = \boldsymbol{\mathcal{S}}_d.$$

Here we introduce a constraint over a whole state, but similarly, it is possible to constrain multiple states or only some variable of a state. Generally, one constraint corresponds to one row in the $\mathbf{A}_{\mathcal{S}}$, and $\mathbf{b}_{\mathcal{S}}$ matrices.
Similarly to what has been made before, it is possible to constrain dynamically two states together, if we take back our previous example, and we ask that state $\boldsymbol{\mathcal{S}}_1$ is equal to $\boldsymbol{\mathcal{S}}_3$, the resulting matrices will be:

$$\mathbf{A}_{\mathcal{S}} = \begin{bmatrix} \mathbf{I} & \mathbf{0} & -\mathbf{I} \end{bmatrix}, \mathbf{b}_{\mathcal{S}} = \mathbf{0}.$$

It is particularly interesting, since we do not have to bother about the meeting state, since it is computed by the QP solver.
Once we have calculated the corresponding state space constraint matrices, we can transform them into their corresponding command space formulation with the equations presented above.

2.2.4 Performing LQT in a model predictive control way (MPC)

The main limitation of the LQT is that it needs a linear system to work. If our system is not linear, we can not directly apply LQT on our task. To take a robotics example, if we define our states in the task space of the robot and we want to have a command in the joint space (e.g. in joint velocities.), we have to introduce the joint space to task space transformation inside the $\mathbf{A}$, and $\mathbf{B}$ matrices:

$$\mathbf{x}_{t+1} = \underbrace{\mathbf{I}}_{\mathbf{A}} \mathbf{x}_t + \underbrace{\mathrm{dt}\mathbf{J}}_{\mathbf{B}} \dot{\mathbf{q}},$$

where $\mathbf{x}_i$ denotes an end-effector position at time i, and dt denotes the time step of our discrete-time system. The problem with this formulation is that it is not linear since the Jacobian evolves in function of the state of the robot, it is not constant over time. Thus, it is impossible to apply LQT directly on that system. The proposed solution to overcome this issue is to perform the LQT in a model predictive control way.

Basically, for each time step, we solve a LQT with a fixed horizon of $\boldsymbol{\mu}$ (from t to $t + H$), in which we assume that the $\mathbf{A}$, and $\mathbf{B}$ matrices are constant. From this LQT, we execute only the first control command.

Data: $\boldsymbol{\mu}$, a desired state sequence of length T, and h a window size.
$t = 0$;
while *t not equal to $T - 1$* **do**
  $\boldsymbol{\mu}_{part} = \boldsymbol{\mu}[t \cdot \text{nbStateVar}:(t+h) \cdot \text{nbStateVar}\ ]$;
  $\boldsymbol{\mu}_{part}$[0:nbStateVar] = robot.getCurrentState() ;
  $\mathbf{A}$, $\mathbf{B}$ = getSystemMatrices() ;
  ctrl = LQT($\mathbf{A}$,$\mathbf{B}$,$\boldsymbol{\mu}_{part}$) ;
  cmd = ctrl.getCommand(timestep=0) ;
  robot.sendJointVelCommand(cmd) ;
  t += 1 ;
end

It is not described in the algorithm, but when selecting the horizon of time step t, we have to check if $t + h$ is not bigger than T. If yes, we crop it to get the sequence till the end of $\boldsymbol{\mu}$.

An advantage of this technique is that it automatically compensates for the error since, at each time step, the $\boldsymbol{\mu}_{part}$ vector is updated with the current state of the system.

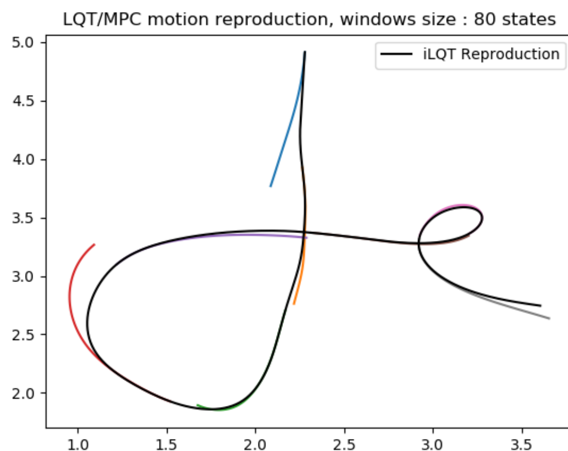


Figure 2.7: Example of a reproduced end-effector motion made with LQT/MPC. The black curve corresponds to the reproduction. Color curves are partial results obtained if we use all control commands instead of only the first one. It shows that at the beginning of the horizon, it is not absurd to assume that the system matrices are constant.

2.2.5 Applications of LQT

Until now, the LQT and its different ways to solve it was presented in a really mathematical manner, to the point of forgetting robotics for a while. In this chapter, some robotics use of LQT will be presented and explained. Here, how to solve a normal LQT is not important, the focus will be on how to use this brick to build a robotics application.

Example of constrained LQT

As said during the constrained LQT introduction. This way of solving LQT can be very interesting to solve a planning problem. Solving the planning problem means to generate a trajectory $\boldsymbol{\mu}$, which respects all the constraints of the task. For example, if the robot has to insert an object in a hole, it would be interesting to constraint the last 25% of the motion to have a velocity only in the direction of the axis of the hole, instead of constraining all the velocities from the 75% to the end of the motion to be in the direction of the hole axis (it would be really heavy). It would be more efficient to ask the robot to be on top of the hole at 75% of the motion. Doing so, the resulting motion only needs to move in the direction of the axis. This ensures a velocity in the desired orientation for the last part of the motion.
To do this, the states are expressed in the task space and the commands too (it avoids a non-linearity problem, thus we can directly use LQT on the task). The system is modelized as a double integrator system:

$$\begin{bmatrix} \mathbf{x}_{t+1} \\ \dot{\mathbf{x}}_{t+1} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{dt} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{t} \\ \dot{\mathbf{x}}_{t} \end{bmatrix} + \begin{bmatrix} \frac{\mathbf{dt}^2}{2} \\ \mathbf{dt} \end{bmatrix} \ddot{\mathbf{x}}_t,$$

where $\mathbf{x}_t$ denotes the end-effector position at time step t. $\mathbf{x}_{75\%}$, the position at 75% of the motion should be equal to:

$$\mathbf{x}_{75\%} = \boldsymbol{\mathcal{S}}_T + d\,\mathbf{e_h},$$

where:

- $\boldsymbol{\mathcal{S}}_T$ is the final position of the motion (a point in the hole).
- $\mathbf{e_h}$ the unit norm hole axis vector.
- d the desired distance to the final position.

From the equation above, the corresponding $\boldsymbol{A}_{\mathcal{S}}$ matrix and $\boldsymbol{b}_{\mathcal{S}}$ vector are built and used to solve the corresponding LQT problem. Before solving the LQT, we need to pay attention to the $\mathbf{Q}$ matrix, since in this case we only know the initial and final state of the trajectory, the diagonal of the LQT matrix should only contain ones where the corresponding state is defined:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{I} \end{bmatrix},$$

each element inside the definition of the **Q** matrix corresponds to a six by six matrix to match the size of one state (three variables for the position, and three variables for the velocity).

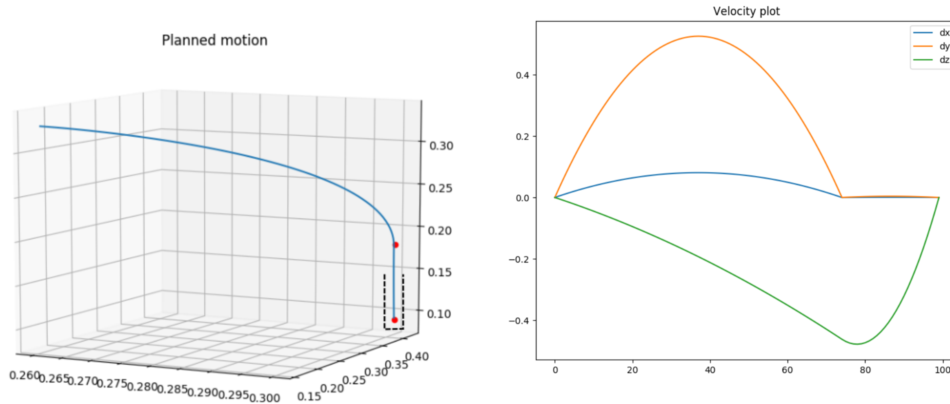


Figure 2.8: Generated motion and velocities of the problem. Constraining the motion like this allows the system to correctly goes into the hole. This kind of constrained motion can also be used in a grasping task to ensure a correctly grasp of an object.

Multiple agents control

Until now, only the control of one agent with a LQT has been presented, but the formulation of LQT allows to use the same regulator for multiple agents:

$$\begin{bmatrix}\mathbf{x}_{1,t+1}\\ \mathbf{x}_{2,t+1}\end{bmatrix} = \begin{bmatrix}\mathbf{A}_1 & \mathbf{0}\\ \mathbf{0} & \mathbf{A}_2\end{bmatrix}\begin{bmatrix}\mathbf{x}_{1,t}\\ \mathbf{x}_{2,t}\end{bmatrix} + \begin{bmatrix}\mathbf{B}_1 & \mathbf{0}\\ \mathbf{0} & \mathbf{B}_2\end{bmatrix}\begin{bmatrix}\mathbf{u}_{1,t}\\ \mathbf{u}_{2,t}\end{bmatrix},$$

where the first index of matrices and vectors denotes the agent in question, and the second index denotes the time step of the variable, if not present, the variable is assumed to be constant over the motion (N.B. in this formulation, $\mathbf{x}_{i,j}$ denotes the predicted state of agent i at time step j, it is not necessary (only) the end-effector position). The challenge in the multiple agent formulations consists in correctly instantiate the different variables of the LQT, from the linear system equation above, we can infer that the desired state sequence vectorized $\boldsymbol{\mu}$ should be constructed as follow:

$$\boldsymbol{\mu} = [\boldsymbol{\mathcal{S}}_{1,1}^\top \quad \boldsymbol{\mathcal{S}}_{2,1}^\top \quad \dots \quad \boldsymbol{\mathcal{S}}_{1,T}^\top \quad \boldsymbol{\mathcal{S}}_{2,T}^\top]^\top.$$

The different agents do not need to have the same structure of **A** and **B** matrices for the regulator to work. But in practice, we have no reasons to control different agents in different manners, so if we are not using the MPC solution of LQT (i.e. we have no information relative to the current robot state in the matrices), the system matrices will be the same for the different agents.
A big advantage of having multiple agents in one LQT is that we can constrain the agents together by using the QP formulation. For example, we can add a meeting

point dynamically between the two agents at a defined time step. It is particularly interesting if we want our robots to interact during their task.

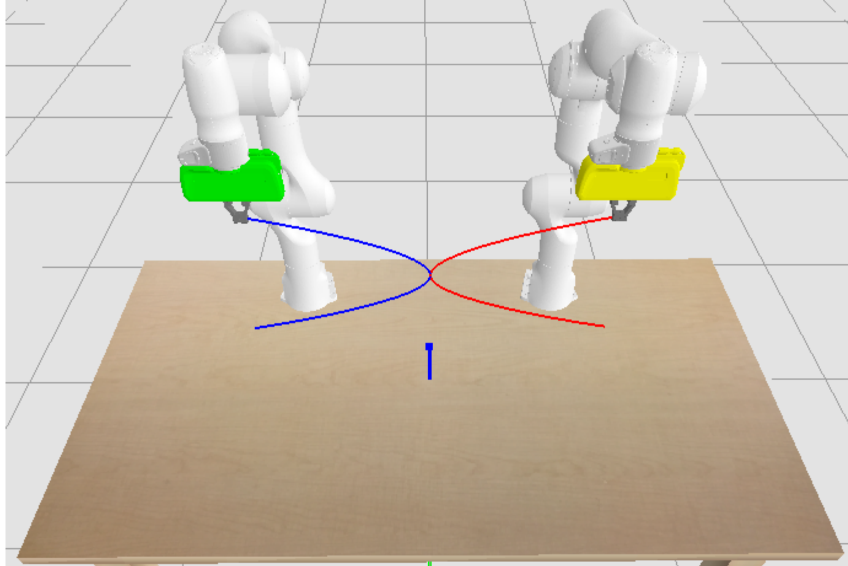


Figure 2.9: In this image, we use a constrained LQT to ask the two agents to meet in the middle of their motion. From this LQT, we can retrieve the corresponding motion for each agent (plotted in red and blue on the figure). The result of this LQT will be given to another regulator who would have to track the motion.

2.3 Riemannian manifold

In chapter 2.1.1, we discussed the difficulties of handling the end-effector orientation in robotics applications. We ended up with the conclusion that the representation by unit quaternion is the one chosen in this project, without really explaining why. In this chapter, we are going to answer this question by presenting a way to track the orientation of robots by relying on the S^d Riemannian manifold.

2.3.1 Motivation

The quadratic cost function of the LQT can be divided into two parts, the tracking part, where we ensure that the predicted state sequence is close to the desired, and the normalizing part, where we want to minimize the cost to go from a state to another:

$$J = \underbrace{||\boldsymbol{\mu} - \mathbf{x}||^2_{\mathbf{Q}}}_{\text{Tracking part}} + \underbrace{||\mathbf{u}||^2_{\mathbf{R}}}_{\text{Normalizing part}} .$$

In the tracking part, we measure the tracking error by assuming that the state space is an Euclidean space (i.e., the difference between two states can be measured by subtracting one state to another). As presented with the most common orientation representations in robotics (chapter 2.1.1), none of them can accurately measure an orientation difference with an Euclidean distance. Thus, it is impossible to track or plan orientation by using a LQT.

It is where the Riemannian manifold comes. A Riemannian manifold is a smooth and differentiable manifold, which locally behaves like an Euclidean space. In [12] the author briefly presented some manifolds, which belong to the Riemannian family. From this list, we will only consider the sphere manifold S^d [13] [14], which represents all points lying on the manifold as points being on the surface of $(d+1)$-dimensional sphere. Since unit quaternions can be seen as points lying on the hyper-sphere [2] S^3, we can use the corresponding manifold to find a space, where all orientations behave locally as an Euclidean space, and apply LQT on this manifold.

2.3.2 Definition of $\mathrm{S^d}$ manifold

For a point $\mathbf{p}$ lying on the manifold $\mathcal{M}$, there exists a tangent space $\mathcal{T}_{\mathbf{p}}\mathcal{M}$, which behaves locally as an Euclidean space. The action of projecting a point $\mathbf{y} \in \mathcal{M}$ into the tangent space of $\mathbf{p}$ is called the logarithmic map. The reverse action is called the exponential map.

[2] If you want to know, how a 4-d sphere looks like, i recommend this video: `https://www.youtube.com/watch?v=dy_MUfBuq2I` (audio in French, but subtitles are available in English).

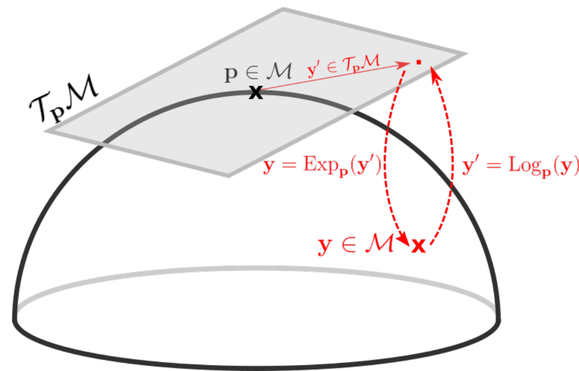


Figure 2.10: Visualization of the S^2 manifold (a 3d sphere), where we project a point $\mathbf{y}$ lying on the manifold into the tangent space of the point $\mathbf{p}$.

The advantage of projecting a point into the tangent space of another point is that inside the tangent space, the difference between the two points can be computed by using an Euclidean distance. Thus, we can apply LQT inside this tangent space.

Something important to notice is that a vector on the tangent space of $\mathbf{p_1}$ is not equivalent to the same vector but expressed in the tangent space of $\mathbf{p_2}$. If a point is moving at constant speed on the manifold, its velocity vector at time t will be in the tangent space of the current position of the point, but if we are analyzing the velocity vector at another time, it will not be the same as the previous one (somehow, the vector will need to adapt to the curvature of the sphere). The action of transporting a vector $\mathbf{v}$ from $\mathcal{T}_{\mathbf{p_1}}\mathcal{M}$ to $\mathcal{T}_{\mathbf{p_2}}\mathcal{M}$ is called the parallel transport of $\mathbf{v}$.

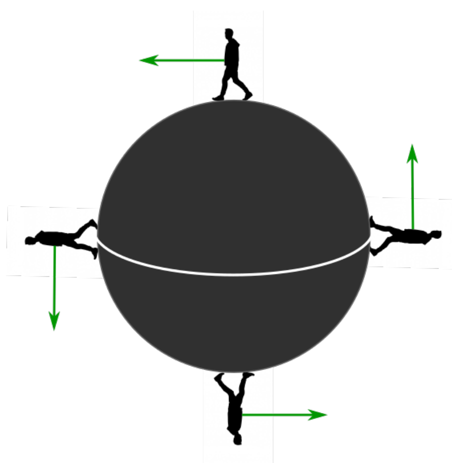


Figure 2.11: The easiest example of parallel transport could be a human walking for a (very) long time in a fixed direction on earth (earth can be assumed to be spherical, thus it can act like a S^2 sphere). Sooner or later, he will come back to its starting point, but if we take a look at the evolution of its velocity vector during its journey, we notice that the vector changed all along the way (the vector is presented in green on the figure above), but from his point of view, his velocity never changed.

