POLITECNICO DI TORINO

Master degree course in MECHATRONICS ENGINEERING

Master Degree Thesis

Analysis, design and implementation of a time optimal planner for robotic applications

Supervisor
Prof. Alessandro RIZZO

Candidate
Gaia ZINNI
s249871

ALTEC Supervisor
Ing. Federico SALVIOLI

October 2019

Abstract

This dissertation focuses on time-optimal path-constrained trajectory planning. Given a geometric path, the aim is to find the optimal time law $\lambda(t)$ which minimizes the time, without violating the robot kinematic and dynamic constraints. The time law is transformed into a joint-space trajectory or open-loop torques, according to the input required by the robot controller. The proposed optimization algorithm is based on the dynamic programming approach, which easily handles several types of cost functions and places few restrictions on the constraints.

The time optimal planner is deeply integrated into the available framework of ALTEC. The existing framework offers the ROS/MoveIt! extension for redundancy resolution, using dynamic programming. Since the underlying resolution technique is the same, the existing architecture is refactored in order to develop a new one to capable to allow users to apply dynamic programming on a generic constrained optimization problems, separating problem specific details from the dynamic programming algorithm. The specific applications are integrated through the creation of specialized components. So, the robot-agnostic extension module responsible for the time optimal trajectory generation is developed, able to parametrize paths in both task and joint space.

Finally, a set of simulations are performed, using the 3D dynamic simulator Gazebo, to analyze and test the planner. The validation of algorithm and other aspects deriving from the design of the planner are demonstrated: in particular, the effects of the discretization method on the result accuracy and the evaluation of the effects of the space resolution on solution's quality and computation time.

Contents

List of Figures 3

1 Introduction 5
- 1.1 Background . . . 5
- 1.2 Focus . . . 6
- 1.3 Objectives . . . 6
- 1.4 Outline . . . 7

2 Time-optimal planning of robotic systems 8
- 2.1 Robotic manipulator modelling . . . 8
  - 2.1.1 Kinematic equations . . . 8
  - 2.1.2 Dynamic equations . . . 11
  - 2.1.3 Inverse Dynamics . . . 12
- 2.2 Motion planning . . . 12
  - 2.2.1 Path and trajectory planner . . . 13
- 2.3 Time parametrization . . . 16
- 2.4 Dynamics parameterization . . . 17
- 2.5 Constraints parametrization . . . 18
- 2.6 Time optimal planning . . . 21
  - 2.6.1 Analysis of phase plane plot . . . 22
  - 2.6.2 Analysis of $\dot{\lambda}^2 - \ddot{\lambda}$ plane . . . 24
  - 2.6.3 Maximum velocity curve . . . 25
  - 2.6.4 Switching Points method . . . 27

3 Resolution with dynamic programming 31
- 3.1 Dynamic programming method . . . 31
- 3.2 Redundancy resolution with dynamic programming . . . 34
  - 3.2.1 Redundancy kinematics . . . 35
  - 3.2.2 Problem formulation . . . 35
- 3.3 Design of a time-optimal planner using dynamic programming . . . 38

3.3.1 Grid representation . . . 38
3.3.2 Discrete approximation of local cost function . . . 40
3.3.3 Discrete approximation of *pseudo-accelerations* . . . 41
3.3.4 Time discretization of joint variables . . . 42
3.3.5 Algorithm design . . . 45

4 Implementation in ROS 47
4.1 ROS and MoveIt! . . . 47
4.1.1 Time optimal trajectory generation in MoveIt! . . . 50
4.2 Gazebo and ROS control . . . 51
4.3 Software architecture . . . 53
4.3.1 Redundancy resolution module . . . 54
4.3.2 Architecture design . . . 55
4.3.3 Context . . . 57
4.3.4 Grid generalization and dynamic programming solver . . . 57
4.3.5 Implementation of time optimal trajectory generation module 59
4.3.6 Trajectory control . . . 61

5 Simulation and results 65
5.1 Application to a 2R planar robot . . . 65
5.1.1 First simulation: comparison with the switching point method 66
5.1.2 Second simulation: comparison between different discretization formulations . . . 68
5.1.3 Third simulation: increase of grid resolution . . . 70
5.1.4 Fourth simulation: JointTrajectoryController . . . 74

6 Conclusions 79
6.1 Results . . . 79
6.2 Future works . . . 80

Bibliography 81

List of Figures

2.1 Kinematic chain . . . 9
2.2 Joint and task space in a three-links manipulator [49] . . . 10
2.3 General control scheme for trajectory planning . . . 13
2.4 Path and trajectory representation . . . 13
2.5 Trapezoidal profile example [49] . . . 15
2.6 S-curve profile [38] . . . 20
2.7 General phase plane plot and maximum velocity curve . . . 23
2.8 Representation of *pseudo-acceleration* limits as arrows in phase plane plot . . . 23
2.9 Three-dimensional phase space with jerk limits [44] . . . 24
2.10 Polygon of feasible *pseudo-acceleration* in the $\dot{\lambda}^2 - \ddot{\lambda}$ plane . . . 25
2.11 Polygon of feasible *pseudo-acceleration* in the $\dot{\lambda}^2 - \ddot{\lambda}$ plane with a critical point . . . 26
2.12 Representation of critical points . . . 28
2.13 Optimal trajectory with single switching point . . . 29
2.14 Resolution with multiple switching points . . . 30
2.15 Fields of *pseudo-accelerations* . . . 30

3.1 Example of optimal path . . . 32
3.2 Stage transition process [21] . . . 33
3.3 Example of a grid explored by dynamic programming . . . 34
3.4 Example of 7-DOF manipulator [22]: Panda by Franka-Emika [1] 35

4.1 Message communication scheme [46] . . . 48
4.2 Circular blends at waypoint $\mathbf{q_i}$. [39] . . . 51
4.3 Scheme representing the architecture overview on the simulated robot in Gazebo, the real robot and the controllers in ROS [2] . . . 54
4.4 UML context diagram . . . 58
4.5 UML decomposition/class diagram . . . 60
4.6 UML sequence diagram . . . 64

5.1 Shiller robot model and desired trajectory 66
5.2 First simulation. The optimal trajectories in phase plane. 67
5.3 First simulation. Comparison between planned torques. 68
5.4 Second simulation. The optimal trajectories in phase plane. 69
5.5 Second simulation: The comparison between planned torques. . . 70
5.6 Second simulation. The comparison between simulated joints' position and ideal ones. 71
5.7 Second simulation: The comparison between simulated joints' velocities and ideal ones. 72
5.8 Third simulation. The comparison between phase plane trajectories. 73
5.9 Third simulation. The comparison between planned torques. . . . 74
5.10 Third simulation. The comparison between simulated joints' positions, the blue curve, and ideal ones, the red curve. 75
5.11 Third simulation. The comparison between simulated joints' velocities, the blue curves, and ideal ones, the red curves. 76
5.12 Fourth simulation. The comparison between planned and interpolated joints' velocities and accelerations. 77
5.13 Fourth simulation. The comparison between simulated joints' positions and planned ones. 77
5.14 Fourth simulation. The comparison between simulated joints' velocities and planned ones. 78
5.15 Fourth simulation. The comparison between planned torques and the applied one. 78

Chapter 1

Introduction

1.1 Background

Robotics is spreading widely in aerospace field: explorative missions, generally, are expensive, riskly and located in hostile environment, resulting dangerous and not convenient to be performed by human beings. For this reasons, robotics assumes a critical role in space exploration: robot are able to replace the human beings or to give a substantial help in the performed activities. Space robots must be capable to survive in harsh environments and to perform the main useful tasks, such as manipulation, exploration on uneven terrains or service.