Mathematical notation

Now that all the different actions than can be performed on the manifold has been presented, the different mathematical formulations for these actions will be presented. These formula work for the S^d manifold for $d \geq 1$.
The exponential map of a point $\mathbf{u} \in \mathcal{T}_{\mathbf{x}}\mathcal{M}$:

$$\mathbf{y} = \text{Exp}_{\mathbf{x}}(\mathbf{u}) = \mathbf{x}\cos(||\mathbf{u}||) + \frac{\mathbf{u}}{||\mathbf{u}||}\sin(||\mathbf{u}||).$$

The logarithmic map in the tangent space of $\mathbf{x}$ of a point $\mathbf{y} \in \mathcal{M}$:

$$\mathbf{u} = \text{Log}_{\mathbf{x}}(\mathbf{y}) = d(\mathbf{x}, \mathbf{y})\frac{\mathbf{y} - \mathbf{x}^\top\mathbf{y}\mathbf{x}}{||\mathbf{y} - \mathbf{x}^\top\mathbf{y}\mathbf{x}||},$$

with $d(\mathbf{x}, \mathbf{y})$, the distance between $\mathbf{x}$ and $\mathbf{y}$:

$$d(\mathbf{x}, \mathbf{y}) = \begin{cases} \arccos(\mathbf{x}^\top\mathbf{y}) - \pi, & \mathbf{x}^\top\mathbf{y} < 0 \\ \arccos(\mathbf{x}^\top\mathbf{y}), & \text{otherwise} \end{cases},$$

The parallel transport of $\mathbf{v} \in \mathcal{T}_{\mathbf{x}}\mathcal{M}$ to $\mathcal{T}_{\mathbf{y}}\mathcal{M}$:

$$\Gamma_{\mathbf{x}\to\mathbf{y}} = \mathbf{v} - \frac{\text{Log}_{\mathbf{x}}(\mathbf{y})^\top\mathbf{v}}{d(\mathbf{x}, \mathbf{y})^2}(\text{Log}_{\mathbf{x}}(\mathbf{y}) + \text{Log}_{\mathbf{y}}(\mathbf{x})).$$

2.3.3 LQT with $\mathbf{S^d}$ manifold

Now that we are more familiar with the S^d manifold. We will take a look on how to use this tool in an orientation tracking or planning problem for a robotics application. It exists two different manners to perform a LQT by relying on the Riemannian manifold:

1. Solve the problem with only one LQT performed in the tangent space of the initial state (it means that we project all parts relative to the orientation in $\{\boldsymbol{\mathcal{S}}_i\}_{i=1}^{i=T}$ into $\mathcal{T}_{\boldsymbol{\mathcal{S}}_1}\mathcal{M}$).
2. A LQT/MPC solution where for each time step, we project a fixed horizon of the motion into the tangent space of the current state.

Simple LQT solution

Given $\boldsymbol{\mu}$, the vector containing the desired state sequence with all states corresponding to orientations expressed in unit quaternions and their corresponding angular velocities (if we want to perform acceleration control on the system). We first need to transform the angular velocities of each time step into quaternion derivatives, [15] gives us a way to do this:

$$\dot{\boldsymbol{\epsilon}} = \frac{1}{2}\mathbf{H}(\boldsymbol{\epsilon})^\top\mathbf{w},$$

with:

$$\mathbf{H}(\boldsymbol{\epsilon}) = \begin{bmatrix} -\epsilon_1 & \epsilon_0 & -\epsilon_3 & \epsilon_2 \\ -\epsilon_2 & \epsilon_3 & \epsilon_0 & -\epsilon_1 \\ -\epsilon_3 & -\epsilon_2 & \epsilon_1 & \epsilon_0 \end{bmatrix},$$

and the inverse transformation:

$$\mathbf{w} = 2\mathbf{H}(\boldsymbol{\epsilon})\dot{\boldsymbol{\epsilon}}.$$

Something important to notice is that the velocity of a unit quaternion is a vector lying in the tangent space of the current position (e.g., if you attach an extremity of a string to a weight, and you use the other extremity to make the weight turn, its velocity will always be tangent to the circle delimited by the string).
Now, the vector $\boldsymbol{\mu}$ looks like:

$$\boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\epsilon}_1 \\ \dot{\boldsymbol{\epsilon}}_1 \\ \vdots \\ \boldsymbol{\epsilon}_T \\ \dot{\boldsymbol{\epsilon}}_T \end{bmatrix},$$

and $\boldsymbol{\mathcal{S}}_i$ is equal to $[\boldsymbol{\epsilon}_i^\top \quad \dot{\boldsymbol{\epsilon}}_i^\top]^\top$.
Now that the ground truth vector is defined only with information relatives to unit quaternions. We can express this vector in the point of view of $\boldsymbol{\epsilon}_1$. For this, we project all $\{\boldsymbol{\epsilon}_i\}_{i=1}^{i=T}$ into the tangent space of the first orientation, and we transport all $\{\dot{\boldsymbol{\epsilon}}_i\}_{i=2}^{i=T}$ from their tangent space to the initial tangent space (we start at $i = 2$, because $\dot{\boldsymbol{\epsilon}}_1$ is already in the tangent space). It gives the following vector:

$$\boldsymbol{\mu} = \begin{bmatrix} \mathrm{Log}_{\boldsymbol{\epsilon}_1}(\boldsymbol{\epsilon}_1) \\ \dot{\boldsymbol{\epsilon}}_1 \\ \mathrm{Log}_{\boldsymbol{\epsilon}_1}(\boldsymbol{\epsilon}_2) \\ \Gamma_{\boldsymbol{\epsilon}_2 \to \boldsymbol{\epsilon}_1}(\dot{\boldsymbol{\epsilon}}_2) \\ \vdots \\ \mathrm{Log}_{\boldsymbol{\epsilon}_1}(\boldsymbol{\epsilon}_{T-1}) \\ \Gamma_{\boldsymbol{\epsilon}_{T-1} \to \boldsymbol{\epsilon}_1}(\dot{\boldsymbol{\epsilon}}_{T-1}) \\ \mathrm{Log}_{\boldsymbol{\epsilon}_1}(\boldsymbol{\epsilon}_T) \\ \Gamma_{\boldsymbol{\epsilon}_T \to \boldsymbol{\epsilon}_1}(\dot{\boldsymbol{\epsilon}}_T) \end{bmatrix}.$$

If we are interested in solving a planning problem, often, all states of $\boldsymbol{\mu}$ are not known, and we want to use a LQT to find them. These unknown states are filled with zeros in the $\boldsymbol{\mu}$ vector. Since we do not want these unknown states to interfere in the cost function of the LQT, we have to hack the precision matrix $\mathbf{Q}$ with zeros in its diagonal for the corresponding states. With the desired trajectory defined in the tangent space of the initial point, all the states are part of an Euclidean space, thus it can be solved with a double integrator LQT as seen before. The extraction of the result is not the same if we are solving a tracking or a planning problem. For a planning problem, we are only interested in the resulting trajectory and velocities.

The trajectory can be transformed back to the manifold by the reverse function of Log, Exp.
Similarly, for the velocities, they can be transported to their original tangent space with the parallel transport function. Thus, we end up with a full trajectory defined with unit quaternions. For a tracking problem, the control command at time t is a unit quaternion acceleration defined in the tangent space of the initial state. This control command can be transported into its corresponding tangent space, but unfortunately, robots do not understand these kinds of commands. Before, a mapping between quaternions velocities and angular velocities was presented. Similarly, in a double integrator system, to send a command to the robot, we need a mapping between quaternions accelerations and angular accelerations. It can be achieved by deriving the previous equation with respect to time:

$$\begin{aligned}\frac{d}{dt}(\mathbf{w}) &= 2\frac{d}{dt}(\mathbf{H}(\boldsymbol{\epsilon})\dot{\boldsymbol{\epsilon}}),\\ &= 2(\dot{\mathbf{H}}(\boldsymbol{\epsilon})\dot{\boldsymbol{\epsilon}} + \mathbf{H}(\boldsymbol{\epsilon})\ddot{\boldsymbol{\epsilon}}).\end{aligned}$$

Since $\mathbf{H}(\boldsymbol{\epsilon})$ is nothing more than a matrix representation of the imaginary part of a quaternion (conjugate), its time-derivative is then equal to:

$$\dot{\mathbf{H}}(\boldsymbol{\epsilon}) = \mathbf{H}(\dot{\boldsymbol{\epsilon}}),$$

and the resulting angular acceleration can be used as control command with the help of the operational space dynamics.

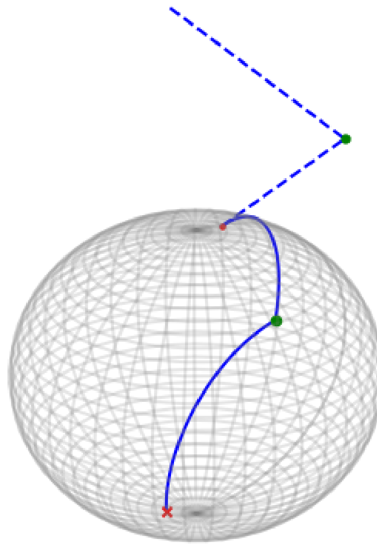


Figure 2.12: Example of a planning task on the S^2 manifold (simple integrator system). The tilted blue line corresponds to the solution in the tangent space of the initial point. The green point is a via-point in the middle of the trajectory, and the red cross is the desired final state.

If you pay attention to figure 2.12, you can see that the resulting trajectory on the manifold is not really optimal. A better solution would be to directly connect the

three points with a line on the sphere. Unfortunately, the solution given by the algorithm is optimal only from the point of view of the initial state. To improve this solution, and have an optimal solution for the entire trajectory, a LQT/MPC solution will be described in the next chapter.

LQT/MPC solution

As stated before, the problem with the single LQT on the manifold is that the resulting trajectory or control commands are optimal only for the point of view of the initial state. The main idea in this chapter is to use the LQT/MPC brick presented in chapter 2.2.4 to find a better solution for the planning or the tracking of orientation.

For this, at each timestep t, we project a fixed horizon H of $\boldsymbol{\mu}$ into the tangent space of the current state:

$$\boldsymbol{\mu}_t = \begin{bmatrix} \mathrm{Log}_{\boldsymbol{\epsilon}_t}(\boldsymbol{\epsilon}_t) \\ \dot{\boldsymbol{\epsilon}}_t \\ \vdots \\ \mathrm{Log}_{\boldsymbol{\epsilon}_t}(\boldsymbol{\epsilon}_{t+H}) \\ \Gamma_{\boldsymbol{\epsilon}_{t+H}\to\boldsymbol{\epsilon}_t}(\dot{\boldsymbol{\epsilon}}_{t+H}) \end{bmatrix}.$$

This horizon is only valid if all states are known, if we want to solve a planning problem in which we know only the initial and final states, and a constraint, at each time step we would have to consider the motion from the current state to the final state and adapt the $\mathbf{Q}$ matrix and constraints matrices.

Now that the $\boldsymbol{\mu}_t$ is correctly defined, we can solve a LQT and use only the first control command. For a tracking problem, this control command can be transformed into angular acceleration and sent to the robot with the formula presented above. For a planning problem, we want to calculate the next state in function of this command. It can be done with the discrete-time linear system equation presented in chapter 2.2:

$$\boldsymbol{x}_{t+1} = \mathbf{A}\boldsymbol{\mathcal{S}}_t + \mathbf{B}\mathbf{u}_t,$$

where $\boldsymbol{\mathcal{S}}_t = [\mathrm{Log}_{\boldsymbol{\epsilon}_t}(\boldsymbol{\epsilon}_t)^\top \quad \dot{\boldsymbol{\epsilon}}_t^\top]^\top$ is the current state expressed in its tangent space, and $\boldsymbol{x}_{t+1} = [\mathbf{a}^\top \quad \mathbf{b}^\top]^\top$ is the next state expressed in $\mathcal{T}_{\boldsymbol{\mathcal{S}}_t}\mathcal{M}$, which can be translated to its corresponding quaternion state:

$$\boldsymbol{\epsilon}_{t+1} = \mathrm{Exp}_{\boldsymbol{\epsilon}_t}(\mathbf{a}),$$
$$\dot{\boldsymbol{\epsilon}}_{t+1} = \Gamma_{\boldsymbol{\epsilon}_t\to\boldsymbol{\epsilon}_{t+1}}(\mathbf{b}).$$

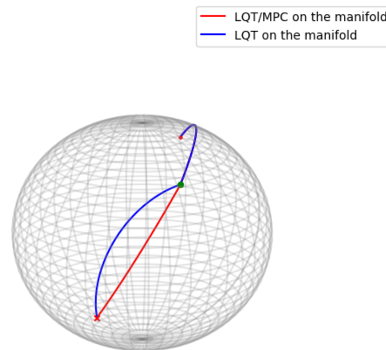


Figure 2.13: Same example as figure 2.12 with the LQT/MPC solution in red. As you can see, the LQT/MPC solution provides a better trajectory, which is optimal for the whole motion.

Even if this algorithm gives a good solution for the planning and the tracking of orientation, its computation is quite expensive (especially if we want to solve a planning problem and not all states are known), and sometimes the result is quite similar to the single LQT method (in fact, if the trajectory is not constrained, both ways give exactly the same result with an advantage for the first one, since the computational cost is quite low).

Chapter 3

Proposed approach

Now that the different methods used in the project have been presented. The focus of this chapter will be on how they are used together to fulfill the goal of this project. As a recall of chapter 1, the purpose of this project is to make a robot fulfill a task by specifying it the goal and not actions that lead to the goal. In the context of this project, a task is fully defined by some entities:

- The duration of the task (in time step, since we use discrete formulation).
- The initial and final state (where we want to start, and where we want to end).
- One or more constraints (equality or inequality), which aim to inform our system what should happen between the beginning and the end of the motion.

The constrained formulation of LQT needed to deal with the constraints of the motion does not offer error corrections (whereas the LQT/MPC and dynamic programming formulations yes), and is quite expensive to compute, thus we can not use a single LQT to plan and track the motion. We have to break the problem into two parts, the planning, and the tracking. The purpose of the planning part will be to find an end-effector trajectory which satisfies the task, and the tracking part will make the robot follows this trajectory by finding a sequence of control command in joint space, which satisfies the planned motion.
Here, a state gives information about the position and orientation of the end-effector at a given time step. If it is a single integrator system, only the position and orientation will be used, but if we consider a double integrator system, at each time step, a state will be defined by the position, orientation, and their time derivatives. The main difference is the kind of control command used, a single integrator will generate velocity commands, and a double integrator will generate acceleration command, more higher is the degree of derivatives that you use to control your system, the more smoother the generated trajectory will be (therefore a motion generated by acceleration commands will look more natural than one generated by velocity commands). In this chapter, we will assume that we always use a double integrator system, since they are more complex to handle, and once we understand a double integrator system it is easy to formulate a task as a single integrator system.

Regarding the constraints, their definitions matrices are the same if we want to create equality or inequality constraints (one time, it will just be $\mathbf{Ax} = \mathbf{b}$ instead of $\mathbf{Ax} \leq \mathbf{b}$). Thus, here, we will assume that we are only using inequality constraints, but if you want to use equality constraints or both, you can build the matrices in the same way and give them to the QP Solver (most performant QP solvers accept equality and/or inequality constraint).

3.1 Planning approach

In chapter 2, we have seen that it is not possible to use the same approach for orientation and position. Thus, we have to plan them separately. Here we are assuming that the orientation and position trajectories are constrained, therefore we will use the quadratic programming formulation of the LQT in both cases (but not in the same way).

As said before, the QP formulation of the LQT, is quite expensive to compute. Solving a QP problem with more than one thousand time step could literally take an eternity to compute (in practice, it is almost always the case, since we use a time step of one millisecond, and a motion lasts longer than one second). Thus, it is impossible to use it directly in a real-time scenario[1].

To overcome this problem, the QP solver will solve a downsampled version of the motion, and its result will be oversampled to match the desired length of the motion. It allows a faster computation of the planning part.

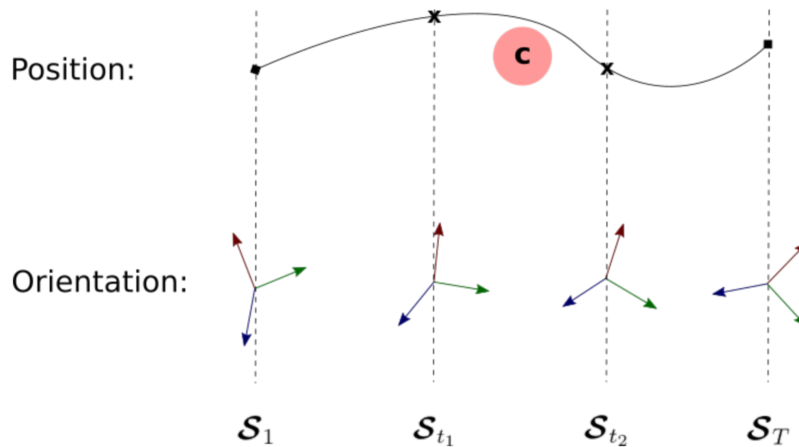


Figure 3.1: Example of a planning problem, where we know the initial and target state for the position and orientation, and a constraint. We want to retrieve the intermediary states.

[1]Here, real-time means that we want to solve the planning at the beginning of each movement of the robot.