So, they are adopted mainly for three reasons [53]:

- **Safety**: some tasks could be too dangerous for humans because of the extreme and unknown environments.
- **Performance**: the given tasks could be too difficult for astronauts because of heavy payloads, precision, long duration and others.
- **Cost**: astronauts are very expensive in space because they need an infrastructure to live and they need to come back to Earth, whereas robots need only power and could be left also in space.

Since the actual scenario is moving from the planetary exploration to a colonization, more and more activities are demanded to robots, in order to study and analyze deeply the environment, performing scientific experiments.

The main issues of space planetary robotics are the mobility and the manipulation. In fact, they must be capable to rove without collision and to adapt on different types of terrains. Besides, in order for the task to be performed, robots must possess great capability of manipulation. For example, the drilling or the gripping can be demanded as requirements. Planetary robots are generally

composed by a mobile base or platform, capable to move also autonomously, on which a manipulator arm is mounted, to accomplish the tasks.

Robotics is used not only for planetary exploration, but is exploited to aid in assistance and maintenance of space infrastructures. They are defined as *orbital robots* and are conceived for repairing and maintain hardware, assembling, moving payloads and others. In order to accomplish these tasks, orbital robots should possess great capabilities of manipulation and motion.

1.2 Focus

When dealing with manipulators, the accomplishment of a task requires the motion control of the arm in order to follow the required trajectory, avoiding collision with external obstacles and dangerous situations for the mechanical structure. The motion problem can be divided into two phases: the planning, responsible for deciding the path and the time law of the motion, and trajectory control, responsible for commanding the robot to perform a correct execution of motion.

In some robotic applications, it is required that the execution of the task along a prescribed path is performed as fast as possible in order to maximize the productivity. In a space mission, in fact, it is equivalent to an increase of the mission return and a decrease of the resources utilisation, both on earth and in space. To perform the trajectory in the minimum time, the robot capabilities, in particular the actuators ones, must be exploited at most, reaching the maximum deliverable torque. The path parametrization involves the robot dynamics, since torque constraints must be satisfied in order for a trajectory to be feasible.

The Aerospace Logistics Technology Engineering Company (ALTEC) is the Italian center of excellence for the provision of engineering and logistics services [3] for supporting operations on the International Space Station (ISS) and on the future explorative missions. ALTEC is currently developing technologies for planning and controlling robotic systems, with the aim to improve mission operations.

1.3 Objectives

The purpose of this dissertation is to contribute to the progress of the aforementioned technologies, developing and implementing software and documentations. Three main objectives are pursued:

- Design and implementation of a time-optimal planner.

- Integration with the available framework at ALTEC, based on ROS and MoveIt!.
- Setup of 3D simulation software to support demonstration and testing of the developed technologies.

The technology is not developed as specific for space application, but it could be applied in several environments, such as the industry one. Indeed, the time optimal planning could be very useful to increase the throughput of production, justifying the need of robots to maximize the productivity.

1.4 Outline

The chapters are arranged as follows:

Chapter 2 presents an overview of the kinematic and dynamic modelling and introduces the motion planning problem, with a presentation of some available technologies to perform the minimum time planning.

Chapter 3 analyzes and describes the dynamic programming, which is the proposed method to optimize the trajectory.

Chapter 4 describes the main features of the implementation and integration of the planner on the available framework.

Chapter 5 analyzes the results obtained through the simulations on a 3D software in order to demonstrate the validity of the planner.

Chapter 6 reviews the aims of the study and the obtained results. Finally, it provides some suggestions for improving future works.

Chapter 2

Time-optimal planning of robotic systems along prescribed paths

This chapter describes the mathematical definition of the kinematic and dynamic equation of a manipulator robot, considered as a series of joints and links. Finally, the motion planning problem is described, in particular the time optimal planning and some resolutive techniques.

2.1 Robotic manipulator modelling

2.1.1 Kinematic equations

A robotic manipulator is a kinematic chain composed by a series of *links* interconnected through *joints*. At one end of the chain, the links are connected to a fixed base, while at the other one there is the so-called *end-effector*, the tool which performs the required task.

In order to control the movement of the *end-effector*, it is necessary to compute the relationships between its position and its orientation, called *pose*, and the joints configuration. These relationships are called *kinematic equations*.

It is possible to define two different work-spaces:

- **Joint Space**: It denotes the space defined by the $n \times 1$ vector $\mathbf{q}$ of joint

variables, as follows

$$\mathbf{q} = \begin{bmatrix} q_1 \\ q_2 \\ \dots \\ q_n \end{bmatrix} \tag{2.1}$$

- **Task Space**: It denotes the space defined by the $m \times 1$ vector of the robot *pose* $\mathbf{x}$, composed by the position and orientation of the *end-effector*.

$$\mathbf{x} = \begin{bmatrix} \mathbf{p} \\ \boldsymbol{\theta} \end{bmatrix} \tag{2.2}$$

where $\mathbf{p}$ represents the position of the end-effector, while $\boldsymbol{\theta}$ the orientation.

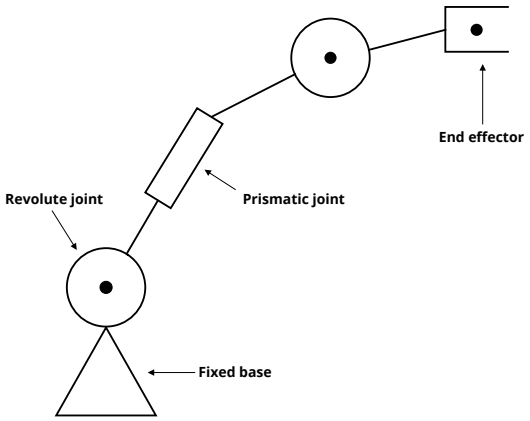

Figure 2.1 – Kinematic chain

Since a joint motion produces an end effector motion, we have to find the equations which relate these space representations.

The *kinematic functions* can be divided into 4 main types:

- **Direct position KF**: from the joint positions to pose.

$$\mathbf{x} = f(\mathbf{q}) \tag{2.3}$$

- **Direct velocity KF**: from the joints velocities to pose velocity.

$$\dot{\mathbf{x}}(\mathbf{t}) = \mathbf{J}(\mathbf{q}(t))\dot{\mathbf{q}}(t) \tag{2.4}$$

- **Inverse position KF**: from the pose to joints configuration.

$$\mathbf{q} = f^{-1}(\mathbf{x}) \tag{2.5}$$

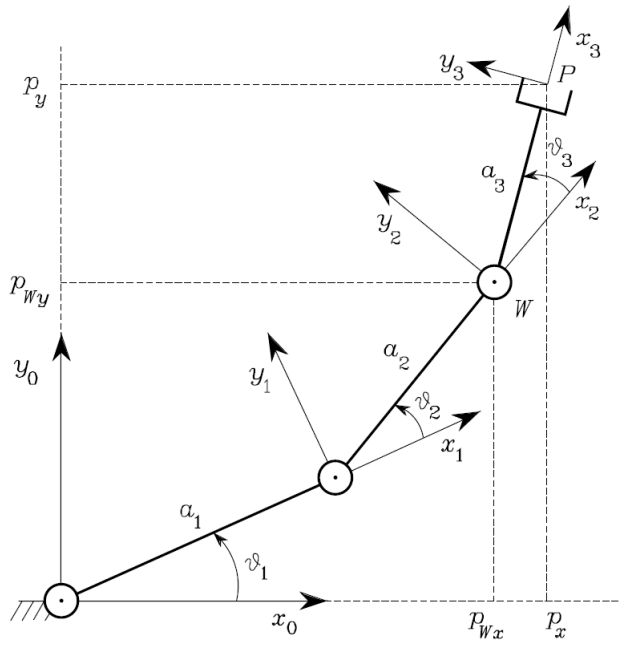

Figure 2.2 – Joint and task space in a three-links manipulator [49]

- **Inverse velocity KF**: from the pose velocity to joints velocities.

$$\dot{\mathbf{q}}(\mathbf{t}) = \mathbf{J}^{-1}(\mathbf{x}(t))\dot{\mathbf{x}}(t) \tag{2.6}$$

where $\mathbf{J}$ is the Jacobian matrix of the function $f(\mathbf{q})$ with respect to $\mathbf{q}$.

While the direct kinematic equations are uniquely and always determined, the inverse kinematic ones are more complex because the equations usually are non linear and may not possible to write them in a *closed-form*. Besides, if the robot is redundant, that is it has more degrees of freedom than those required from the task, the solutions from the inverse kinematics are infinite. Finally, it could also happen that a solution does not exist if the desired pose can't be reached by the robot. In order to handle these equations, numerical solution techniques are often used, especially in control software. Some examples of inverse kinematic solver are *KDL* [4] or *IKFast* [5].

Through the kinematic equations, it is also possible to find the so-called *singularities*. They are particular poses in the task space in which the mobility of the robot is reduced. When the robot is in a singular configuration, it may happen that infinite solutions of the inverse kinematics exist; besides, in the neighbourhood of the singularity, small desired velocities in task space could cause very high velocities in joint space [49]. If the velocity joints limits or torque limits are not present, the movement could provoke serious damages to

the structure; otherwise, the limits could exclude the movement inside the space near the singular point. These points can be found through the Jacobian function: when $\det J(\mathbf{q_s}) = 0$ the matrix J is not invertible and $\mathbf{q_s}$ is called a *singular configuration*. These configurations should be checked during task-space planning phase in order to avoid dangerous situations or some techniques to make the inversion of the Jacobian should be used, such as the *singularity robust inverse* [40].

2.1.2 Dynamic equations

In order to design a time optimal planner, it is necessary to obtain and to study the equations which represent the manipulator dynamics, describing the relationship between the applied torques to the joints and the motion of the structure. There are two main methods to compute the dynamic equation of a robot: the Lagrange formulation and the Newton-Euler formulation.

Assuming to use the Lagrange formulation, the equations depend only on energy of the system and does not depend on the reference frame considered. The dynamic model of a robotic manipulator in the absence of viscous friction forces and contact forces is given by:

$$\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{q}^T\boldsymbol{C}(\boldsymbol{q})\dot{\boldsymbol{q}} + \boldsymbol{g}(\boldsymbol{q}) = \boldsymbol{\tau} \tag{2.7}$$

where $\boldsymbol{H}(\boldsymbol{q})$ is the $n \times n$ generalized inertia matrix of the robot, $\boldsymbol{C}(\boldsymbol{q})$ is the $n \times 1$ array of Coriolis and centrifugal forces, $\boldsymbol{g}(\boldsymbol{q})$ is the $n \times 1$ gravitational forces torques and $\boldsymbol{\tau}$ is the vector of applied torques applied to the joints. (2.7) yields a set of n second-order differential equations, which corresponds to a state-space system with $2n$ state variables, which are the joints' positions $\boldsymbol{q}$ and joints' velocities $\dot{\boldsymbol{q}}$.

Eq. (2.7) can be rewritten in the following form:

$$\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \boldsymbol{\tau} \tag{2.8}$$

where $\boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ contains the remaining terms of (2.7). The coefficients $\boldsymbol{H}(\boldsymbol{q})$ and $\boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ could be computed using an **Inverse Dynamics Algorithm**, as described in [30] and explained in the following section. The numerical computation of these parameters becomes necessary in complex robotic structures, in which the analytical model may be very arduous to obtain.

2.1.3 Inverse Dynamics

The inverse dynamics is used to compute the forces required to obtain desired joint accelerations. To compute the inverse dynamics, it is always possible to use both the Lagrange formulation or the Newton-Euler one. The latter is a recursive algorithm and is more efficient than the former. Although this algorithm doesn't use the $\boldsymbol{H}$ and $\boldsymbol{h}$ parameters, it can be exploited to compute them.

Taking into account (2.8), it is possible to compute $\boldsymbol{H}(\boldsymbol{q})$ e $\boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}})$, as described in [30]. For sake of simplicity, let us call the inverse dynamic function $IDA(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}})$.

$\mathbf{h}(\mathbf{q}, \dot{\mathbf{q}})$ computation

If the acceleration vector is set to zero, (2.8) becomes:

$$\boldsymbol{\tau} = \boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}}) \tag{2.9}$$

So, given position $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$, $\boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ can be interpreted as the forces vector which generates zero acceleration. So through the inverse dynamics function, imposing the actual $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$ and $\ddot{\boldsymbol{q}} = 0$, $\boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ is equal to the applied force and is computed as follows:

$$\boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = IDA(\boldsymbol{q}, \dot{\boldsymbol{q}}, 0) \tag{2.10}$$

$\mathbf{H}(\mathbf{q})$ computation

From the definition of the inverse dynamics function, the robot equation (2.8) and the equation for $\boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ (2.10), it is possible to compute $\boldsymbol{H}(\boldsymbol{q})$ as follows:

$$IDA(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}) = \boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + IDA(\boldsymbol{q}, \dot{\boldsymbol{q}}, 0) \tag{2.11}$$

So, it becomes:

$$\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} = IDA(\boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}) - IDA(\boldsymbol{q}, \dot{\boldsymbol{q}}, 0) \tag{2.12}$$

This method is not very efficient from a computational point of view. *Walker et al.* proposes a more efficient method, called *Composite Rigid Body Method*, where the links are treated as composite rigid bodies and the inertia of single links is used to compute the entire joint space inertia matrix [54].

2.2 Motion planning

The motion planning resolution is a fundamental problem of robotic manipulator control. In order to move the robot as desired, the system can be decomposed in two levels: the higher level is the path and/or trajectory generator, while the

lowest one is the control tracking problem . The entire system can be divided in the following main blocks, as described in Fig. 2.3.

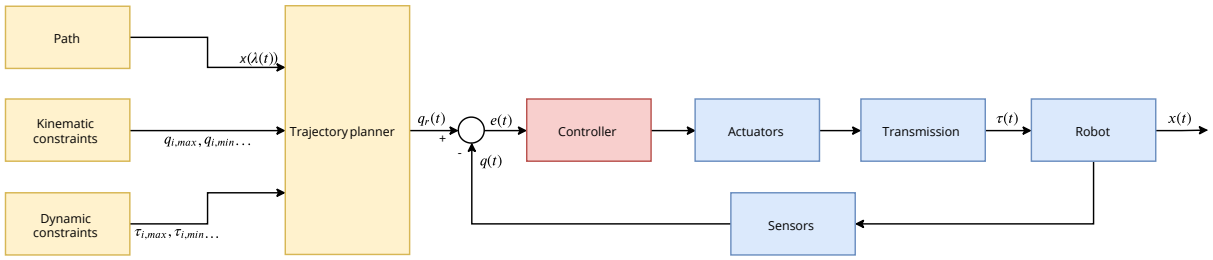

Figure 2.3 – General control scheme for trajectory planning

- **Path planner**: it has the task to generate the sequence of points that the robot has to reach in joint space or in task space, avoiding the obstacles.
- **Trajectory planner**: it has the task to generate the time law of the path, respecting all the kinematics and dynamics constraints.
- **Controller**: given the desired references, the controller generates the torques or the forces to be applied in order to follow the desired trajectory.
- **Robot**: it is the plant of the system.