3.1.1 Position planning

As seen before, dealing with the position is not a big challenge, since the position of the end-effector is described in an Euclidean space, we do not have to make special transformations to plan the position. Given an initial state $\boldsymbol{\mathcal{S}}_1$, a final state $\boldsymbol{\mathcal{S}}_T$, a desired length T, a downsampling ratio D, and the constraints, we will define our downsampled problem as:

$$\begin{aligned} \text{minimize} \quad & \mathbf{u}^\top \underbrace{(\mathbf{S^u}^\top \mathbf{Q} \mathbf{S^u} + \mathbf{R})}_{\mathbf{P}} \mathbf{u} + \underbrace{(\boldsymbol{\mathcal{S}}_1^\top \mathbf{S^x}^\top \mathbf{Q} \mathbf{S^u} - \boldsymbol{\mu}^\top \mathbf{Q} \mathbf{S^u})}_{\mathbf{q}^\top} \mathbf{u}, \\ \text{subject to} \quad & \mathbf{A}_{\mathcal{S}} \mathbf{S^u} \mathbf{u} \leq \mathbf{b}_{\mathcal{S}} - \mathbf{A}_{\mathcal{S}} \mathbf{S^x} \boldsymbol{\mathcal{S}}_1, \end{aligned}$$

with:

$$\boldsymbol{\mathcal{S}}_i = [\mathbf{x}_i^\top \quad \dot{\mathbf{x}}_i^\top]^\top \in \mathbb{R}^6,$$

$$\boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mathcal{S}}_1 \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \boldsymbol{\mathcal{S}}_T \end{bmatrix} \in \mathbb{R}^{6\lfloor \frac{T}{D} \rfloor}, \mathbf{Q} = \begin{bmatrix} \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{I} \end{bmatrix} \in \mathbb{R}^{6\lfloor \frac{T}{D} \rfloor \times 6\lfloor \frac{T}{D} \rfloor}.$$

Since we are solving a downsampled version of the motion, we have to adapt the time step of the system. Thus, in case of a double integrator system, the system matrices become:

$$\mathbf{A} = \begin{bmatrix} \mathbf{I} & \mathbf{dtD} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \in \mathbb{R}^{6\times 6}, \mathbf{B} = \begin{bmatrix} \frac{(\mathbf{dtD})^2}{2} \\ \mathbf{dtD} \end{bmatrix} \in \mathbb{R}^{6\times 3}.$$

The constraint matrices are built with exactly the same logic than before, but the number of columns of $\mathbf{A}_{\mathcal{S}}$ should be equal to the number of rows of $\boldsymbol{\mu}$. In practice, it is better to choose a D that results in an integer number of downsampled states (no need to floor $\frac{T}{D}$). Doing so, the downsampled time step $\mathrm{dt}D$ is exact.

Once that all the elements are computed, the different elements can be given to a QP solver, which will return $\hat{\mathbf{u}} \in \mathbb{R}^{3(\lfloor \frac{T}{D} \rfloor - 1)}$, the sequence of control commands which minimizes the cost function. With this sequence, the downsampled motion can be computed with:

$$\mathbf{x} = \mathbf{S^x} \boldsymbol{\mathcal{S}}_1 + \mathbf{S^u} \hat{\mathbf{u}}.$$

To retrieve the desired trajectory, the downsampled trajectory is upsampled by a factor of D. Upsampling is made through two three-dimensional spline interpolation (one for the positions, and one for the velocities). At the end of the upsampling, we have a vector $\mathbf{x} \in \mathbb{R}^{6T}$ corresponding to the sequence of positions and velocities for each timestep.

3.1.2 Orientation planning

As seen before, there are two different methods to deal with the orientation. Since we are dealing with constrained trajectories, we will focus on the LQT/MPC solution, which gives a more natural result of the trajectory (using the single LQT solution is similar to the position planning except that we have to project everything in the tangent space of the initial orientation). The planning approach for the LQT/MPC solution has been briefly introduced during the description of the method. Since in this case, only constraints, and the initial and the final state are known, the horizon is not constant over time, and has to capture the rest of the motion at each time step. As in the position part, we first solve a downsampled version of the motion (by factor D):

$$\boldsymbol{\mu}_t = \begin{bmatrix} \mathrm{Log}_{\boldsymbol{\epsilon}_t}(\boldsymbol{\epsilon}_t) \\ \dot{\boldsymbol{\epsilon}}_t \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathrm{Log}_{\boldsymbol{\epsilon}_t}(\boldsymbol{\epsilon}_T) \\ \Gamma_{\boldsymbol{\epsilon}_T \to \boldsymbol{\epsilon}_t}(\dot{\boldsymbol{\epsilon}}_T) \end{bmatrix} \in \mathbb{R}^{8(\lfloor \frac{T}{D} \rfloor - (t-1))}, \mathbf{Q}_t = \begin{bmatrix} \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{I} \end{bmatrix} \in \mathbb{R}^{8(\lfloor \frac{T}{D} \rfloor - (t-1)) \times 8(\lfloor \frac{T}{D} \rfloor - (t-1))},$$

with:

$$\boldsymbol{\mathcal{S}}_t = [\mathrm{Log}_{\boldsymbol{\epsilon}_t}(\boldsymbol{\epsilon}_t)^\top \quad \dot{\boldsymbol{\epsilon}}_t^\top]^\top \in \mathbb{R}^8.$$

Since the motion is downsampled, the $\mathbf{A}$ and $\mathbf{B}$ also have to be modified with the downsampled time step $\mathrm{d}tD$, and at each time step, the constraint matrices have to be expressed from the point of view of the current state (basically, at each timestep, we remove eight columns of $\mathbf{A}_{\mathcal{S}}$, since a state is composed of eight variables). From the result $\hat{\mathbf{u}}$, the first control command $\hat{\mathbf{u}}_0$ is extracted (the first four elements of $\hat{\mathbf{u}}$), and the next state is computed in the tangent space of the current state:

$$\begin{bmatrix} \mathrm{Log}_{\boldsymbol{\epsilon}_t}(\boldsymbol{\epsilon}_{t+1}) \\ \Gamma_{\boldsymbol{\epsilon}_{t+1} \to \boldsymbol{\epsilon}_t}(\dot{\boldsymbol{\epsilon}}_{t+1}) \end{bmatrix} = \mathbf{A} \begin{bmatrix} \mathrm{Log}_{\boldsymbol{\epsilon}_t}(\boldsymbol{\epsilon}_t) \\ \dot{\boldsymbol{\epsilon}}_t \end{bmatrix} + \mathbf{B}\hat{\mathbf{u}}_0,$$

and can be formulated in the manifold with the exponential map for the position, and the parallel transport for the velocity. Once the previous steps have been repeated for each time step, we end up with a planned downsampled orientation trajectory that still needs to be upsampled.

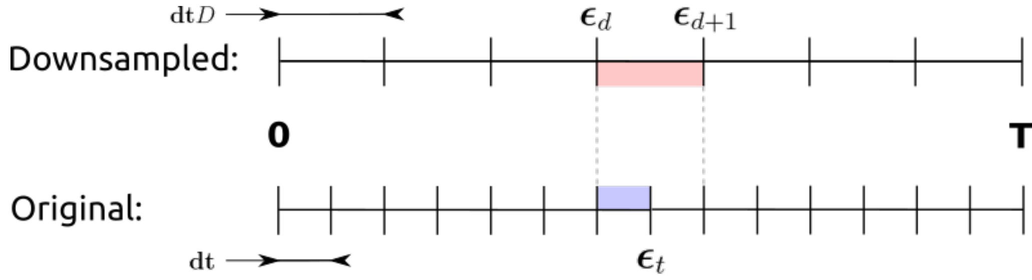
Orientation oversampling

Upsampling orientation data is not as simple as upsampling position data. Since the orientation at each time step is represented by a unit quaternion, and the representation is not part of an Euclidean space, it is impossible to use common interpolation approaches (spline or cubic) with these data.
The approach chosen in this project is to compute the time scale (at which percentage of the motion a state occurs) of the downsampled and desired trajectory.

Afterward, for each state $\boldsymbol{\epsilon}_t$ of the oversampled trajectory, we find two states $\boldsymbol{\epsilon}_d$ and $\boldsymbol{\epsilon}_{d+1}$ in the downsampled trajectory that occurs before and after the current time step, respectively. With these information, we can compute if $\boldsymbol{\epsilon}_t$ is closer to $\boldsymbol{\epsilon}_d$ or $\boldsymbol{\epsilon}_{d+1}$:

$$P = \frac{\mathrm{Time}(\boldsymbol{\epsilon}_t) - \mathrm{Time}(\boldsymbol{\epsilon}_d)}{\mathrm{Time}(\boldsymbol{\epsilon}_{d+1}) - \mathrm{Time}(\boldsymbol{\epsilon}_d)},$$

having P close to zero means that $\boldsymbol{\epsilon}_t$ is closer to $\boldsymbol{\epsilon}_d$ than $\boldsymbol{\epsilon}_{d+1}$. We can use this information to compute $\boldsymbol{\epsilon}_t$, and $\dot{\boldsymbol{\epsilon}}_t$ in the tangent space of $\boldsymbol{\epsilon}_d$ (see figure 3.2).



$$\mathrm{Log}_{\boldsymbol{\epsilon}_d}(\boldsymbol{\epsilon}_t) = \frac{\blacksquare}{\blacksquare}\,\mathrm{Log}_{\boldsymbol{\epsilon}_d}(\boldsymbol{\epsilon}_{d+1})$$

$$\Gamma_{\boldsymbol{\epsilon}_t \to \boldsymbol{\epsilon}_d}(\dot{\boldsymbol{\epsilon}}_t) = \dot{\boldsymbol{\epsilon}}_d + \frac{\blacksquare}{\blacksquare}\,(\Gamma_{\boldsymbol{\epsilon}_{d+1} \to \boldsymbol{\epsilon}_d}(\dot{\boldsymbol{\epsilon}}_{d+1}) - \dot{\boldsymbol{\epsilon}}_d)$$

Figure 3.2: Chosen approach to oversample an orientation trajectory.

At first glance, this oversampling method could look to a linear interpolation between two quaternions, but computing this interpolation in a tangent space leads to the smoothing of it when pushing the information back on the manifold (thanks to the shape of the manifold).

3.2 Tracking approach

Given the planned motions $\boldsymbol{\mu}_P$ for the position, and $\boldsymbol{\mu}_O$ for the orientation, the challenge of the tracking part is to make the robot accurately follow these two trajectories. It was not discussed in the planning part, but these two trajectories may not be compatible, it could be impossible for a robot at time t to be at position $\mathbf{x}_t$ with orientation $\mathbf{e}_t$. How to handle this problem will be discussed in section 3.2.3. We will first see how to track these two trajectories independently. In the planning part, we assumed that we were using a double integrator system, and that the single integrator system is a simplification of this system. Here, since the number of derivative in the system will influence the way to control the robot (a simple integrator system will be used to generate joint velocities, whereas double integrator system aims to generate torque control commands), both cases will be presented.

3.2.1 Position tracking

As discussed in chapter 2.2, there are several ways to perform a LQT. Some are more suitable for the planning (batch, QP formulation), others are more suitable for the tracking (DP, LQT/MPC formulation). Here, we will investigate two possible methods to solve the position tracking method:

1. A LQT/MPC solution, that directly link the trajectory defined in the task space, with control commands defined in the joint space.
2. A QP formulation of the LQT, that generate control commands in the task space, which will be transformed in the joint space with inverse kinematics or inverse dynamics (in function of how we want to control the robot).

The differences between these two approaches will be discussed in section 4, when presenting a real world example.

LQT/MPC solution

As stated before, the advantage of a LQT/MPC formulation to control a robot is that it directly links the task and joint spaces by overcoming the non-linearity in the $\mathbf{A}$ and $\mathbf{B}$ matrices.

Case 1: simple integrator system

$$\boldsymbol{\mu}_P = [\underbrace{\mathbf{x}_1^\top}_{\boldsymbol{\mathcal{S}}_1} \quad \cdots \quad \underbrace{\mathbf{x}_T^\top}_{\boldsymbol{\mathcal{S}}_T}]^\top,$$

$$\mathbf{x}_{t+1} = \underbrace{\mathbf{I}}_{\mathbf{A}} \mathbf{x}_t + \underbrace{\mathrm{dt}\mathbf{J}_\mathbf{t}}_{\mathbf{B}} \dot{\mathbf{q}}_t,$$

with $\mathbf{J}_\mathbf{t}$, the translational part of the Jacobian.

Case 2: double integrator system

$$\boldsymbol{\mu}_P = [\underbrace{\mathbf{x}_1^\top \quad \dot{\mathbf{x}}_1^\top}_{\boldsymbol{\mathcal{S}}_1} \quad \cdots \quad \underbrace{\mathbf{x}_T^\top \quad \dot{\mathbf{x}}_T^\top}_{\boldsymbol{\mathcal{S}}_T}]^\top,$$

$$\begin{bmatrix}\mathbf{x}_{t+1}\\ \dot{\mathbf{x}}_{t+1}\end{bmatrix} = \underbrace{\begin{bmatrix}\mathbf{I} & \mathrm{dt}\mathbf{I} + \frac{\mathrm{dt}^2}{2}\dot{\mathbf{J}}_\mathbf{t}\mathbf{J}_\mathbf{t}^\dagger \\ \mathbf{0} & \mathbf{I} + \mathrm{dt}\dot{\mathbf{J}}_\mathbf{t}\mathbf{J}_\mathbf{t}^\dagger\end{bmatrix}}_{\mathbf{A}} \begin{bmatrix}\mathbf{x}_t\\ \dot{\mathbf{x}}_t\end{bmatrix} + \underbrace{\begin{bmatrix}\frac{\mathrm{dt}^2}{2}\mathbf{J}_\mathbf{t}\mathbf{M}^{-\mathbf{1}} \\ \mathrm{dt}\mathbf{J}_\mathbf{t}\mathbf{M}^{-\mathbf{1}}\end{bmatrix}}_{\mathbf{B}} \boldsymbol{\tau}_t.$$

In this equation we assume that the robot is compensing the coriolis gravity and external forces effects, so the dynamics equation of the robot becomes:

$$\boldsymbol{\tau} = \mathbf{M}\ddot{\mathbf{q}} \rightarrow \ddot{\mathbf{q}} = \mathbf{M}^{-\mathbf{1}}\boldsymbol{\tau},$$

and can be linked to the task space with:

$$\ddot{\mathbf{x}} = \dot{\mathbf{J}}\dot{\mathbf{q}} + \mathbf{J}\ddot{\mathbf{q}} = \dot{\mathbf{J}}\mathbf{J}^\dagger\dot{\mathbf{x}} + \mathbf{J}\mathbf{M}^{-\mathbf{1}}\boldsymbol{\tau}.$$

These $\mathbf{A}$ and $\mathbf{B}$ matrices can be used with LQT/MPC (with a fixed horizon h), to generate the control commands at time step t, $\dot{\mathbf{q}}_t$ or $\boldsymbol{\tau}_t$ in function of the preferred way to control the robot.

Operational space control solution

In this solution, only one LQT is used. This LQT is instantiated at the beginning of the motion and solved with the DP formulation (better time complexity). It allows to compute at each time step a control command in function of the tracking error on the current state:

$$\mathbf{u}_t = \mathbf{K}_t(\boldsymbol{\mathcal{S}}_t - \hat{\boldsymbol{\mathcal{S}}}_t) + \mathbf{f}_t,$$

where $\hat{\boldsymbol{\mathcal{S}}}_t$ is the actual state at time step t.
This LQT is used to compute $\dot{\mathbf{x}}_t$ in case of a single integrator system, or $\ddot{\mathbf{x}}_t$ for a double integrator system. The control command is expressed in task space, thus it needs to be converted into its joint space consideration:

Case 1: single integrator system

$$\dot{\mathbf{q}} = \mathbf{J_t}^\dagger\dot{\mathbf{x}} \qquad \text{(inverse kinematics).}$$

Case 2: double integrator system

$$\boldsymbol{\tau} = \mathbf{J}^\top(\Lambda \begin{bmatrix}\ddot{\mathbf{x}}\\ \mathbf{0}\end{bmatrix} + \boldsymbol{\mu} + \mathbf{p}) \qquad \text{(operational space dynamics).}$$

In this case, the control command is augmented with zeros, which correspond to the desired angular acceleration (because it is not known at this point).

3.2.2 Orientation tracking

In chapter 2.3.3, we saw that there are two methods to plan or track orientation data, even if the time complexity of the single LQT method is quite interesting, it is a naive approach and must be forgotten.
It was not the case during the planning, but now the vector $\boldsymbol{\mu}_O$ is not sparse anymore. Thus we can use a fixed horizon H instead of windowing the vector from the current state to the last state. It will speed up the process.
The choice of the horizon size is subject to interpretation. A small horizon will speed up the process, but the advantages of using a MPC approach will be lost. In a sense, the LQT/MPC solution provides a kind of anticipation in the tracking by increasing the vision of the task at the current state, and with a small horizon we will lose this advantage. A large horizon will be slower to compute, but we will benefit from the anticipation capabilities of the formulation.
For orientation tracking, the priority should be put on the time complexity of the problem, thus the considered horizon will be between ten and twenty time steps. Furthermore, the benefit of anticipation for this use case is questionable, since the planning ensures to have a smooth trajectory between $\boldsymbol{\epsilon}_1$ and $\boldsymbol{\epsilon}_T$, there is no big changes to anticipate.
To track the orientation, we use a single or double integrator system with a LQT/MPC solution as presented in chapter 2.3.3.

Case 1: single integrator system
The control command at each time step is an angular velocity $\mathbf{w}_t$, and can be transformed into joint angle velocities with:

$$\dot{\mathbf{q}} = \mathbf{J_r}^{\dagger}\mathbf{w} \qquad \text{(inverse kinematics)},$$

with $\mathbf{J_r}$ the rotational part of the Jacobian.