2.2.1 Path and trajectory planner

The goal of a path and trajectory planner is to find the reference inputs for the control system. Often the two terms are confused, but there is a substantial difference: the path is defined as the geometrical description of the desired waypoints in the task or joint space, while the trajectory is the path with a time law, defining the positions and the velocities and/or the accelerations at each instant time.

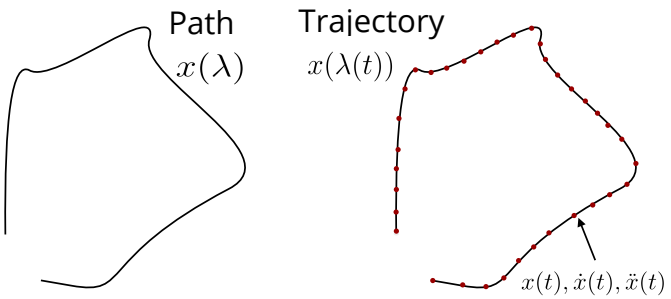

Figure 2.4 – Path and trajectory representation

As [34] explains, the input path can be described in both task and joint space. Usually, the geometric path is described in the task space because the task which has to be performed has meaning in that space. So, planning a path in task space is equivalent to finding the waypoints, or poses, of the end effector at each time instant. Although the task space planning is more convenient and predictable, the joint space planning is often used. The main reason is that the robot is actuated through the motors located at the joints. For this reason, the waypoints are given directly in terms of joint positions or they are obtained from task space through the inverse kinematic equations and joined through an interpolation function. The joint space planning has the advantages that the singularities can be predicted and the reference values can be sent directly to the controller, but the drawback is that the resulting path of the end effector is not predictable, unless generated from task space.

The trajectory planner generates the desired trajectory, taking as input the path and taking into account the robot kinematic and dynamic constraints. The trajectory can be represented by a time-varying scalar quantity λ, called *profile* or *curvilinear coordinate*. This quantity links each waypoint to a time instant and it varies monotonically in time as follows:

$$\lambda(t_0) < \lambda(t) < \lambda(t_f) \tag{2.13}$$

The kinematic and dynamic constraints of the robot can be expressed as constraints of λ and its derivatives:

$$\dot{\lambda}_{min}(t) < \dot{\lambda}(t) < \dot{\lambda}_{max}(t) \tag{2.14}$$

$$-\ddot{\lambda}_{max}(t) < \ddot{\lambda}(t) < \ddot{\lambda}_{max}(t) \tag{2.15}$$

where $\dot{\lambda}(t)$ and $\ddot{\lambda}(t)$ are the derivatives of λ with respect to time.

The desired motion for manipulators can be described in three main ways:

- Motion in free space between two end-points, called *point-to-point motion*.
- Motion between two end-points in a constrained space, containing points to be avoided.
- Full constrained motion along an assigned path.

The first is the most simple and is defined between two points, traveled in a certain time t_f. This planning is chosen when the actual path performed by the robot is not important and only the initial and final points are useful. In order to generate the trajectory, it is possible to adapt different methods: for example it is

possible to choose a performance index to be optimized or a simple *trapezoidal* profile. If a *trapezoidal* profile is chosen, the trajectory is defined as follows: a constant acceleration in the start phase is set until a cruise velocity is reached and finally a constant deceleration is imposed in order to reach the final point. The resulting positions curve is composed by two parabolic segments connected through a linear one. It is called *trapezoidal*, because the velocity profile assumes a trapezoidal curve, as it is possible to see in Fig. 2.5. If the cruise velocity is not reached, the profile becomes *bang-bang*: the acceleration passes immediately from the maximum value to the minimum one and the linear segment disappears.

The second motion type is very similar to the first one, with the only difference that the path must take into account the obstacles to avoid them.

The last motion type expects a sequence of intermediate points, enforcing the entire path performed. It is very useful in some industrial applications, for example in welding. The points can be joined through an interpolation function, whose degree depends on the number of provided parameters: linear if only position is given, cubic if positions and velocities are given, quintic if also accelerations are provided.

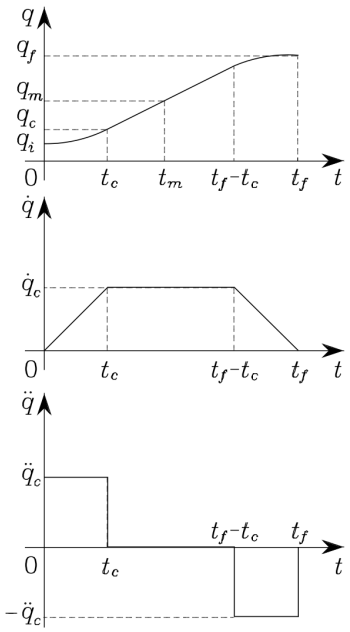

Figure 2.5 – Trapezoidal profile example [49]

In this thesis, the third category of motion is considered.

2.3 Time parametrization

The *path* can be defined as $\boldsymbol{x}(\lambda)$ in the task space or $\boldsymbol{q}(\lambda)$ in the joint space, where $\lambda(t)$ represents the time-law, associating each point of the path to a time. It varies monotonically with respect to time in $[0, \Lambda]$. If Λ is defined as the length of the path, λ is called *abscissa coordinate* and $\dot{\lambda}$ and $\ddot{\lambda}$ represent the path velocity and path accelerations. They are called *pseudo-velocity* and *pseudo-acceleration* respectively [24].

Exploiting the λ-parametrization, the order of original dynamic equation can be reduced from $2n$ to 2 state variables, expressing the joints positions $\mathbf{q}$, joints velocities $\dot{\mathbf{q}}$ and joints accelerations $\ddot{\mathbf{q}}$ in terms of λ and its derivatives [48].

Defined λ, the configuration $\mathbf{q}$ or the pose $\mathbf{x}$ can be computed for each istant time as follows:

$$\mathbf{q}(t) = \boldsymbol{f}_q(\lambda(t)) \tag{2.16}$$

$$\mathbf{x}(t) = \boldsymbol{f}_x(\lambda(t)) \tag{2.17}$$

The aim of the time optimal trajectory planning is to find the time law λ such that the traveled time is minimum and to obtain the trajectory $\boldsymbol{x}(\lambda(t))$ or $\boldsymbol{q}(\lambda(t))$ or the open-loop torques, according to the input required by the robot motion controller.

Let us consider a path in the joint space, which is parameterized with respect to the variable λ. It is possible to compute the time derivative of $\boldsymbol{q}(\lambda(t))$, as a function of the abscissa coordinate and its time derivatives. So the joints' velocities and accelerations are computed as [43]:

$$\dot{\boldsymbol{q}} = \boldsymbol{q}'\dot{\lambda} \tag{2.18}$$

$$\ddot{\boldsymbol{q}} = \boldsymbol{q}'\ddot{\lambda} + \boldsymbol{q}''\dot{\lambda}^2 \tag{2.19}$$

where $\boldsymbol{q}'$ and $\boldsymbol{q}''$ are the derivatives of $\boldsymbol{q}$ with respect to λ and can be defined as the parametric velocities and accelerations.