Case 2: double integrator system
The control command is an angular acceleration $\dot{\mathbf{w}}$, and can be transformed into torques with:

$$\boldsymbol{\tau} = \mathbf{J}^{\top}(\Lambda \begin{bmatrix} \mathbf{0} \\ \dot{\mathbf{w}} \end{bmatrix} + \boldsymbol{\mu} + \mathbf{p}) \qquad \text{(operational space dynamics)}.$$

This time time the zeros corresponds to the linear acceleration, that is unknown at this point.

3.2.3 Merge position and orientation control commands

Until now, the position and orientation tracking were presented independently. However, there are only a few tasks in which we only need to control either the position or the orientation. Thus we need to have a way to fuse together the different control commands. This can occur at the task space level or at the joint space level.

For the task space level, it is done by stacking together the position and orientation commands in a vector. For example with inverse kinematics:

$$\dot{\mathbf{q}} = \mathbf{J}^{\dagger} \begin{bmatrix} \dot{\mathbf{x}} \\ \mathbf{w} \end{bmatrix},$$

we merge a task space linear and angular velocity into a joint angle velocity. By doing this, $\dot{\mathbf{q}}$ is the joint space control command corresponding to the desired linear and angular velocities. Similarly, with the operational space dynamics, we merge a task space linear and angular accelerations into a joint space torque command.
For the joint space level, it is done by summing the position command with the orientation command:

$$\dot{\mathbf{q}}_{tot} = \dot{\mathbf{q}}_{pos} + \dot{\mathbf{q}}_{orn},$$

or:

$$\boldsymbol{\tau}_{tot} = \boldsymbol{\tau}_{pos} + \boldsymbol{\tau}_{orn}.$$

It works because when computing the orientation or position command, we respectively set to zero the part relative to position or the orientation. Or only information about position or orientation is used (e.g., for the single integrator position or orientation tracking, we use either the translational or rotational Jacobian).
Until now, we assumed that at time step t, the desired position and orientation are compatibles together. But it is not always the case. Indeed, the task space of a robot is quite huge, if we only look at the possible positions, but constraining the orientation reduces a lot the size of the task space.
To overcome this problem, we need to prioritize either the position or the orientation. It can be done by projecting the secondary task into the nullspace of the primary task. To build the nullspace of a task, we use the Jacobian:

$$\mathbf{N} = \mathbf{I} - \mathbf{J}^{\dagger}\mathbf{J}.$$

Using the full Jacobian to build the nullspace will construct the nullspace of the global task, however, in our case, we are interested in constructing the nullspace of the position or the orientation only. For this, we can construct the nullspace from the rotational or translational Jacobian only. For example, to prioritize the position over the orientation:

$$\dot{\mathbf{q}}_{tot} = \dot{\mathbf{q}}_{pos} + \mathbf{N_t}\dot{\mathbf{q}}_{orn},$$

where $\mathbf{N_t}$ is the nullspace of $\mathbf{J_t}$.
The task determines which information should be prioritized. For example, in a grasping task, we would prefer to track the position accurately over the orientation, because having a bad position could lead to having an end-effector not at the position of the grasped object, but if the orientation is not as expected, the robot is more likely to still grasp the object.

Chapter 4

Experiments

In this chapter, the focus will be put on the experiments that have been developed to show the feasibility of the proposed approach. We will firstly focus on simulator experiments, and end this chapter with an example with real robots.

4.1 Simulator developments

The simulator aims to ensure that the developed algorithms and techniques are feasible and can be implemented on real robots without taking the risk to break them. It is why the experiments developed on the simulator will mainly be toy examples without any real purpose to validate the proposed approach.
Even if simulators aim to model the real world, their modelization is not hundred percent accurate, thus it is not because something work in the simulator that it will work in practice. In robotics, reducing the gap between simulation and practice is an active research topic, that is not solved yet[1].

Every experiments presented in this chapter are available online through jupyter notebooks : `https://gitlab.idiap.ch/jmaceiras/simulator_demonstration`.

4.1.1 The simulator

The chosen simulator for these experiments is the Pybullet physics simulator, which aims to be an easy-to-use Python module for robotics simulations. The robot is modelized through a URDF file, which is an XML format for representing a robot model.

[1] See `https://www.idiap.ch/en/scientific-research/projects/LEARN-REAL`

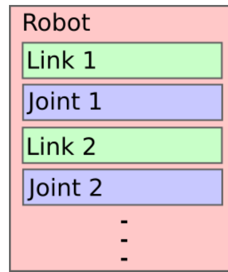


Figure 4.1: DOM schema of a standard URDF file. A robot is composed of one ore more links and joints.

3D objects can be attached to the links of a URDF file to have a visual representation of the robot in the simulator. The robot used in the simulator is the Panda robot from Franka Emika, which is the one also used in practice. In [16], the author gave us a way to compute the necessary information to build an accurate URDF model of the Panda robot.

4.1.2 The Python module

A brief introduction to the simulator is available at [17]. By taking a closer look to the architecture of the simulator, it is clear that this simulator contains a lot of functionalities, but unfortunately, they are not optimized for a robotics application, and their use are quite heavy. To solve this problem, it has been chosen to develop a Library Abstraction Layer (LAL), which aims to:

1. Harmonize the use of a robot for a robotics application.

2. Simplify the modification of the environment (easily remove/add/change robots or object).

3. Optimize the computation of some informations. By avoiding to recompute information that did not change.

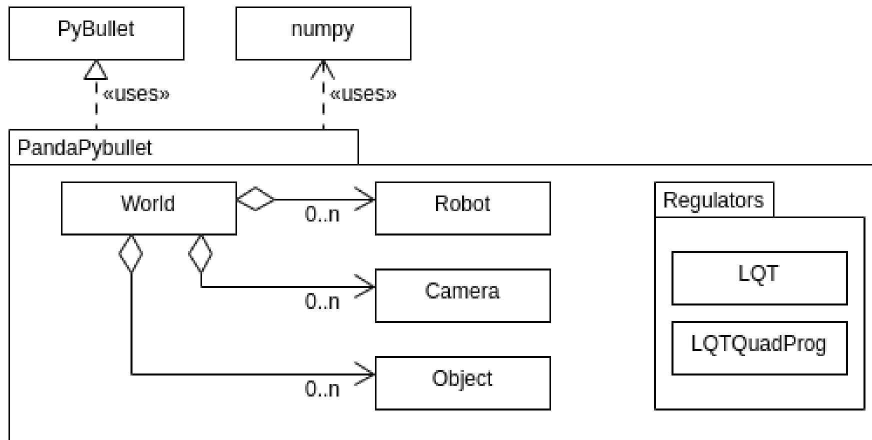


Figure 4.2: UML Class diagram of the LAL.

As pointed out by figure 4.2, the LAL breaks the simulator into four sub-parts:

1. A *World* class, which aims to assemble all functionalities of Pybullet that modify the simulation environment. It is the class that links everything together and determines how the environment will be. It is responsible for the gravity, the scene (basically the floor of the simulator), and is used to set up all physical elements that will be part of the simulation environment.

2. A *Robot* class, which has for purpose to be the abstraction of a robot in the simulator. The use of this class wants to be as close as possible of a normal robotics SDK to get information relative to a robot state (e.g., `robot.x()` give the end-effector position, `robot.q()` the joint angle positions,...).

3. An *Object* class, which is the abstraction of all phisical elements that are not robots (e.g., tables, balls,...). These object are not controlable, thus only their position in the world is interesting.

4. A *Camera* class, which allows to take images of the simulation. Cameras are attached to a robot link, and can take several kind of pictures of the simulator(depth image, RGB image, segmented image).

The *Regulators* package contains several regulators used in the project. They do not strictly use PyBullet, but are inserted in the LAL to facilitate the control of robots with it.
The main advantage of the LAL is that it is more friendly to use than the original PyBullet simulator. To illustrate this with an example, if we want to get the Jacobian of the robot, with only PyBullet the code is:

```
1   import pybullet as pb
2   import numpy as np
3
4   #===================================
```

```
5     # Set up the simulator
6     #====================================
7
8     states_infos = np.asarray(pb.getJointStates(robot_id, np.arange(7).tolist()))
9     q = states_infos[0,:]
10    dq = states_infos[1,:]
11    Jt, Jr = p.calculateJacobian(
12        robotId, 7, ee_transform, q.tolist(),
13        dq.tolist(), np.zeros(7).tolist()
14        )
15    J = np.vstack((Jt,Jr))
```

While with the LAL, the journey to get the Jacobian becomes:

```
1     import pandapybullet as pb
2
3     #====================================
4     # Set up the simulator
5     #====================================
6
7     J = robot.get_J()
```

As you can see, in pure PyBullet, to get one information, you need to pass to the simulator, all information that influences it. That is why to get the Jacobian, we need to give as parameters the joint angle positions and velocities (in theory velocities are not needed, but the simulator asks for them). Computing all these prerequisites is a waste of time. Thus, in the LAL these quantities are computed only when a perturbation on the environment occurs (when a control command is sent, or a collision occurs).

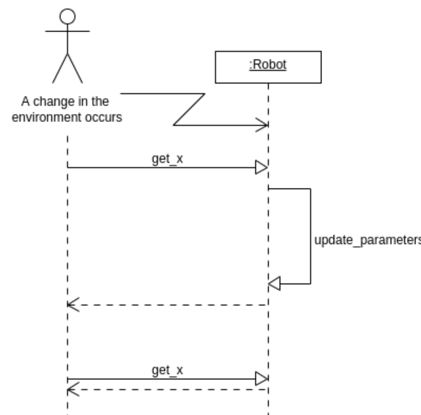


Figure 4.3: Sequence diagram which shows that an information about the robot state is computed only if the robot state has changed.

After a while of development, the LAL interested other members of the RLI group, who have joined the development of this library. The LAL is called PandaPybBullet and can be found here: `https://gitlab.idiap.ch/rli/pandapybullet`. A notebook presenting more in details the different functionalities of the library can be found on the repository.

4.1.3 Developments

Orientation planning and tracking on the manifold

This experiment aims to show an example on how to plan and track the orientation of the end-effector of a robot, the planning and the tracking are made through two LQT/MPC on the manifold. Even if for this use case it is possible to complete the experiment with only one LQT/MPC, which merges the planning and the tracking, to better represent a real example, it has been chosen to solve these two separately, because with only one LQT/MPC the time complexity to solve one time step would be bigger than the real duration of one time step (because of the sparsity of the definition of the task, the horizon of the controller would have to capture the rest of the motion at each time step). Furthermore, in this example the planned motion is not oversampled, since here, we only care about the orientation, we do not need to have a planned orientation trajectory of the same size than the position trajectory, and waiting a bit to compute the planning is not a problem in simulation.

> The code of this experiment is available through two jupyter notebooks on the gitlab of the simulator experiments: `https://gitlab.idiap.ch/jmaceiras/simulator_demonstration/-/tree/master/notebooks`, the notebook `Orientation_tracking_with_velocity_commands.ipynb` presents the inverse kinematics solution, and the notebook `Orientation_tracking_with_acceleration_commands.ipynb` presents the operational space dynamics solution.

In this experiments, a way to control the orientation in joint angle velocities and torque are presented. The end-effector starts at an orientation $\boldsymbol{\epsilon}_1$, and want to reach the orientation $\boldsymbol{\epsilon}_T$.

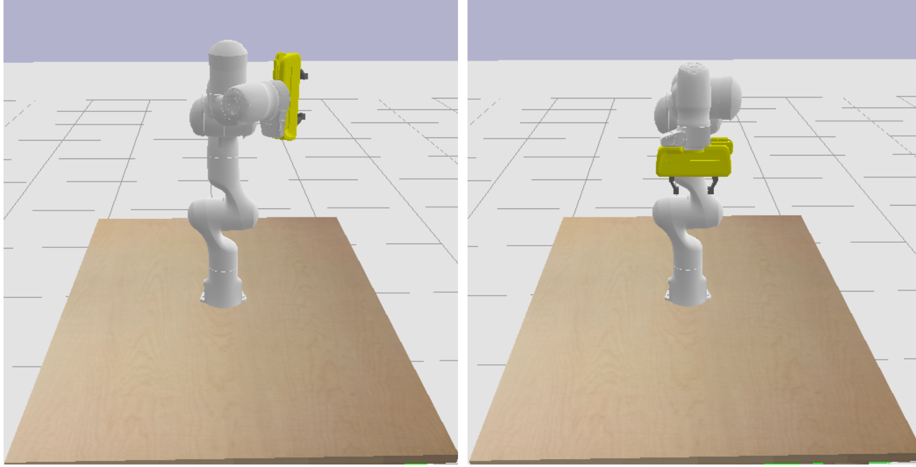


Figure 4.4: The left image represents the orientation of the end-effector at time step 0, and the right image the desired orientation at timestep T, since in this experiment, we only regulate the orientation of the end-effector, its final position may not be the same as the one in the right image. Basically, at the end of the motion, we want to have the end-effector of the robot pointing in the direction of the table.

With the initial, and final orientation presented in figure 4.4, a task of duration $T = 100$ time steps can be set up for the convenience of this example, we will assume that no constraints are needed to perform the tracking. So, the task can be defined with the sparse vector $\boldsymbol{\mu}$:

$$\boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\epsilon}_1 \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \boldsymbol{\epsilon}_T \end{bmatrix} \in \mathbb{R}^{4T},$$

for a single integrator system, or:

$$\boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\epsilon}_1 \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{0} \\ \boldsymbol{\epsilon}_T \\ \mathbf{0} \end{bmatrix} \in \mathbb{R}^{8T},$$

for a double integrator system, where we want to track a desired position, and a desired angular velocity. In the $\boldsymbol{\mu}$ vector, the angular velocity is represented by the quaternion time-derivative. That is why the dimension of $\boldsymbol{\mu}$ is twice bigger for the double integrator system.

Now that we have a very simple idea of our task, we need to fill the missing information in the $\boldsymbol{\mu}$ vector (currently, we know only the first and final state). For this, we use a first LQT/MPC regulator, that will solve the planning problem by proposing a trajectory to link the initial and final states.

In this case, it is difficult to visualize, the planned trajectory, since the orientation is defined as a quaternion in $\mathbb{R}^4$. A possible way to visualize this trajectory is to transform each orientation into a rotation matrix, and visualize the evolution of the end-effector frame by plotting its axis in $\mathbb{R}^3$.

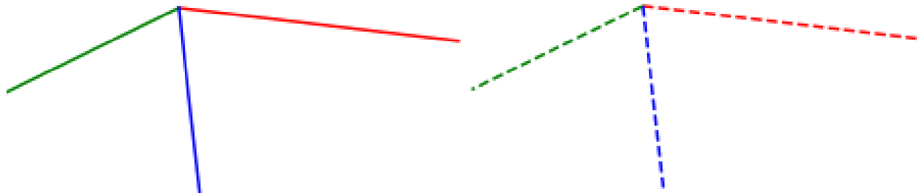


Figure 4.5: The left image represents the desired final orientation of the end-effector frame, and right image represents the planning solution to the final orientation of the end-effector frame. As you can see, these two orientations are the same. This is a visual test to see if the planning worked as expected.

The method presented in figure 4.5 is a quick visual test to see if the planning worked as expected, but it is subject to the visualization of the user. As a more mathematical method, we can measure how much the predicted final state $\hat{\boldsymbol{\epsilon}}_T$ is similar to the desired final state $\boldsymbol{\epsilon}_T$ with the distance function presented in chapter 2.3: $d(\boldsymbol{\epsilon}_T, \hat{\boldsymbol{\epsilon}}_T)$, where a result close to zero means that the two orientations can be assumed to be the same.

With the planning done, it is now possible to make the end-effector of the robot track the desired orientation over time. For this, another LQT/MPC regulator is used, since $\boldsymbol{\mu}$ is not sparse anymore, the horizon of the regulator does not need to capture the entire motion at each time step. Thus, with a smaller horizon, the time to compute one time step will be reduced. The returned control command will depend on the structure of the $\mathbf{A}$ and $\mathbf{B}$, for a single integrator system, the control command will be a quaternion velocity $\dot{\boldsymbol{\epsilon}}$, and a quaternion acceleration $\ddot{\boldsymbol{\epsilon}}$ for a double integrator system. These control commands can be transformed into angular velocity or acceleration with the formulas presented in section 2.3.

Similarly to what has been made to ensure the smooth running of the planning, to validate the tracking, we can either visualize the resulting orientations or use the distance function.

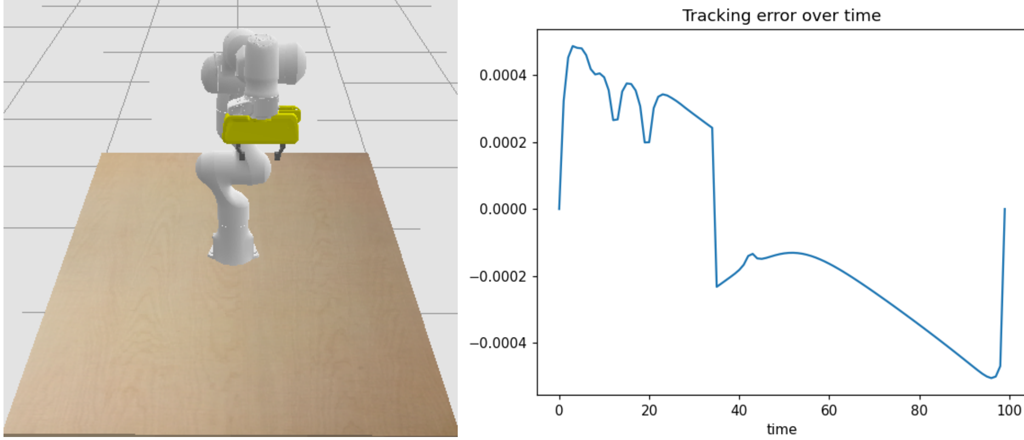


Figure 4.6: The left image represents the resulting final orientation of the tracking, as yon can see, the orientation of the end-effector is similar to the desired one presented in figure 4.4. To validate this first assumption, the right image shows the tracking error evolving over time, with an average error about `1e-4`, we can assume that the tracking performed as expected.

Figure 4.6 shows the tracking result with a single integrator solution (i.e., with inverse kinematics to retrieve the corresponding joint angles velocities). The solution using the operational space dynamics to retrieve the torque produces a bad result, after some investigations, this bad result is due to the structure of the inertia matrix reported to the end-effector $\boldsymbol{\Lambda}$ used in the equation:

$$\boldsymbol{\mathcal{F}} = \boldsymbol{\Lambda} \begin{bmatrix} \ddot{\mathbf{x}} \\ \dot{\mathbf{w}} \end{bmatrix} + \boldsymbol{\mu} + \mathbf{p}.$$

In chapter 2, $\boldsymbol{\mathcal{F}}$ was presented as a vector of forces and wrenches occurring on the end-effector, but this vector can clearly be decomposed into:

$$\boldsymbol{\mathcal{F}} = \begin{bmatrix} \mathbf{F} \\ \mathcal{T} \end{bmatrix},$$

with $\mathbf{F} \in \mathbb{R}^3$ the forces applied on the end-effector, which are mostly responsible for the change in position, and $\mathcal{T} \in \mathbb{R}^3$ the wrenches applied on the end-effector, which are mostly responsible of the change in orientation.

By visualizing $\boldsymbol{\Lambda}$ at a given time, the last third rows, which are responsible to create $\mathcal{T}$, contain value close to zero, thus the resulting wrenches are small, and does not allow to correctly track the orientation with torque commands. A quick hotfix for this problem is to replace $\boldsymbol{\Lambda}$ with an identity matrix. Thanks to the regulator it works but gave worst results than the inverse kinematics solution.

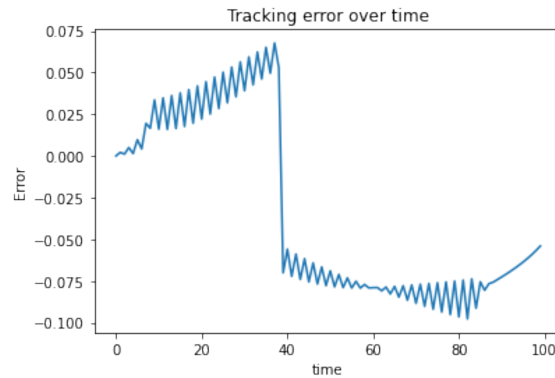


Figure 4.7: Tracking error over time for the operational space dynamics with an identity matrix solution. As you can see, this trick allows to keep an acceptable tracking error over time (even if it performs worst than the IK solution). By modifying $\boldsymbol{\Lambda}$ the control command does not respect the dynamics of the robot anymore, thus it adds a sawtooth perturbation to the error signal, that can leads to the instability of the regulator.

Position planning and tracking

This experiment aims to show an example of how to plan and track the position of the end-effector of two robots. The planning is made through a constrained LQT, that will constrain the two robots to meet at the middle of their motion. For the tracking, two solutions are explored, the LQT/MPC solution, which directly returns a control command in torque, and a dynamic programming LQT, which returns a Cartesian acceleration that has to be transformed into torque through the operational space dynamics. In this experiment, only the double integrator case is explored.

> The code of this experiment is available through two jupyter notebooks on the gitlab of the simulator experiments: `https://gitlab.idiap.ch/jmaceiras/simulator_demonstration/-/tree/master/notebooks`, the notebook `Position_tracking_MPC_solution.ipynb` presents the LQT/MPC solution, and the notebook `Position_tracking_OS_solution.ipynb` presents the operational space dynamics solution.

In this experiment, robot one starts from a position $\mathbf{x}_{1,1}$ with zero speed and want to reach position $\mathbf{x}_{1,T}$ with zero speed, similarly robot two starts at $\mathbf{x}_{2,1}$ and ends

at $\mathbf{x}_{2,T}$. At this point, the $\boldsymbol{\mu}$ vectors for each robot are defined as:

$$\boldsymbol{\mu}_1 = \begin{bmatrix} \mathbf{x}_{1,1} \\ \mathbf{0} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{x}_{1,T} \\ \mathbf{0} \end{bmatrix}, \boldsymbol{\mu}_2 = \begin{bmatrix} \mathbf{x}_{2,1} \\ \mathbf{0} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{x}_{2,T} \\ \mathbf{0} \end{bmatrix} \in \mathbb{R}^{6T},$$

with a state at a given time defined by the position and velocity (only linear) of the end-effector. Since we want to control the two robots with one regulator, we have to merge these two vectors:

$$\boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mathcal{S}}_{1,1} \\ \boldsymbol{\mathcal{S}}_{2,1} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \boldsymbol{\mathcal{S}}_{1,T} \\ \boldsymbol{\mathcal{S}}_{2,T} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_{1,1} \\ \mathbf{0} \\ \mathbf{x}_{2,1} \\ \mathbf{0} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{x}_{1,T} \\ \mathbf{0} \\ \mathbf{x}_{2,T} \\ \mathbf{0} \end{bmatrix} \in \mathbb{R}^{2 \cdot 6T},$$

where $\boldsymbol{\mathcal{S}}_{i,j}$ denotes the position and velocity of robot i at time step j. Thus, the state of the global system at time step t is given by:

$$\boldsymbol{\mathcal{S}}_t = [\boldsymbol{\mathcal{S}}_{1,t}^\top \quad \boldsymbol{\mathcal{S}}_{2,t}^\top]^\top.$$

The advantage of building $\boldsymbol{\mu}$ like this is that it makes easier the construction of the $\mathbf{A}$ and $\mathbf{B}$ matrices for the global system. Given the linear system equations of the two robots:

$$\boldsymbol{\mathcal{S}}_{1,t+1} = \mathbf{A}_1 \boldsymbol{\mathcal{S}}_{1,t} + \mathbf{B}_1 \mathbf{u}_{1,t},$$
$$\boldsymbol{\mathcal{S}}_{2,t+1} = \mathbf{A}_2 \boldsymbol{\mathcal{S}}_{2,t} + \mathbf{B}_2 \mathbf{u}_{2,t},$$

with $\mathbf{A}_{\{1,2\}}$ and $\mathbf{B}_{\{1,2\}}$ being the same for each robot (double integrator system). The linear system equation for the global system becomes:

$$\boldsymbol{\mathcal{S}}_{t+1} = \begin{bmatrix} \boldsymbol{\mathcal{S}}_{1,t+1} \\ \boldsymbol{\mathcal{S}}_{2,t+1} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_2 \end{bmatrix} \boldsymbol{\mathcal{S}}_t + \begin{bmatrix} \mathbf{B}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2 \end{bmatrix} \begin{bmatrix} \mathbf{u}_{1,t} \\ \mathbf{u}_{2,t} \end{bmatrix}.$$

Similarly to what has been made in the previous experiment, a constrained LQT will be used to plan the trajectories of the two robots, and force them to meet in

the middle of their motion. The meeting is defined by the equation:

$$\boldsymbol{\mathcal{S}}_{1,T/2} - \boldsymbol{\mathcal{S}}_{2,T/2} = \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix},$$

where $\mathbf{b}$ is a small distance to avoid a collision between the robots. In this constraint, we are not interested in constraining the velocity, so we have to carefully build the $\mathbf{A}_{\mathcal{S}}$ matrix and $\mathbf{b}_{\mathcal{S}}$ vector. With:

$$\mathbf{A}_{\mathcal{S}} = \begin{bmatrix} \mathbf{0} & \cdots & \mathbf{I} & \mathbf{0} & -\mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} \end{bmatrix},$$

and:

$$\mathbf{b}_{\mathcal{S}} = \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix},$$

we build a constraint such that:

$$\mathbf{A}_{\mathcal{S}}\mathbf{x} = \mathbf{b}_{\mathcal{S}} \rightarrow \mathbf{x}_{1,T/2} - \mathbf{x}_{2,T/2} = \mathbf{b},$$

with $\mathbf{x}$ a prediction of $\boldsymbol{\mu}$, and $\mathbf{x}_{i,j}$ the position of robot i at time step j.
With these informations it is possible to build a constrained LQT to solve the planning part, because of the sparsity of $\boldsymbol{\mu}$, the $\mathbf{Q}$ matrix of the regulator has to be adapted to have ones in the diagonal only for defined states (i.e., ones in the first and last twelve elements of the diagonal).

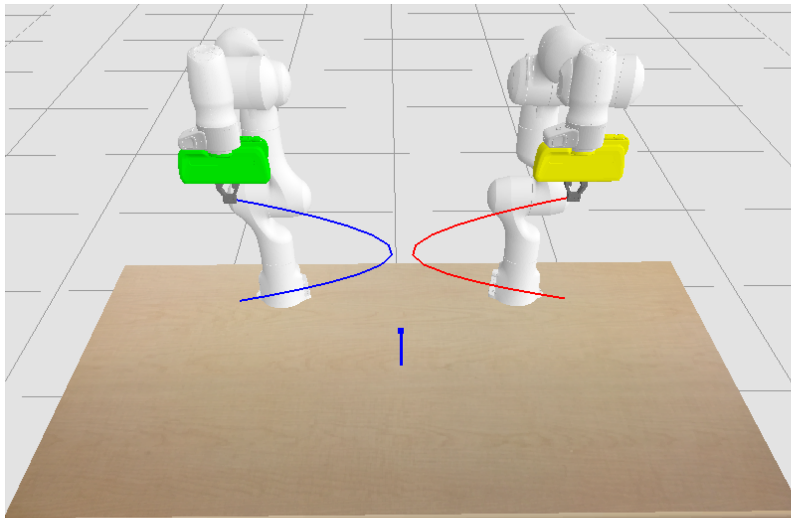


Figure 4.8: Result of the planning part, where a trajectory for each robot is generated. As you can see, the planner chooses a meeting point for the two robots with a small distance to avoid collision.

In the experiment, the planner solves a downsampled version of the motion to fast up the resolution of the problem. The upsampling is made through a multidimensional cubic interpolation (in this example, a cubic interpolation was used instead of spline by convenience).

Now that the vector $\boldsymbol{\mu}$ is fully defined, the robot will be able to track the motion. As stated before, two different manners will be explored to track the desired position. For the dynamic programming solution, we will reuse a double integrator system for the $\mathbf{A}$ and $\mathbf{B}$ matrices, and get at each time step an acceleration command that is built in function of the trajectory and the current tracking error. This control command is transformed into torque with the operational space dynamics equation. With this solution, we can think that we are doing exactly the same thing as in the planning part, and therefore the planning and tracking can be made in one step, with a single constrained LQT. But even if in theory it could be true, it is not the case for two particular reasons:

1. The time to solve a constrained LQT is too high, if in the simulator robots move only when you send them a control command. In practice, the robot will expect a control command at a fixed rate, and if you do not give it to them, they will generate an error and stop their motion. As a fix around to this problem, we could imagine to send them several times the same control command to match our downsampled constrained LQT, but from the personal point of view of the author, it would be like riding a Ferrari at 50 km/h, it is possible, but nobody wants to do it.

2. The biggest advantage of the LQT/MPC or the dynamic programming solution of the LQT is that it automatically corrects the tracking error. With these solutions, if you push a robot during its motion, the robot will try to come back to the desired position, but with a constrained LQT, if you move the robot during the motion, it will not realize and will continue its motion as if nothing had happened.

With the LQT/MPC solution, we use the formulation presented in section 3, and solve the problem over a fixed horizon for each time step. Since the regulator directly returns a joint torque command, it can directly be sent to the robots.

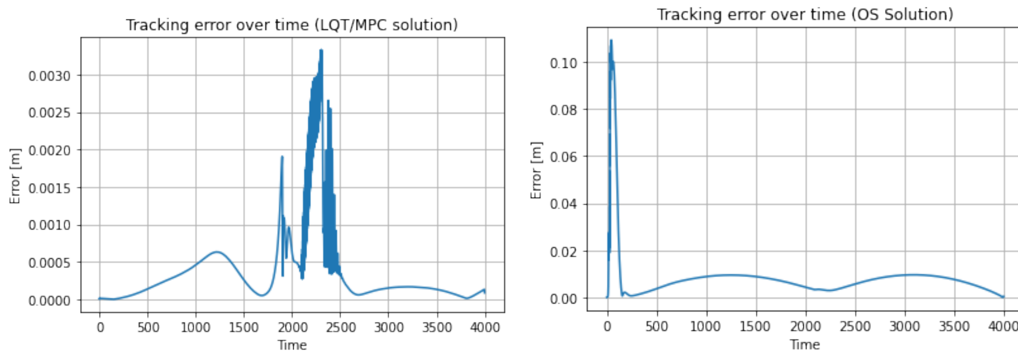


Figure 4.9: Left image shows the tracking error for the LQT/MPC approach, and right image shows the tracking error for the operational space dynamics approach.

As you you can see in figure 4.9, the tracking error with the LQT/MPC solution is lower than the operational space dynamics solution (median error for LQT/MPC: 0.1

[mm], median error for OS solution: 7 [mm]). The perturbation on the LQT/MPC plot between time step 2000 and 2500 is due to a bad configuration of the joints of the robot at this moment (the robot is in a bad position, thus it can not move its joints as desired by the controller, and the desired task space acceleration is not guaranteed). The perturbation at the beginning of the OS solution error plot is due to the fact that the DP solution of the LQT, works as a proportional regulator, and if at one point the error is zero (which is the case at the beginning of the motion), the control command will be close to zero and thus affect the accuracy of the regulator at the beginning.

In this case, the horizon of the LQT/MPC solution is set to ten time steps. Since in the simulator the robot is waiting for a control command, we do not have to bother about the time complexity for one time step of the controller, but in practice, we would have to find a good compromise between the time complexity, and the accuracy of the regulator. With the general rule that the larger the horizon is, the more the tracking will be accurate, and the more the time complexity to solve the problem will increase.

By analyzing the result presented in figure 4.9, it is clear that for the simulator, the LQT/MPC solution is probably the best. The median tracking error of the operational space is too high to ensure that all tasks will be correctly performed. Instead of vanishing the operational space dynamics solution, since the model of the robot is only an approximation of the real one, it is preferable to keep this solution and wait for the experiments on real robots to make a choice. Since the URDF of the robot is only an approximation of its real model, we can not exclude that something works or does not work, because of this approximation.

4.1.4 Conclusion of the simulator experiments

These experiments aimed to give a first validation for the approach presented in section 3. To perfectly match the presented approaches an experiment of position tracking with a single integrator is missing. It has been made on purpose since this approach is almost the same as its orientation counterpart (except that we use the translational Jacobian instead of the rotational Jacobian), and inverse kinematics is a widely used technique, that is known to work well. At this point, there are no approaches that are considered as unreliable, thus all of them will be tested in practice, and the result of the experiments made in practice will say which approaches to choose.

4.2 Real example

During the development of this project, I had the opportunity to develop a real robotics application proposed by «Les Fondues Wyssmüller», a famous Swiss fondue maker. To surf on the buzz of the raclette robot[2], they want to develop a project with robots that cook a fondue. The overall purpose of this project is not to replace the human work behind the preparation of fondue, but to increase the visibility of the company during events, and by the way, promote the work of the robotics group of the Idiap research institute.
That is why in this chapter, you will not only see the development on real robots of the methods presented in chapter 3, but also other developments that are needed for this project.
The development is made on the two Panda robots available at the Idiap Research Institute, which allow to perform robotics bi-manual tasks.

4.2.1 Specifications and first analysis

As stated before, the purpose of the project is to make robots prepare a fondue, preparing the fondue does not only mean make the cheese melt, but robots should respect the recipe given by the producer:

1. Put wine in the fondue bowl (0.7 dl per person).
2. Warm the wine until it reaches 73 degrees (Celsius).
3. Add the cheese mix in the fondue bowl (200g per person).
4. Brew the fondue until the mix reaches 73 degrees (temperature at which the cheese is assumed to be melted).
5. Serve the fondue.

For this project, robots will prepare each time a fondue for two persons, and should not need the help of human interactions.
From the recipe presented above, a first analysis can be made. It is clear that the temperature of the mix plays an important role in the preparation. A too high temperature will burn the fondue, and add an undesirable taste, thus a waterproof temperature sensor needs to be developed. Furthermore, during the task, robots will grasp various objects with a really different shape, thus the grippers should be adapted to the different objects that are present in the task.
At first glance, it would be impossible for a robot to put accurately 1.4 dl of wine in the fondue bowl. Hopefully, the fondue producer has 1.4 dl white wine bottles that he uses exclusively for the fondue, and fondue mix are sold in 400g packet, thus robots do not have to measure any quantities.

[2] Avideopresentingtherobocletteproject:https://www.youtube.com/watch?v=a6KpHR6jkSc

The description of the process above allows us to have a first idea of the project, thus we can already imagine the setup of the project and the tasks of each robot.

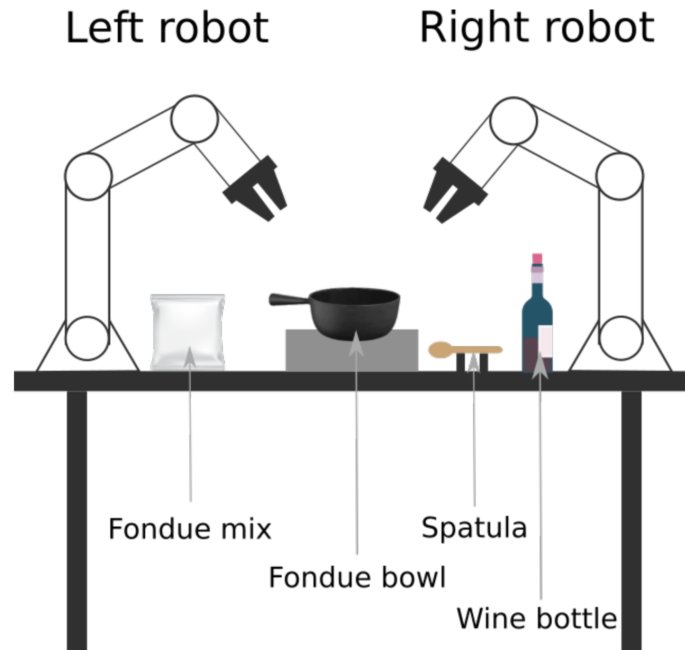


Figure 4.10: Visualization of the setup for the project.