It is possible to compute the same parameterization also if the path is given in the task space, using the inverse kinematic function. Assuming that the path does not meet singularities and the jacobian matrix $\mathbf{J}$ is known, the parametric joints' velocities and accelerations as function of the task space positions and velocities are computed using the inverse kinematics:

$$\boldsymbol{q}' = \boldsymbol{J}^{-1}\boldsymbol{x}' \tag{2.20}$$

$$\boldsymbol{q}'' = \boldsymbol{J}^{-1}(\boldsymbol{x}'' - J'\boldsymbol{q}') \tag{2.21}$$

where $\boldsymbol{x}'$ and $\boldsymbol{x}''$ are the derivatives of $\boldsymbol{x}$ with respect to λ and can be defined as parametric task space velocities and accelerations, while $\boldsymbol{J}'$ is the λ-derivative of the jacobian matrix. Now it is possible to compute the joint velocities and accelerations [51]:

$$\dot{\boldsymbol{q}} = \boldsymbol{q}'\dot{\lambda} = J^{-1}\boldsymbol{x}'\dot{\lambda} \tag{2.22}$$

$$\ddot{\boldsymbol{q}} = \boldsymbol{q}'\ddot{\lambda} + \boldsymbol{q}''\dot{\lambda}^2 = J^{-1}\boldsymbol{x}'\ddot{\lambda} + J^{-1}(\boldsymbol{x}'' - J'\boldsymbol{q}')\dot{\lambda}^2 \tag{2.23}$$

2.4 Dynamics parameterization

Using the parametrization, it is possible to transform the original robot dynamics equations into a set of equations dependent only on the λ-parameter, reducing the problem complexity [25].

Substituting (2.18) and (2.19) in (2.8), the parameterized set of robot dynamic equations is obtained as follows:

$$\boldsymbol{a}(\lambda)\ddot{\lambda} + \boldsymbol{b}(\lambda)\dot{\lambda}^2 + \boldsymbol{h}(\boldsymbol{q}(\lambda), \dot{\boldsymbol{q}}(\lambda, \dot{\lambda})) = \boldsymbol{\tau} \tag{2.24}$$

where

$$\boldsymbol{a}(\lambda) = \boldsymbol{H}(\boldsymbol{q}(\lambda))\boldsymbol{q}'(\lambda) \tag{2.25}$$

$$\boldsymbol{b}(\lambda) = \boldsymbol{H}(\boldsymbol{q}(\lambda))\boldsymbol{q}''(\lambda) \tag{2.26}$$

are two $n \times 1$ vectors. The nonlinear term $\boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ depends both on joints' positions and velocities, so it is a function of both λ and $\dot{\lambda}$. If a discretized state-space system of λ and $\dot{\lambda}$ is defined, it is possible to compute numerically the value of $\boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ for each couple $(\lambda, \dot{\lambda})$.

The system of equations can be rewritten for each joint in the following way:

$$a_i(\lambda)\ddot{\lambda} + b_i(\lambda)\dot{\lambda}^2 + h_i(\boldsymbol{q}(\lambda), \dot{\boldsymbol{q}}(\lambda, \dot{\lambda})) = \tau_i \tag{2.27}$$

This equation is valid for the i-th torque applied to the i-th joint.

Alternatively, the complete dynamics equation could be parametrized too. Substituting (2.18) and (2.19) in (2.7), the set of equations yields [48]:

$$\boldsymbol{m}(\lambda)\ddot{\lambda} + \boldsymbol{s}(\lambda)\dot{\lambda}^2 + \boldsymbol{g}(\lambda) = \boldsymbol{\tau} \tag{2.28}$$

where

$$\boldsymbol{m}(\lambda) = \boldsymbol{H}(\boldsymbol{q}(\lambda)\boldsymbol{q}'(\lambda) \tag{2.29}$$

$$\boldsymbol{s}(\lambda) = \boldsymbol{H}(\boldsymbol{q}(\lambda)\boldsymbol{q}''(\lambda) + \boldsymbol{q}'^T\boldsymbol{C}(\boldsymbol{q}(\lambda))\boldsymbol{q}'(\lambda) \tag{2.30}$$

For each joint, it becomes:

$$m_i(\lambda)\ddot{\lambda} + s_i(\lambda)\dot{\lambda}^2 + g_i(\lambda) = \tau_i \tag{2.31}$$

Exploiting this parametrization, the original state-space system is reduced to 2 state variables, which are λ and $\dot{\lambda}$ [48].

2.5 Constraints parametrization

The robot constraints can be defined as function of the variable λ, such that they can be easily included in the trajectory planning algorithm.

Torque limits

In real world, the actuators, such as the electric motors, are characterized by maximum torque limits, which we define as $\tau_{i,min}$ and $\tau_{i,max}$. So, for each joint, the following constraint must be satisfied [20]:

$$\tau_{i,min} \leq \tau_i \leq \tau_{i,max} \tag{2.32}$$

The limited torques result in a maximum deceleration and acceleration of the joints, which can be translated into a constraint of the second derivative of λ, as follows [24]:

$$L_i(\lambda, \dot{\lambda}) \leq \ddot{\lambda} \leq U_i(\lambda, \dot{\lambda}) \tag{2.33}$$

where L_i and U_i are computed substituting (2.32) in (2.31) :

$$L_i = \frac{\tau_{i,min}\delta_i(\lambda) + \tau_{i,max}(1 - \delta_i(\lambda)) - s_i(\lambda)\dot{\lambda}^2 - g_i(\lambda)}{m_i(\lambda)} \tag{2.34}$$

$$U_i = \frac{\tau_{i,max}\delta_i(\lambda) + \tau_{i,min}(1 - \delta_i(\lambda)) - s_i(\lambda)\dot{\lambda}^2 - g_i(\lambda)}{m_i(\lambda)} \tag{2.35}$$

where δ_i depends on the sign of m_i. In fact $\delta_i = 1$ if $m_i(\lambda) > 0$ and $\delta_i = 0$ if $m_i(\lambda) < 0$. If $m_i = 0$, it means that the i-th actuator does not contribute to the bounds of the *pseudo-accelerations* and the acceleration bounds are due to the other $n - 1$ actuators [48]. It is worth to underline that this actuator does contribute to the *pseudo-velocity* limits. Indeed, imposing $m_i = 0$ and assuming $g_i \neq 0$, (2.31) becomes:

$$s_i(\lambda)\dot{\lambda}^2 + g_i = \tau_i \tag{2.36}$$

Substituting (2.32) in (2.36), the following constraint of $\dot{\lambda}^2$ is obtained [24]:

$$\tau_{i,min} - g_i(\lambda) \leq s_i(\lambda)\dot{\lambda}^2 \leq \tau_{i,max} - g_i(\lambda) \tag{2.37}$$

$$\frac{\tau_{i,min}\beta_i + \tau_{i,max}(1-\beta_i) - g_i(\lambda)}{s_i(\lambda)} \leq \dot{\lambda}^2 \leq \frac{\tau_{i,max}\beta_i + \tau_{i,min}(1-\beta_i) - g_i(\lambda)}{s_i(\lambda)} \tag{2.38}$$

where β_i depends on the sign of $s_i(\lambda)$. In fact, $\beta_i = 1$ if $s_i(\lambda) > 0$, while $\beta_i = 0$ if $s_i(\lambda) < 0$.