Figure 4.10 presents the imagined setup of the project. The position of each element will influence which robot should perform which task. With the presented setup, the tasks are gathered as follow:

1. Left robot is responsible to put the fondue mix in the fondue bowl, and to maintain and move the fondue bowl.

2. Right robot is responsible to add the wine, and to brew the fondue with the spatula.

With this division of labor, it is now possible to imagine specific grippers for each robot, that will be helpful to perform their own tasks. In this setup, the temperature sensor was not mentioned, after a bit of thinking, the best solution for this sensor is to create a custom support that will be clipped in the fondue bowl by a robot. The support will be inspired by the design of a hairpin to add and remove the sensor from the bowl easily.

4.2.2 Mechanical developments

All the mechanical developments presented in this chapter are 3D Models made with Autodesk Inventor, and 3D printed in PLA. Plans of the gripper connector for the Panda robots are available online in the Panda Manual. The homemade grippers will be connected to the robot through this connector.

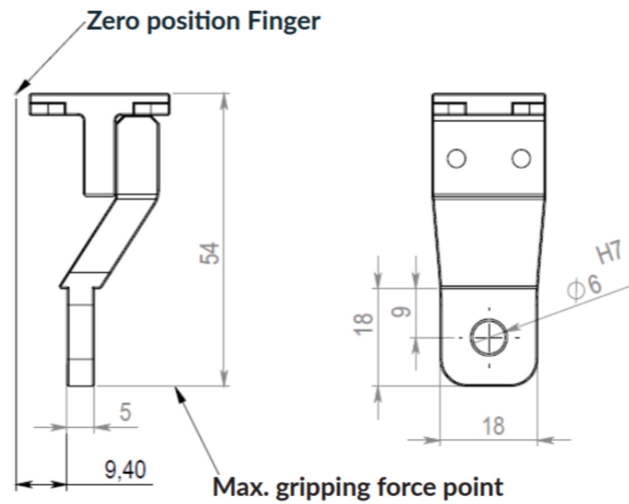


Figure 4.11: Plan of the Panda gripper physical connector.

At the moment, not all pieces have been 3D printed or modelized, the focus has been set on the robotics part of the project, thus only the mechanical vital parts of the project have been created.

The 3D files are available in Autodesk or .obj format at `https://gitlab.idiap.ch/jmaceiras/fonduebot-3d-pieces`.

Left robot gripper

As stated before, the left robot gripper has to grab the fondue mix and the fondue bowl. These tasks are complete opposite, in the first one the robot has to grab the mix enough to lift and serve it, but not too much, otherwise, the mix will not fall into the fondue bowl. For the second task, the gripper needs to strongly grasp the fondue bowl to displace or maintain it during the brew. Since the handle of the fondue bowl is not straight, the task of grasping it correctly is more complicated.

Figure 4.12: Image of the fondue bowl, as you can see the handle is not straight, and it could lead to difficulties for correctly grasping it.

The chosen approach was to design a hybrid gripper with one part that can correctly grab the fondue mix, and the other part that can correctly grab the fondue bowl, this part also will marry the shape of the handle to correctly grasp it.

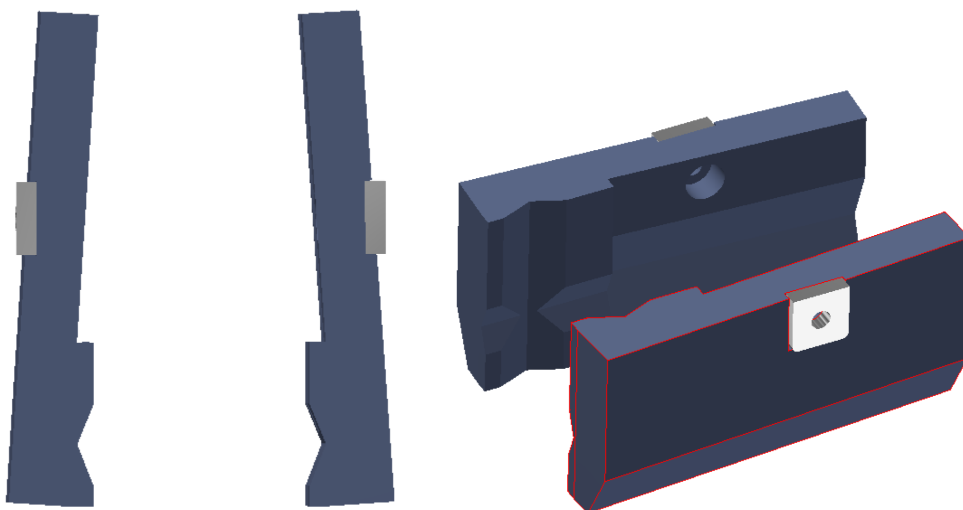


Figure 4.13: 3D model of the left gripper. On the left image, you can see that the top part of the gripper aims to grab the fondue bowl, by marrying the shape of the handle. The bottom part of the gripper is a normal gripper to grasp casual objects.

Figure 4.14: Test with the 3D printed left gripper, it can correctly grasp and lift the bowl.

Right robot gripper

The right robot gripper has to grab and serve the wine bottle, and to grab the spatula to brew the mix. These tasks have the advantage of being compatible together, so that a normal grasping gripper can be designed and used for both tasks. This gripper should just be sufficiently big to grasp the wine bottle and the spatula correctly.

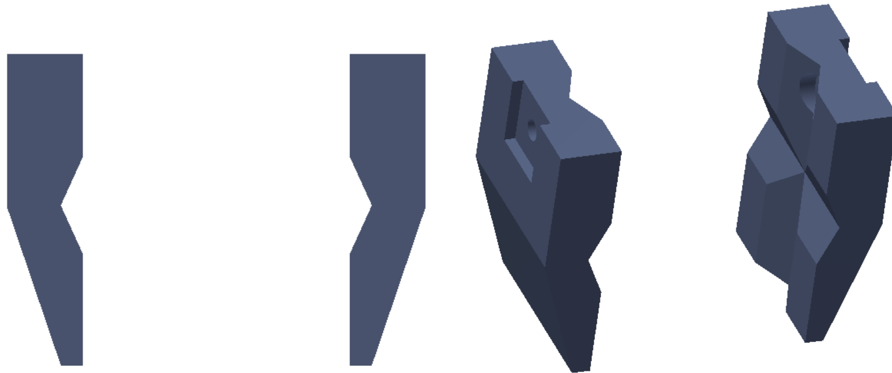


Figure 4.15: 3D model of the right gripper. This gripper is made to grasp objects horizontally or vertically.

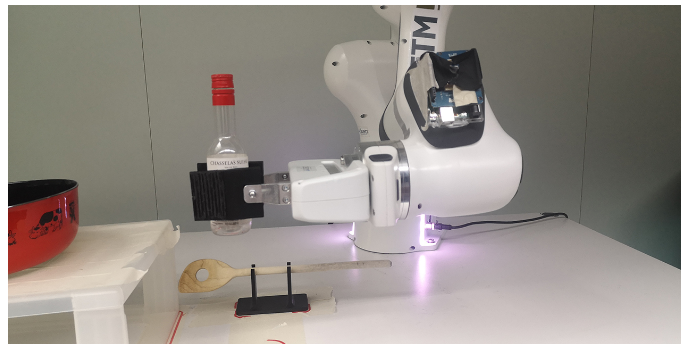


Figure 4.16: Test with the 3D printed right gripper, where it grasp the wine bottle

Figure 4.17: Test with the 3D printed right gripper, where it grasps the spatula. In this picture, you can also see the left robot maintaining the fondue bowl.

Spatula support

Since a robot hand does not have the same dexterity of a human hand, the spatula can not lie on the floor of the table, it has to be put in a position where a robot can easily come and take it. That is the purpose of this support, it raises the spatula to let a robot grab it.

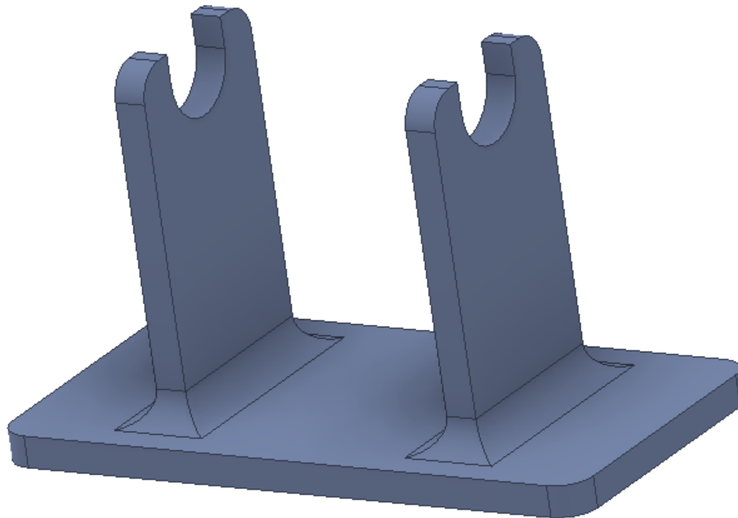


Figure 4.18: 3D model of the spatula support.

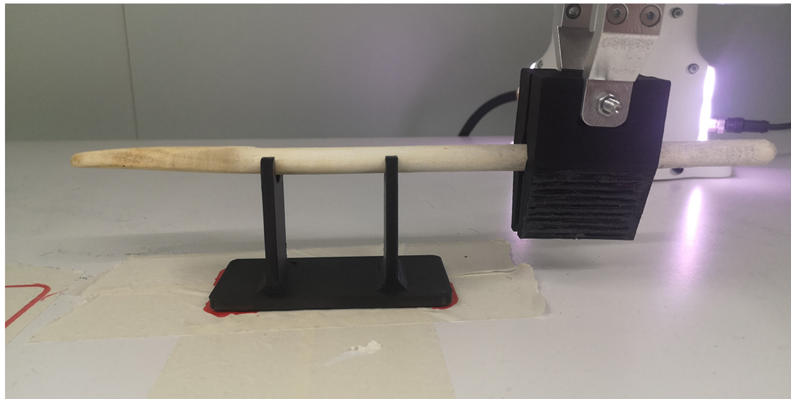


Figure 4.19: 3D printed spatula support with the spatula and the robot that is grasping it.

4.2.3 Temperature sensor development

As stated before, the temperature sensor needs to be waterproof, and measure a temperature between the ambient temperature (assumed to be around 20°C) and 73°C. For this use case, an accuracy around the degree is assumed to be sufficient. The temperature sensor is connected to the controller of the robots (a PC), and its value should be read as wanted.

After investigating an all-in-one solution that would need no specific developments, this track has been abandoned, all sensors found were either expensive (around 200 CHF), either not waterproof.

The chosen solution was to develop a homemade temperature sensor from different components that we can buy online, that is to say:

- A temperature sensor, the choice has been made on a PT1000 waterproof temperature sensor, which has an internal resistor that vary precisely with the temperature.

- An amplifier to correctly read the output voltage of the sensor (ref: MAX31865).

- A micro-controller that read the voltage value, transform it into its temperature conterpart, and send it to the controller (ref: Arduino Nano).

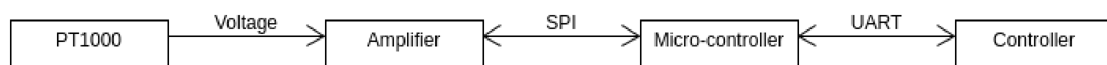


Figure 4.20: Schema on how the different parts of the sensor are connected together

As pointed out by figure 4.20, the voltage at the terminals of the sensor is read by the amplifier and sent to the micro-controller through SPI (Serial Peripheral Interface), which allows to send information between two peripheral at a high bit rate. The micro-controller communicates with the controller through UART (Universal Asynchronous Receiver Transmitter).

The micro-controller does not continually read and send the temperature value, it is waiting for a `GET` command from the controller.

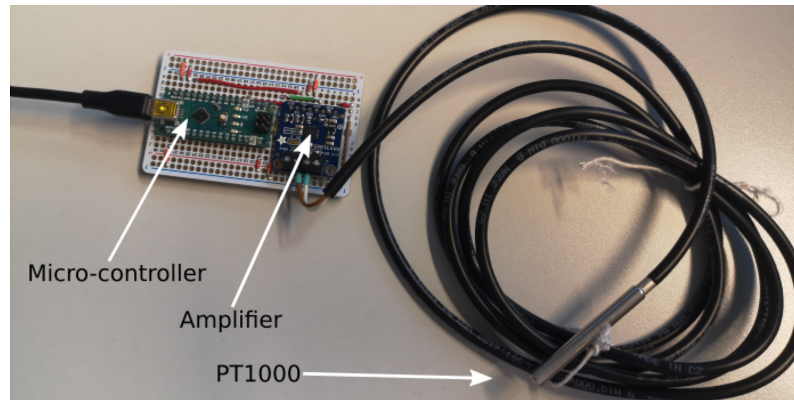


Figure 4.21: Developed temperature sensor

The evaluation of the accuracy of the sensor was a bit more challenging since no accurate ground truth was available, the evaluation has to be made with another temperature sensor that has an unknown accuracy as ground truth.

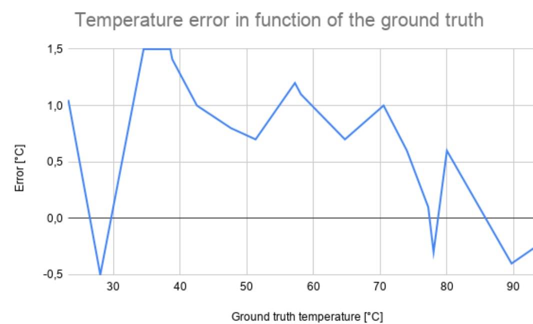


Figure 4.22: Accuracy of the temperature sensor (average error: 0.63 °C).

As you can see in figure 4.22, The temperature difference between the two sensors are almost always below the degree (desired accuracy). These measures have been made by submerging the two sensors into water which is warmed between twenty and ninety degrees, they come from two independent measure sets. Due to the lack of knowledge on the ground truth sensor, we can only assume that the accuracy of the developed sensor is good. Furthermore, in these measures, we assumed that heat propagates the same way in the two sensors, then the result of these measurements must be analyzed with caution. We will just assume that the accuracy of our sensor is enough for our task.

The code of the temperature sensor is available at `https://gitlab.idiap.ch/jmaceiras/wyssmuller/-/tree/master/temp_sensor/temp_sensor`. Thanks to the manufacturer of the amplifier, the code is really small (they provide a library to read a PT1000 temperature from their amplifier). The Arduino IDE is needed to run the code on a micro-controller.

4.2.4 Robotics developments

The scope of this chapter consists in the implementation and tests of the techniques presented in chapter 3. The experiments to test these techniques are oriented to the robot fondue project in order to see if it is possible to use them in a concrete robotics application. The setup used in these experiments is the setup presented in figure 4.10, that has been created.

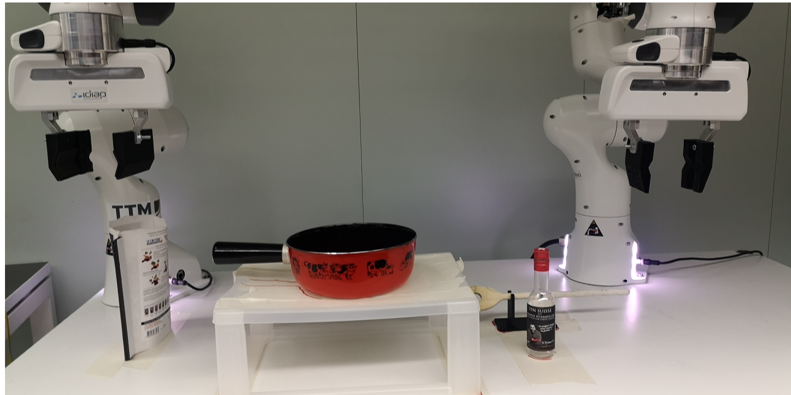


Figure 4.23: Real setup inspired by figure 4.10

The code is available at: `https://gitlab.idiap.ch/jmaceiras/wyssmuller`

Environment

Usually, in the industry, there are two different ways to program a robot, either use an SDK provided by the manufacturer or use ROS (Robot Operating System). ROS is an open-source set of tools that aims to facilitate the development of robotics applications. Nowadays, all serious robots are compatible with ROS, but manufacturers continue to provide an SDK in order to give the user an alternative to ROS. It is based on the publisher-subscriber architecture, that allows a node (an executable) to read or write values in the desired topics.

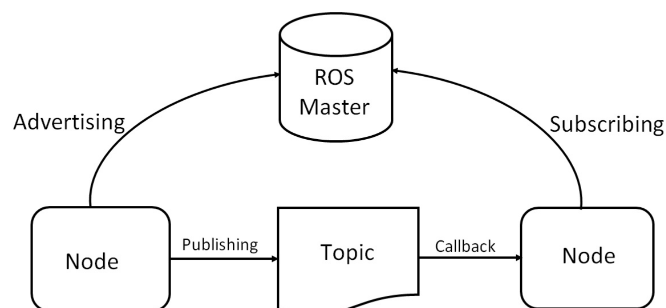


Figure 4.24: Schema representing the architecture of ROS.