The same observations could be done if a reduced dynamics equation is used to model manipulator. Substituting (2.32) in (2.27), the *pseudo-acceleration* bounds are:

$$L_i = \frac{\tau_{i,min}\delta_i(\lambda) + \tau_{i,max}(1-\delta_i(\lambda)) - b_i(\lambda)\dot{\lambda}^2 - h_i(\lambda,\dot{\lambda})}{a_i(\lambda)} \tag{2.39}$$

$$U_i = \frac{\tau_{i,max}\delta_i(\lambda) + \tau_{i,min}(1-\delta_i(\lambda)) - b_i(\lambda)\dot{\lambda}^2 - h_i(\lambda,\dot{\lambda})}{a_i(\lambda)} \tag{2.40}$$

where δ_i is computed as before, substituting $m_i(\lambda)$ with $a_i(\lambda)$. Since the pseudo-acceleration constraint must be verified for each joint, it can be rewritten in a shorter way as follows:

$$L(\lambda,\dot{\lambda}) = \max_i L_i(\lambda,\dot{\lambda}) \tag{2.41}$$

$$U(\lambda,\dot{\lambda}) = \min_i U_i(\lambda,\dot{\lambda}) \tag{2.42}$$

and the constraint becomes:

$$L(\lambda,\dot{\lambda}) \leq \ddot{\lambda} \leq U(\lambda,\dot{\lambda}) \tag{2.43}$$

Torque rate limits

In real actuators, the torques can not change immediately, because of the inductive behaviour of electric motors which affects the provided current. So the torque rate limits, or called actuator *jerk*, should be considered as constraints. The actuator *jerk* is defined as the derivative of the torque and indicates the rate at which it increases or decreases. A violation of the *jerk* constraint causes vibration in the mechanical structure and a jerky motion [27]. For example, the *trapezoidal* profile at Fig. 2.5 causes an infinite jerk when the acceleration passes from the maximum or minimum value to zero and viceversa. In order to avoid this problem, a *s-profile* for the velocity is chosen such that the acceleration does not change immediately, but in a gradual way.

The torque rate limit can be defined as:

$$\dot{\tau}_{i,min} \leq \dot{\tau}_i \leq \dot{\tau}_{i,max} \tag{2.44}$$

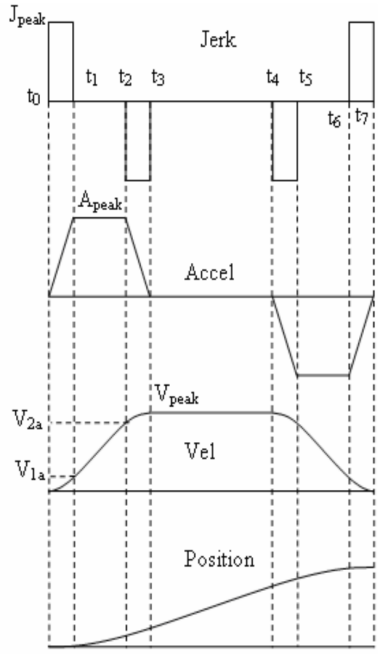

Figure 2.6 – S-curve profile [38]

The torque rate bounds can be transformed into *pseudo-accelerations* and *pseudo-jerk* limits. Since the dynamic equations (2.7) takes into account only the torque variables, the third-order derivative is required [27]:

$$\boldsymbol{H}(\boldsymbol{q})\dddot{\boldsymbol{q}} + \dot{\boldsymbol{H}}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q},\dot{\boldsymbol{q}})\ddot{\boldsymbol{q}} + \dot{\boldsymbol{C}}(\boldsymbol{q},\dot{\boldsymbol{q}})\dot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q},\dot{\boldsymbol{q}})\ddot{\boldsymbol{q}} + \dot{\boldsymbol{g}}(\boldsymbol{q}) = \dot{\boldsymbol{\tau}} \tag{2.45}$$

It is possible to express the third derivative of $\boldsymbol{q}$ as function of λ:

$$\dddot{\boldsymbol{q}} = \boldsymbol{q}'''\dot{\lambda}^3 + 3\boldsymbol{q}''\dot{\lambda}\ddot{\lambda} + \boldsymbol{q}'\dddot{\lambda} \tag{2.46}$$

while in task space it is:

$$\dddot{\boldsymbol{q}} = \boldsymbol{J}^{-1}(\boldsymbol{x}''' - \boldsymbol{J}''\boldsymbol{q}' - 2\boldsymbol{J}'\boldsymbol{q}'') \tag{2.47}$$

where $\boldsymbol{J}''$ is the second derivative of $\boldsymbol{J}$ with respect to λ. By substituting (2.45) into (2.44), it is possible to translate the torque rate limits into limits of the λ derivatives.

As [27] demonstrates, they determine a constraint on *pseudo-jerk*, *pseudo-acceleration* and *pseudo-velocity*:

$$j_{min}(\lambda,\dot{\lambda},\ddot{\lambda}) \leq \dddot{\lambda} \leq j_{max}(\lambda,\dot{\lambda},\ddot{\lambda}) \tag{2.48}$$

$$a_{\tau,min}(\lambda,\dot{\lambda}) \leq \ddot{\lambda} \leq a_{\tau,max}(\lambda,\dot{\lambda}) \tag{2.49}$$

$$\dot{\lambda} \leq v_{\tau,max}(\lambda) \tag{2.50}$$

where j_{min}, j_{max}, $a_{\tau,min}$, $a_{\tau,min}$ and $v_{\tau,max}$ are the maximum and minimum limits from (2.44).

Joint acceleration and velocity limits

Joints have limits on accelerations and velocities too, because of the mechanical structure. They are defined as:

$$\ddot{q}_{i,min} \leq \ddot{q}_i \leq \ddot{q}_{i,max} \tag{2.51}$$

$$-\dot{q}_{i,min} \leq \dot{q}_i \leq \dot{q}_{i,max} \tag{2.52}$$

The accelerations limits are transformed into *pseudo-acceleration* constraints substituting (2.19) into (2.51) [39]:

$$a_{q,min}(\lambda,\dot{\lambda}) \leq \ddot{\lambda} \leq a_{q,max}(\lambda,\dot{\lambda}) \tag{2.53}$$

while the velocities constraints lead to the following limit of the *pseudo-velocity*, substituting (2.18) into (2.52):

$$v_{q,min}(\lambda,\dot{\lambda}) \leq \dot{\lambda} \leq v_{q,max}(\lambda,\dot{\lambda}) \tag{2.54}$$

2.6 Time optimal planning

The aim of the time optimal planning is to generate the reference inputs to perform the path desired in the minimum time. The problem can be formalized in the following way:

$$\begin{aligned} \min_{u(t)} \quad & \int_0^{tf} 1dt \\ \text{s.t.} \quad & L(s(t)) - u(t) \leq 0 \ \forall t, \\ & u(t) - U(s(t)) \leq 0 \ \forall t, \\ & s(0) = s_0, \\ & s(t_f) = s_f \end{aligned} \tag{2.55}$$

where $u(t)$ is the input of the system and it is equal to the *pseudo-acceleration* $\ddot{\lambda}$, while $s(t)$ is the state of the system and it is equal to $s(t) = [\lambda(t), \dot{\lambda}(t)]$.

So the problem can be described as:

"Given the initial state $s(0) = [\lambda(0), \dot{\lambda}(0)]$*, find the optimal control inputs* u *that move the robot along the prescribed path to the final state* $s(t_f) = [\lambda(f), \dot{\lambda}(f)]$ *in the minimum time, without violating the constraints."*

As [24] states, thanks to to the Extended Pontryagin's Maximum Principle, which allows to include state-dependent constraints, it is possible to demonstrate that, given these constraints, the optimal control input $u(t) = \ddot{\lambda}(t)$ always takes its maximum or minimum value, except for particular cases, such as singular points. Along prescribed paths, *only one* actuator will be always saturated and the others will work to keep the robot in the right position, whereas in *point-to-point* motion *at least* one actuator is saturated. So, the optimal trajectory becomes *bang-bang* in torques [25]. However, this strategy is not convenient because the requested torques rates are not physically realizable and it would induce vibrations in the structure [55]. In order to avoid this problem, [27] tries to obtain a smooth trajectory, adding a constraint on the torque rate.

As *Casalino et al.* describes, it is possible to divide the resolution methods for optimal time trajectory planning in three main categories [23]:

- **Direct methods**: They transform the optimization problem into a convex linear programming one.
- **Indirect methods**: They use the phase plane plot to construct the optimal trajectory, which is given by a sequence of maximum accelerations and deceleration.
- **Dynamic programming method**: They are based on Bellman principle and are able to solve a big class of optimization problems. In fact, they can include several performance indices and constraint functions.

The indirect methods will be analyzed: their results are used as a term of comparison for the simulations in Chapter 5. Besides, they are based on the phase plane plot which allows an accurate analysis of the problem.

2.6.1 Analysis of phase plane plot

Any parametrized trajectory can be represented as a sequence of points in the $\lambda - \dot{\lambda}$ plane, generating a curve. Obviously, not all the curves in the phase plane plot are feasible. The constraints on $\ddot{\lambda}$ can be translated into constraints of the *pseudo-velocity*, limiting the feasible area of the phase-plane plot. In fact, a *pseudo-velocity* is feasible if $L(\lambda, \dot{\lambda}) \leq U(\lambda, \dot{\lambda})$. It is possible to represent the constraint on the plane through a curve, called *Maximum Velocity Curve (MVC)*, composed by the limit values of $\dot{\lambda}$ which do not satisfy the disequality. All the feasible trajectories must be below this curve.

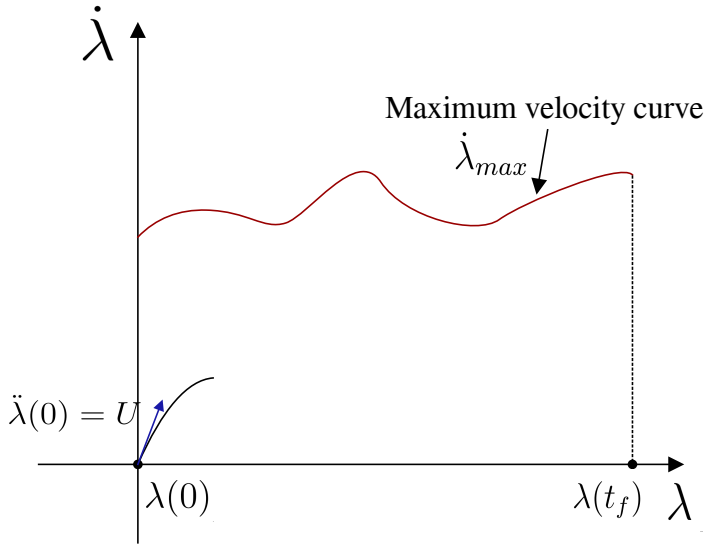

Figure 2.7 – General phase plane plot and maximum velocity curve

For each point of this plane, it is possible to represent two vectors for representing the minimum and maximum applicable acceleration, e.g. $L(\lambda, \dot{\lambda})$ and $U(\lambda, \dot{\lambda})$ respectively. When the scissors close and the two arrows overlap, it means that the point is on the *MVC*: in fact, only one value of the acceleration is possible [36], but some particular points, which will be described later. In order for a path to be feasible, the tangent in every point of the phase plane trajectory, which corresponds to *pseudo-acceleration*, must be included in the range defined by the arrows.

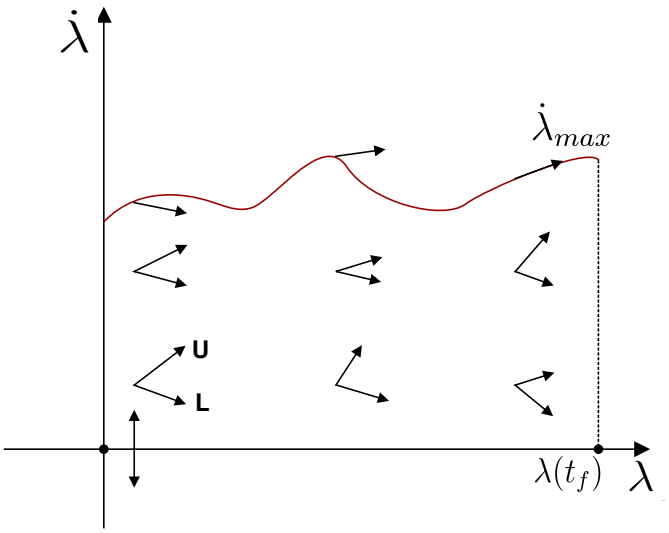

Figure 2.8 – Representation of *pseudo-acceleration* limits as arrows in phase plane plot

It is possible to make an analysis of the feasible trajectories in a 3-D phase space too, whose axes are λ, $\dot{\lambda}$, $\ddot{\lambda}$, as showed in Fig. 2.9. This plot gives a poor indication about the admissible trajectories, but explains the accelerations capability of the manipulator at each point [43]. Besides, it becomes very useful when jerk constraints are added to optimization problem. In fact, to avoid violations of jerk constraints, it has been demonstrated that the time optimal trajectory follows successive switches between maximum and minimum jerk, instead of *pseudo-acceleration*, showing a *bang-bang* profile in jerk [52]. So, if no singularities occurs, the profile will have a *max-min-max* structure, as showed in Fig. 2.9.

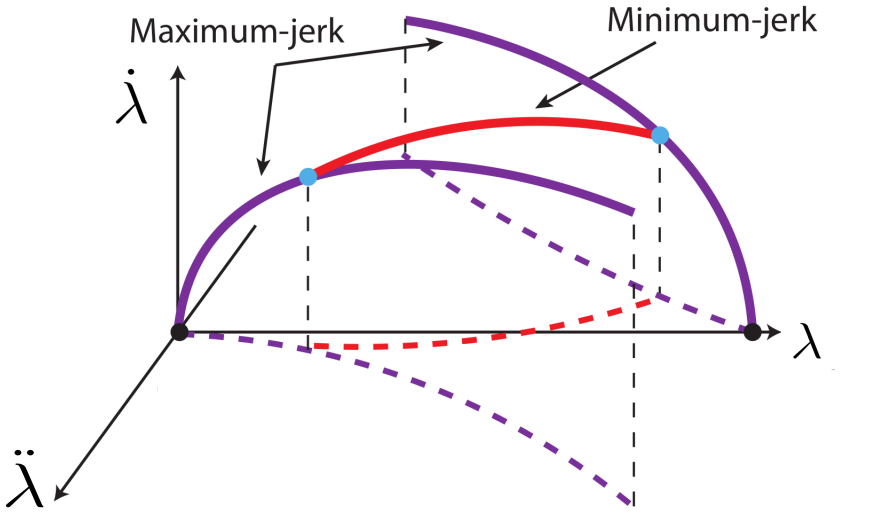

Figure 2.9 – Three-dimensional phase space with jerk limits [44]

2.6.2 Analysis of $\dot{\lambda}^2 - \ddot{\lambda}$ plane

Taking into account the parametrized dynamics (2.28) and the constraint on torque limits, the following inequalities yield:

$$\begin{aligned} m_i(\lambda)\ddot{\lambda} + s_i(\lambda)\dot{\lambda}^2 &\leq \tau_{i,max} - g_i(\lambda) \\ m_i(\lambda)\ddot{\lambda} + s_i(\lambda)\dot{\lambda}^2 &\geq \tau_{i,min} - g_i(\lambda) \end{aligned} \tag{2.56}$$

For a fixed λ and for each joint, it is possible to represent these limits as two straight lines into the $\dot{\lambda}^2 - \ddot{\lambda}$. The intersection of these lines creates a polygon which characterizes the feasible *pseudo-accelerations*. For each λ, since the parameters a_i and b_i vary, the shape of polygon changes.