In these experiments, instead of using ROS, the development will be made with the help of the SDK provided by Franka Emika: `libfranka`. The main reason behind this choice is that the group wants to know if `libfranka` can be a viable alternative to ROS or not. Furthermore, ROS can be very helpful if we want to include special sensors in the application easily (e.g, a camera, laser, ...), but for a pure robotics application, the way to use it is a bit heavy.
`libfranka` is a C++ SDK that allows to read and control a Panda robot. It manages the communications with the robot and provides interfaces to:

- Execute non-realtime commands.
- Execute realtime commands (basically control commands).
- Read in real time the state of a robot.
- Access the model of the robot to compute our desired kinematic or dynamic parameters.

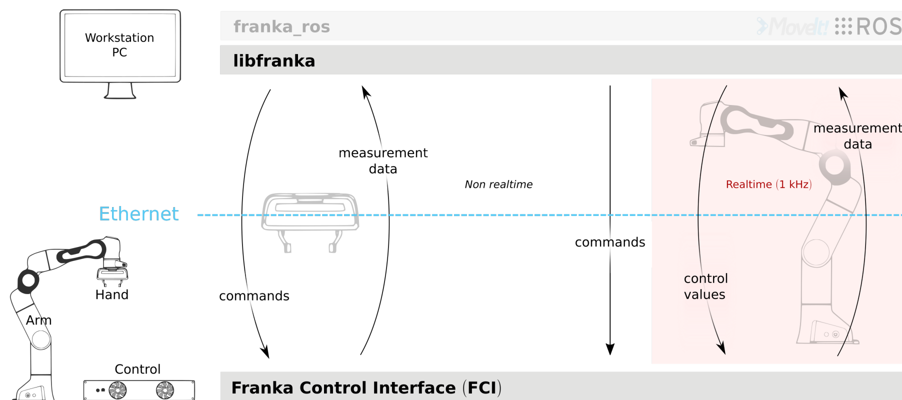


Figure 4.25: Architecture of `libfranka`, source: `https://frankaemika.github.io/docs/libfranka.html`.

As you can see in figure 4.25, the use of `libfranka` is divided into a real-time and non-real-time part. The non-realtime commands are blocking commands sent to the robot. They encompass all commands related to the end-effector or to the configuration of the robot (collision behavior, internal controllers gains,...). The realtime part of `libfranka` is related to the control of the robots. When controlling the robots, a callback is called every millisecond, and the SDK expects to have a control command to send to the robot. If the computation of the control command takes longer than one milliseconds (in practice, the control command still have to be transmitted through ethernet, so we should not take longer than about 0.7 milliseconds to compute a command), the control loop will be broken, and the robot will throw an error that will kill the connection.

Realtime method commands are UDP based (TCP for non-realtime commands), and their methods are deterministic in time to not violate the one millisecond constraint. They are related to the control of the robot but also to the lecture of the robot state. The SDK offers lots of possibilities, but in the frame of this project, the use of `libfranka` can be summed up in five classes:

- A `Network` class that is responsible of the connection between the computer and the robot.
- A `Robot` class which is the abstraction of the Panda robot. This class is used to start a control loop, or to get the current state or model of the robot. The control loop can accept torque, joint velocities, joint positions, end-effector pose as control commands.
- A `Gripper` class which is the end-effector abstraction of the robot. This class contains only non-realtime commands (homing, grasping of the gripper).
- A `Model` class that is used to generate the kinematic or dynamic parameters of the robot. These parameters can be computed with the current or an arbitrary robot state.
- A `RobotState` class that contains the information of the current state of the robot.

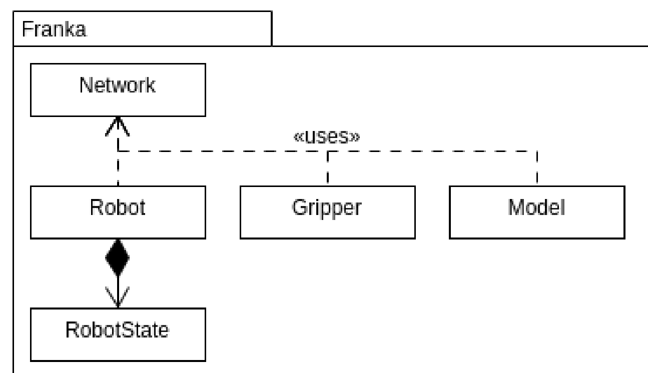


Figure 4.26: Minimalist class diagram of `libfranka`.

First tests

Before building, the robot fondue application, we need to verify if the algorithms presented in section 3 can work in practice as they work in the simulator. For this, we will explore:

1. A LQT/MPC solution to track position and orientation with control commands in torque.

2. A LQT/MPC solution to track orientation and a dynamic programming LQT to track the position with control commands in torque.

3. A LQT/MPC solution to track orientation and a dynamic programming LQT to track the position with control commands in joint velocities.

The planning algorithms were also implemented in C++, but they will not be described here, as we already validated their smooth execution with the simulator experiments, their implementations are basically a translation from Python to C++, thus the approach remains the same.

LQT/MPC solution for position and orientation with commands in torque

For the tracking of the position, the LQT/MPC solution was the one which performs best in the simulator, but as said before, the LQT/MPC solution performs best if and only if it can perform with a sufficient big horizon, it is not easy to define what an big horizon is, since it depends on the task, but usually, we would prefer to use a MPC solution with an horizon bigger than twenty time steps. The problem with that is that the bigger the horizon is, the longer the algorithm will take to compute the control command at each time step. As stated before, the time on a Panda robot to compute one control command should not exceed 0.7 milliseconds otherwise, the connection with the robot will be broken. The main question here is to know what is the maximal possible horizon in practice and does it give an acceptable tracking accuracy.
After some tests, it comes out that the LQT/MPC solution for position tracking is hardly realisable in practice. The tracking only performed successfully with a relatively short horizon of five time steps, and the tracking accuracy was horrible (bigger than two centimeters). There are no results to present since the algorithm never gave results that could be interesting. This problem could be due to several elements:

- As said before, the time complexity plays an important role in the control of a robot. To respect the real-time constraints of the robot, the horizon was decreased to a point where there are no advantages to use such technique (i.e., with an horizon of two time steps, the controller would be exactly the same as solving at each time step a finite horizon LQR between the current and next states).

- Even with a small horizon, the time complexity of the problem remains higher than solving a normal LQT in the task space and transforming the control commands into joint velocities or torque, and we may face-off moments where a control command is missing for a millisecond (it was not said before, but the robot does not directly stop if the one-millisecond constraint is violated, it will stop if the constraint is too often violated). These losses of control commands will generate discontinuities in the robot dynamics, and therefore reduce the quality of our robot model.

Despite the position tracking solution does not work, the LQT/MPC orientation tracking solution works a lot better. As stated before, the horizon does not really affect the performances of the algorithm, mainly because in the position tracking solution, the purpose of the LQT/MPC is to linearize the system around the current point and to give a notion of anticipation to the controller, whereas in the orientation tracking or planning solution the purpose of the LQT/MPC is more to present information from the point of view of the current point. Of course, it is a kind of linearization, but thanks to the fact that orientation changes are smooth, and that the system matrices remain constant, it is not a problem to have a small horizon for this part.

This test showed that another technique is needed to track the position of the robot, but fortunately, the LQT/MPC solution works well for the orientation tracking, which is a good point since it is the only reliable tracking method for position presented in this project (the single LQT solution presented in chapter 2.3, give a sub-optimal solution, and is hardly applicable for a tracking problem). More details about the orientation tracking will be given below.

LQT/MPC solution for orientation, and dynamic programming LQT solution for position with commands in torque or joint velocities

Since we already know that the LQT/MPC solution for orientation tracking works, here, we will mainly focus on the position tracking part, the differences between torque and joint velocities commands, and the error plots of these different techniques.

To compare these two ways to control a robot, the same task is used for both scenarios (the robot needs to grasp an object that lies on the floor). At each time step, the differences between the desired state and current state is computed. For the position, the norm of the Euclidean distance is used, and for the orientation, the distance function between two quaternions is used. Even if this measure can inform us if the regulator accurately tracks the desired trajectory or not, this measure can be perturbed by the joint configuration of the robot, thus we have to use it carefully. To validate one method or the other, other aspects of the motion need to be taken into account (i.e., the naturalness of the motion, the joint configuration resulting from the control commands,...). Unfortunately, there are no mathematical measures to represent these aspects.

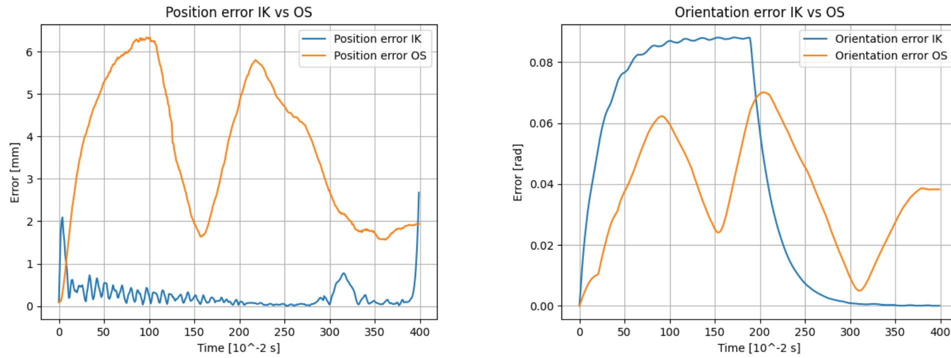


Figure 4.27: Tracking error on a real application.

As pointed out by figure 4.27, the two methods are able to track a trajectory. The position error plot shows us that at some moments, the position tracking is not really accurate with the operational space dynamics solution. It is difficult to say if it is due to the joint configuration of the robot at these moments or not, but in any case, a six millimeters deviation from the target does not guarantee the success of a task. But, as you can see on the OS position tracking error plot, at the end of the task, the regulator successfully compensates for the tracking error (the error is only of 2 millimeters). It is because the speed of the end-effector decreases slowly with a double integrator planning, thus the regulator can more easily compensate for the tracking error than with a single integrator solution in which the position changes more abruptly. It is important to notice that for this example, position and orientation tracking are performed with the same priority. The position tracking error of the inverse kinematics solution is particularly interesting since during the majority of the movement, it is close to the minimal accuracy promoted by the manufacturer of the robot (1 millimeter). In both cases, the orientation tracking works well, the worst case appends with the inverse kinematics solution, which has temporary an absolute deviation of 0.08 radians (= 4.6 degrees), which is acceptable. Leaving aside these error plots, and taking into account the naturalness of these two motions, the torque controlled version of the task looks more natural than the velocity version, which produces a more mechanical motion. This naturalness difference between the two versions is not only due to the tracking but the planning also influences the naturalness of the motion. Since the planning was made with respect to the tracking (i.e., single integrator for the IK solution and double integrator for the OS solution), the planned motion for the OS motion will look more natural since it will be smoother. To overcome the lack of naturalness for the IK solution, we can imagine to plan the motion with a double integrator system and track only the resulting positions and orientations (since we are sending velocity commands, we can not track the velocity of the system). Anyways, this experiment showed up that the two methods can be used to control a robot, thus the method to use will mainly depend on the task of the robot. We would prefer an IK solution for an industrial robot that does not need to look natural, whereas the OS solution would be better if the robot has to perform some human-robot interactions.

It was not presented before, but if during its motion a robot puts itself in an unfavorable joint configuration that does not allow it to accurately perform its task, we can imagine to add a less prioritized task that will maintain as much as possible the robot in a good joint configuration:

$$\dot{\mathbf{q}}_{\text{cmd}} = \dot{\mathbf{q}}_{\text{task}} + \mathbf{N}\frac{\mathbf{q}_{\text{stable}} - \mathbf{q}_t}{\mathrm{dt}},$$

where $\dot{\mathbf{q}}_{\text{task}}$ is the control command resulting from the regulator, $\mathbf{q}_t$ the current joint configuration of the robot, $\mathbf{q}_{\text{stable}}$ a joint configuration that is assumed to be stable for the robot, and $\mathbf{N}$ the nullspace of the full Jacobian.

Architecture of the project

Now that we know which regulators work in practice, we can encapsulate them in a more convenient architecture for the development of the FondueBot project. This architecture aims to facilitate the use of the planning and tracking techniques presented in the project by providing to the programmers interfaces that would directly produce the expected result without worrying about LQT, LQT/MPC, oversampling, ...

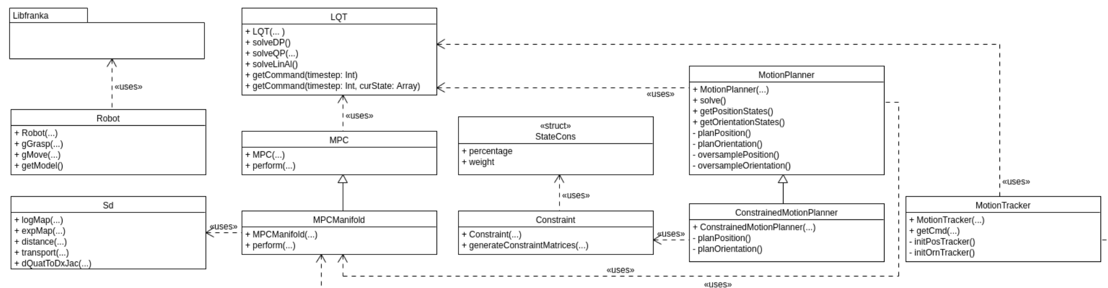


Figure 4.28: Developed architecture for the FondueBot project. It is not specified in the diagram, but all classes use Eigen (a linear algebra library) as mathematical library.

The main advantage of the architecture presented in figure 4.28 is that it simplifies the planning and tracking problem by providing a `ConstrainedMotionPlanner` class that plan a motion which satisfies the task and the constraints, and a `MotionTracker` class that tracks the trajectory created by the planning class. With this architecture, we can easily make a robot achieve a task by specifying:

- The initial and final state for position and orientation.
- A set of constraints.
- The time step and total duration of the motion.

Furthermore, the `ConstrainedMotionPlanner` and `MotionTracker` classes work either with single or double integrator systems. It means that only minimal intervention is needed to control a robot with torque instead of joint velocities.

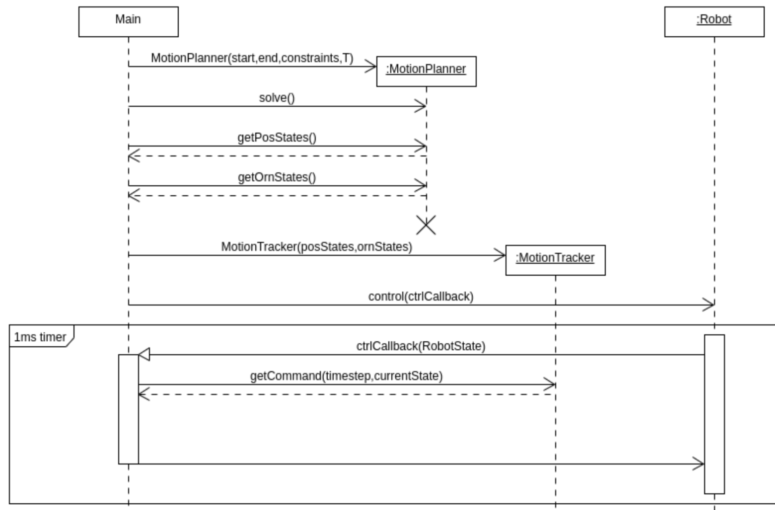


Figure 4.29: Sequence diagram representing the execution of a task

Making the definition of constraints dynamic was a bit challenging. For this, a class `Constraint` has been created, and the constraints of a task are defined as a list of `Constraint` objects. To create the constraint matrices, a method `generateConstraintMatrices()`, which generates the constraint matrices for one constraint has been implemented. The global constraint matrices of the task are build by stacking the $\mathbf{A}_\mathcal{S}$ matrix and $\mathbf{b}_\mathcal{S}$ vector of each constraint together.

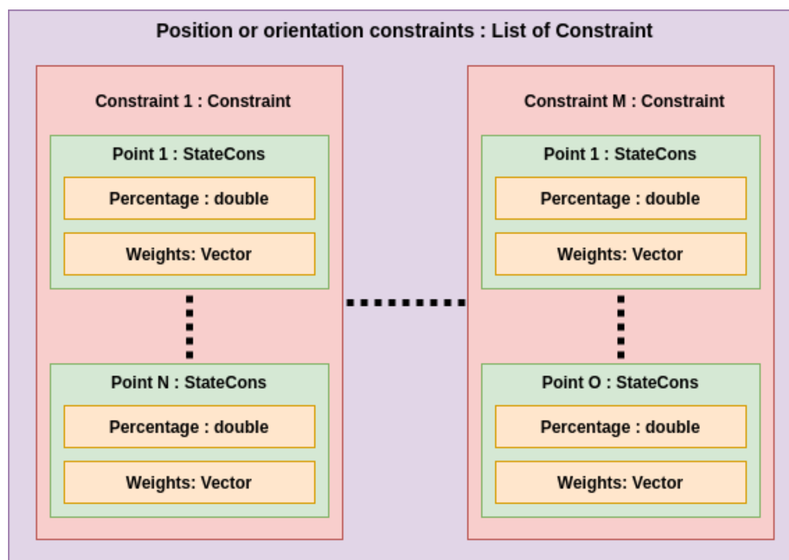


Figure 4.30: Structure of the constraints of a task.

Figure 4.30 shows how constraints are constructed. A constraint is defined by a list of points that are constrained together, and a constraint vector $\mathbf{b}_\mathcal{S}$ (not present in the figure) such that:

$$\sum_{i=1}^{i=N} \mathbf{W}_i \mathbf{p}_i = \mathbf{b}_\mathcal{S},$$

where $\mathbf{W}_i$ is the weight diagonal matrix that contains in its diagonal the weight vector corresponding to its current point, each point $\mathbf{p}_i$ is the state that occurs at the desired percentage of the motion. This structure allows to build constraints that are given to the motion planner.
If we do not want to constrain our motion, a `MotionPlanner` class exists. It works similarly as the constrained version except that it does not need constraints to perform the planning.

The code corresponding to this architecture is available at `https://gitlab.idiap.ch/jmaceiras/wyssmuller`

4.2.5 Ongoing developments

In this chapter, the planning and tracking methods were implemented and tested on a simulator and real robots. From these methods, a programmer-friendly architecture has been created. Now the remaining work is to implement the FondueBot project, with respect to the specifications of the client, from these specifications a finite state machine (where each state corresponds to a specific task for the robots) was thought.

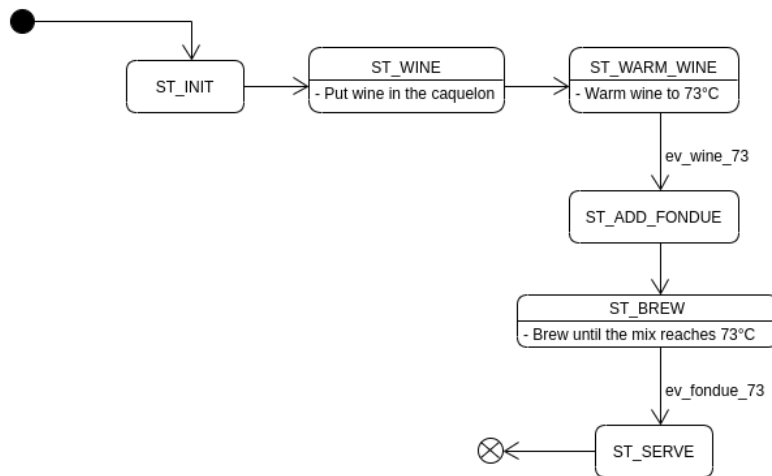


Figure 4.31: State machine of the FondueBot project.

More than the remaining software development, some pieces still need to be modelized and printed, especially the support for the temperature sensor. But the project will be able to progress serenely now that we know that the methods work in practice.

Chapter 5

Conclusion

5.1 Discussion

This project showed up that with only a few information about a task, it is possible to build a whole trajectory that can be accurately tracked by a robot. The developed architecture in this project allows an easy use of these methods, and make the re-programmability of robots accessible for an industrial use. If we compare the proposed method and the pre-recorded motions method, the method presented in this thesis allows a smarter use of robots by making them understand their goal and how to achieve it.

At present, priors information about a task (target, constraints) are given by a human expert. It involves that the human expert needs to know perfectly the workspace of the robot, otherwise he may create tasks that would lead to break the robot. Knowing these priors information is not always trivial. For example, it would be impossible to program a Martian rover with this approach since, at the moment, no human experts are present on the red planet to identify the workspace of the rover. To overcome this issue, we can imagine to add an augmented reality application that will compensate for the lack of knowledge of the human expert by adding a new communication channel between the robot and the human expert. The simplest form of augmented reality system that we can imagine is a smartphone application that shows what the robot sees, and by clicking on a point of interest the robot moves to this point.

The compatibility of the proposed approach with velocity commands is particularly interesting, since torque-level robots are costly, in the industry, we prefer to use cheaper robots that do not offer torque encoders. Torque level robots were for a long time reserved for the world of research or particular tasks. Still, now we can see a new trend that wants to make these robots more affordable, this is notably the case with the Panda robot, which aims to be an alternative to expensive torque-level robots in the frame of the industry 4.0.

5.2 Further works

The main disadvantage of the proposed approach is that planning is done without information about the model of the robot. Thus we may face off a situation where the robot can not track the planned motion. Thanks to the fact that in our experiments we used a redundant robot, and that the prior information of the tasks were set with respect to the robot kinematics and dynamics, this situation was not observed in the frame of this project. But it is a problem that we may face off while applying this approach to other robots. A possible way to solve this problem would be to merge the planning and tracking in one method who would take into account the robot configuration. Similarly to what has been presented with the LQT/MPC approach for the position tracking, we would like to have an algorithm that directly outputs a sequence of torque commands that respects the desired final state and the constraints expressed in the task space of the robot. Unfortunately, this LQT/MPC solution proves difficult to achieve in practice, particularly because of the time complexity of the problem. Furthermore, the LQT/MPC solution assumes that we do not know how the dynamic of our system evolves over time, thus we need to linearize it at every time step. But it is not really the case since, with a good model of the robot, we are able to know the kinematic and dynamic parameters of the robot in function of a predicted state.
Recent searches [18] [19] [20] propose an extension of LQR control that is able to plan out and optimize a sequence, mindful of the changing dynamics of the system. They called this algorithm iterative LQR (iLQR). The algorithm is based on the fact that given an estimated control sequence $\hat{\mathbf{u}}$, we can accurately model the resulting sequence of state $\hat{\mathbf{x}}$. From this sequence, the residual is used ($\mathbf{\Delta x} = \mathbf{x} - \hat{\mathbf{x}}$) to solve another LQR problem to get the residual of the control command $\mathbf{\Delta u}$, which is added to the last estimate to form the new estimate of the control command sequence ($\hat{\mathbf{u}}_{t+1} = \hat{\mathbf{u}}_t + \mathbf{\Delta u}$) The process can be repeated until convergence.
Since we need to know what effect a sequence of control command would have on the robot, iLQR needs an accurate model of the robot but has the advantage to take time to be solved only before the motion, once the algorithm converged, we just apply the resulting sequence of control commands, then there are no risks of breaking the real-time constraint of the robot.
This technique could be used to merge the planning and tracking part, especially [18] proposes a constrained approach for this problem that could fit our scenario. It is something that will be investigated in parallel with the FondueBot project that still needs to be finished.

5.3 Personal conclusion

Here ends this master thesis, that started with some low-level examples on a 2D planar robot, and ended up with a concrete application of the developed techniques. From a personal point of view, this project was my first step in the wide world of robotics. Thanks to it, I discovered something that was totally unknown to me, and

that immediately interested me. I particularly liked the different challenges that occur when using artificial intelligence techniques for a robotics application, since they are used with few data, and permit fewer errors than in other applications their use is really challenging. Furthermore, the fact that with one simple brick (the LQT), and a good use of it, we were able to build a really complex application is really impressive. Jobs still need to be done in the FondueBot project, but thanks to this master thesis, I can see this project with a good point of view, and I can move forward in the development now that I know that the developed techniques are applicable in practice.

Vex, the 15th June 2020 Jérémy Maceiras

Appendix A

Quaternion algebra

A.1 Addition

$$\epsilon_1 + \epsilon_2 = \begin{bmatrix} \epsilon_{10} + \epsilon_{20} \\ \epsilon_{11} + \epsilon_{21} \\ \epsilon_{12} + \epsilon_{22} \\ \epsilon_{13} + \epsilon_{23} \end{bmatrix}$$

A.2 Identity quaternion

$$\epsilon_I = 1 + 0i + 0j + 0k = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = 1$$

A.3 Conjugate

$$\widetilde{\epsilon} = \epsilon_0 - \epsilon_1 i - \epsilon_2 j - \epsilon_3 k = \begin{bmatrix} \epsilon_0 \\ -\epsilon_1 \\ -\epsilon_2 \\ -\epsilon_3 \end{bmatrix}$$

A.4 Multiplication

$$\epsilon_1 \epsilon_2 = \mathbf{E}(\boldsymbol{\epsilon}_1)\boldsymbol{\epsilon}_2$$

$$\mathbf{E}(\boldsymbol{\epsilon}) = \begin{bmatrix} \epsilon_0 & -\epsilon_1 & -\epsilon_2 & -\epsilon_3 \\ \epsilon_1 & \epsilon_0 & -\epsilon_3 & \epsilon_2 \\ \epsilon_2 & \epsilon_3 & \epsilon_0 & -\epsilon_1 \\ \epsilon_3 & -\epsilon_2 & \epsilon_1 & \epsilon_0 \end{bmatrix}$$

A.5 Inverse

Given a quaternion ϵ, we are looking for a quaternion ϵ^{-1}, such that:

$$\epsilon\epsilon^{-1} = \epsilon_I = 1.$$

Solving the equation above results in:

$$\epsilon^{-1} = \frac{\epsilon_0 - \epsilon_1 i - \epsilon_2 j - \epsilon_3 k}{\|\boldsymbol{\epsilon}\|^2}.$$

An interesting point with this formula, is if ϵ denotes a unit quaternion, its inverse is equal to its conjugate.

A.6 Rotation matrix to quaternion transformation

Given a rotation matrix $\mathbf{R}$, the corresponding quaternion can be computed with:

$$\boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_0 \\ \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{1+r_{11}+r_{22}+r_{33}}}{2} \\ \frac{r_{32}-r_{23}}{4\epsilon_0} \\ \frac{r_{13}-r_{31}}{4\epsilon_0} \\ \frac{r_{21}-r_{12}}{4\epsilon_0} \end{bmatrix},$$

where r_{ij} stands for the element in the i-th row and j-th column of $\mathbf{R}$. This transformation is accurate only under certain conditions (if $\epsilon_0 = 0$, a division by zero appends). A transformation accurate under all circumstances is possible by using the trace of the rotation matrix[1].

[1] http://www.euclideanspace.com/maths/geometry/rotations/conversions/matrixToQuaternion/index.htm

Appendix B

Proof of QP formulation for LQT

For the cost function, we want to express:

$$J = (\boldsymbol{\mu} - \mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1 - \mathbf{S}^{\mathbf{u}}\mathbf{u})^\top \mathbf{Q}(\boldsymbol{\mu} - \mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1 - \mathbf{S}^{\mathbf{u}}\mathbf{u}) + \mathbf{u}^\top \mathbf{R}\mathbf{u},$$

under the form:

$$J = \mathbf{u}^\top \mathbf{P}\mathbf{u} + \mathbf{q}^\top \mathbf{u}.$$

We start by developing the cost function:

$$J = (\boldsymbol{\mu}^\top \mathbf{Q} - \boldsymbol{\mathcal{S}}_1^\top \mathbf{S}^{\mathbf{x}\top}\mathbf{Q} - \mathbf{u}^\top \mathbf{S}^{\mathbf{u}\top}\mathbf{Q})(\boldsymbol{\mu} - \mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1 - \mathbf{S}^{\mathbf{u}}\mathbf{u}) + \mathbf{u}^\top \mathbf{R}\mathbf{u}.$$

From this point, we discard all terms not containing $\mathbf{u}$:

$$J = -\underbrace{\boldsymbol{\mu}^T \mathbf{Q}\mathbf{S}^{\mathbf{u}}\mathbf{u}}_{\mathbf{A}} + \underbrace{\boldsymbol{\mathcal{S}}_1^\top \mathbf{S}^{\mathbf{x}\top}\mathbf{Q}\mathbf{S}^{\mathbf{u}}\mathbf{u}}_{\mathbf{B}} - \underbrace{\mathbf{u}^\top \mathbf{S}^{\mathbf{u}\top}\mathbf{Q}\boldsymbol{\mu}}_{\mathbf{C}} + \underbrace{\mathbf{u}^\top \mathbf{S}^{\mathbf{u}\top}\mathbf{Q}\mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1}_{\mathbf{D}} + \mathbf{u}^\top \mathbf{S}^{\mathbf{u}\top}\mathbf{Q}\mathbf{S}^{\mathbf{u}}\mathbf{u} + \mathbf{u}^\top \mathbf{R}\mathbf{u}.$$

If $\mathbf{Q}$ is a diagonal matrix (usually the case), then $\mathbf{A} = \mathbf{C}$ and $\mathbf{B} = \mathbf{D}$:

$$J = 2(\boldsymbol{\mathcal{S}}_1^\top \mathbf{S}^{\mathbf{x}\top}\mathbf{Q}\mathbf{S}^{\mathbf{u}} - \boldsymbol{\mu}^T \mathbf{Q}\mathbf{S}^{\mathbf{u}})\mathbf{u} + \mathbf{u}^\top(\mathbf{S}^{\mathbf{u}\top}\mathbf{Q}\mathbf{S}^{\mathbf{u}} + \mathbf{R})\mathbf{u},$$

with:

$$\begin{aligned}\mathbf{q}^\top &= 2(\boldsymbol{\mathcal{S}}_1^\top \mathbf{S}^{\mathbf{x}\top}\mathbf{Q}\mathbf{S}^{\mathbf{u}} - \boldsymbol{\mu}^T \mathbf{Q}\mathbf{S}^{\mathbf{u}}),\\ \mathbf{P} &= \mathbf{S}^{\mathbf{u}\top}\mathbf{Q}\mathbf{S}^{\mathbf{u}} + \mathbf{R}.\end{aligned}$$

The constraint in the state space is:

$$\mathbf{A}_{\mathcal{S}}\mathbf{x} = \mathbf{b}_{\mathcal{S}},$$

subtracting $\mathbf{x}$ by $\mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1 + \mathbf{S}^{\mathbf{u}}\mathbf{u}$ gives:

$$\mathbf{A}_{\mathcal{S}}(\mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1 + \mathbf{S}^{\mathbf{u}}\mathbf{u}) = \mathbf{b}_{\mathcal{S}},$$

$$\underbrace{\mathbf{A}_{\mathcal{S}}\mathbf{S}^{\mathbf{u}}}_{\mathbf{A}}\mathbf{u} = \underbrace{\mathbf{b}_{\mathcal{S}} - \mathbf{A}_{\mathcal{S}}\mathbf{S}^{\mathbf{x}}\boldsymbol{\mathcal{S}}_1}_{\mathbf{b}},$$

the command space formulation of a state space constraint.

Bibliography

[1] How the tesla model s is made | tesla motors part 1 (wired). https://www.youtube.com/watch?v=8_lfxPI5ObM. Accessed: 2020-06-01.

[2] Electrolux professional laundry — factory tour (ljungby, sweden). https://www.youtube.com/watch?v=JCmGNSoYL-Y. Accessed: 2020-06-01.

[3] Desktop computer host automatic assembly line. https://www.youtube.com/watch?v=GNqNVgLk1Mg. Accessed: 2020-06-01.

[4] Franck C. Park Kevin M. Lynch. *Modern Robotics*. Cambridge University Press, 2017.

[5] Ricardo Campa and Hussein De La Torre. Pose control of robot manipulators using different orientation representations: A comparative review. In *2009 American Control Conference*, pages 2855–2860. IEEE, 2009.

[6] J Luh, M Walker, and R Paul. Resolved-acceleration control of mechanical manipulators. *IEEE Transactions on Automatic Control*, 25(3):468–474, 1980.

[7] Ricardo Campa, Karla Camarillo, and M Ceccarelli. Unit quaternions: A mathematical tool for modeling, path planning and control of robot manipulators. *Robot manipulators, M. Ceccarelli (ed.), In-Teh*, pages 21–48, 2008.

[8] Samuel R Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *IEEE Journal of Robotics and Automation*, 17(1-19):16, 2004.

[9] Herman Bruyninckx and Joris De Schutter. Symbolic differentiation of the velocity mapping for a serial kinematic chain. *Mechanism and machine theory*, 31(2):135–148, 1996.

[10] S. Calinon and D. Lee. Learning control. In P. Vadakkepat and A. Goswami, editors, *Humanoid Robotics: a Reference*, pages 1261–1312. Springer, 2019.

[11] Eduardo D Sontag. *Mathematical control theory: deterministic finite dimensional systems*, volume 6. Springer Science & Business Media, 2013.

[12] S. Calinon. Gaussians on Riemannian manifolds: Applications for robot learning and adaptive control. *IEEE Robotics and Automation Magazine (RAM)*, 2020.

[13] M. J. A. Zeestraten, I. Havoutis, J. Silvério, S. Calinon, and D. G. Caldwell. An approach for imitation learning on Riemannian manifolds. *IEEE Robotics and Automation Letters (RA-L)*, 2(3):1240–1247, June 2017.

[14] Joao Silvério, Leonel Rozo, Sylvain Calinon, and Darwin G Caldwell. Learning bimanual end-effector poses from demonstrations using task-parameterized dynamical systems. In *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 464–470. IEEE, 2015.

[15] ETH Zurich Robotic Systems Lab. *Robot Dynamics: Lecture notes.* ETHZ Press, 2017.

[16] Claudio Gaz, Marco Cognetti, Alexander Oliva, Paolo Robuffo Giordano, and Alessandro De Luca. Dynamic identification of the franka emika panda robot with retrieval of feasible parameters using penalty-based optimization. *IEEE Robotics and Automation Letters*, 4(4):4147–4154, 2019.

[17] Pybullet quickstart guide. `https://docs.google.com/document/d/10sXEhzFRSnvFcl3XxNGhnD4N2SedqwdAvK3dsihxVUA/edit#heading=h.2ye70wns7io3`. Accessed: 2020-06-01.

[18] Marc Toussaint. A tutorial on newton methods for constrained trajectory optimization and relations to slam, gaussian process smoothing, optimal control, and probabilistic inference. In *Geometric and numerical foundations of movements*, pages 361–392. Springer, 2017.

[19] Marc Toussaint. Robot trajectory optimization using approximate inference. In *Proceedings of the 26th annual international conference on machine learning*, pages 1049–1056, 2009.

[20] Emanuel Todorov and Weiwei Li. A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 300–306. IEEE, 2005.