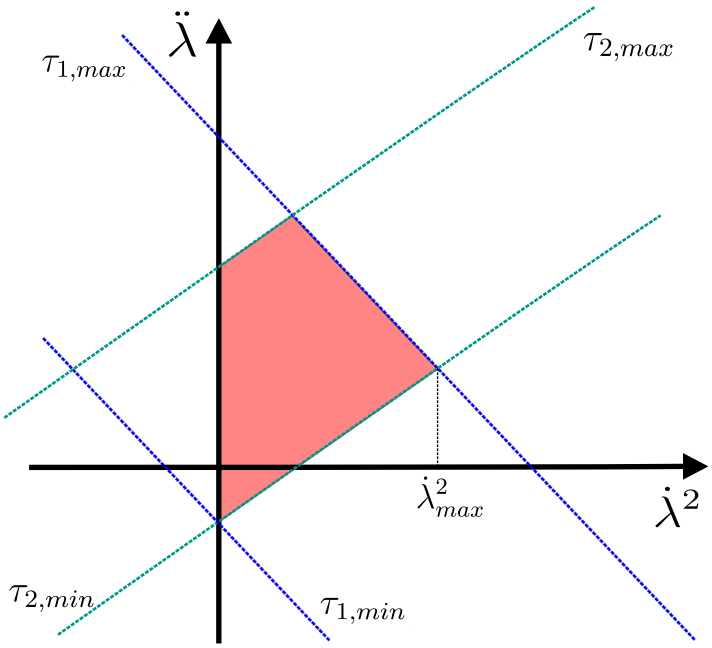

Figure 2.10 – Polygon of feasible *pseudo-acceleration* in the $\dot{\lambda}^2 - \ddot{\lambda}$ plane

This plot gives useful information about the admissible *pseudo-velocities* and *pseudo-accelerations*. Since the *pseudo-accelerations* take always the minimum or maximum value, for each $\dot{\lambda}$, it will be equal to the corresponding upper or lower bound of the polygon along $\ddot{\lambda}$. Besides, the right bound of $\dot{\lambda}$ represents the maximum velocity admissible at fixed λ and it represents the conditions when $L(\lambda, \dot{\lambda}) = U(\lambda, \dot{\lambda})$ because only one *pseudo-accelerations* is allowed [43]. This point is unique unless the straight line of the bounding actuator is parallel to the $\dot{\lambda}$-axis. When this condition happens, the $\ddot{\lambda}$ value at maximum $\dot{\lambda}^2$ is not unique anymore, as in Fig. 2.11. It occurs when the bounding joint is characterized by $a_i(\lambda) = 0$, which means that the torque does not contribute to the *pseudo-acceleration*. Indeed, the (2.56) equation becomes:

$$\begin{aligned} s_i(\lambda)\dot{\lambda}^2 &\leq \tau_{i,max} - g_i(\lambda) \\ s_i(\lambda)\dot{\lambda}^2 &\geq \tau_{i,min} - g_i(\lambda) \end{aligned} \tag{2.57}$$

So only $\dot{\lambda}^2$ is limited by the torque constraint, as explained in Sect. 2.5. This point is called *critical* and it assumes a key role in planning, as will be explained later.

2.6.3 Maximum velocity curve

In order to plan an optimal time trajectory through the phase-plane plot, the maximum velocity curve is required. Under the assumption that no joint has

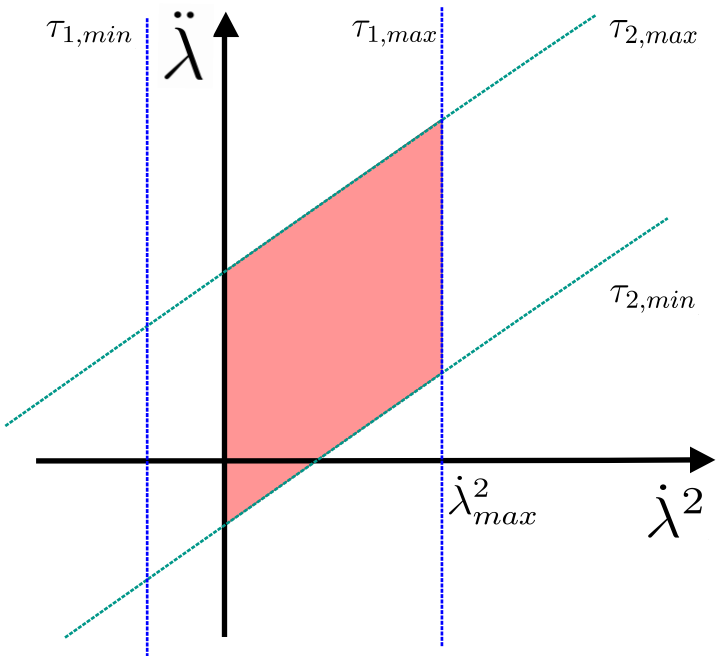

Figure 2.11 – Polygon of feasible *pseudo-acceleration* in the $\dot{\lambda}^2 - \ddot{\lambda}$ plane with a critical point

$a_i(\lambda) = 0$, it is possible to compute it by looking for the $\dot{\lambda}^2$ values which satisfy the following condition:

$$L(\lambda, \dot{\lambda}_{max}) = U(\lambda, \dot{\lambda}_{max}) \tag{2.58}$$

where $L = L_i(\lambda, \dot{\lambda})$ is the lower limit corresponding to i-th joint, while $U = U_j(\lambda, \dot{\lambda})$ is the upper limit corresponding to j-th joint. If the complete dynamics equation (2.7) is assumed, (2.41) and (2.42) can be substituted in (2.58), obtaining [48]:

$$\dot{\lambda}^2_{max} = \min_{i,j}\left\{\max_{\tau_i,\tau_j}\left(\frac{m_j(\tau_i - g_i(\lambda)) - m_i(\tau_j - g_j(\lambda))}{m_j s_i - m_i s_j}\right)\right\} \tag{2.59}$$

where i and j span all the joint indices. If $m_i s_j - m_j s_i = 0$, it means that the i-th and j-th actuators do not contribute to the maximum velocity bound. If the bounding joint has $m_i(\lambda) = 0$, the maximum velocity curve at λ is obtained as the upper limit of (2.38).

Assuming to use the reduced dynamics equation (2.8), it is not possible to compute the MVC using the equality (2.58) because the term $\boldsymbol{h}(\boldsymbol{q}(\lambda), \dot{\boldsymbol{q}}(\lambda, \dot{\lambda}))$ depends on $\dot{\lambda}$ itself. A solution could be transforming the search for the maximum

velocity curve into a linear programming problem, as follows [32]:

$$\begin{aligned} \max \quad & \dot{\lambda}^2 \\ \text{s.t.} \quad & a_i(\lambda)\ddot{\lambda} + b_i(\lambda)\dot{\lambda}^2 + h_i(q(\lambda), \dot{q}(\lambda, \dot{\lambda})) - \tau_{i,max} \leq 0, \\ & -a_i(\lambda)\ddot{\lambda} - b_i(\lambda)\dot{\lambda}^2 - h_i(q(\lambda), \dot{q}(\lambda, \dot{\lambda})) + \tau_{i,min} \leq 0 \end{aligned} \tag{2.60}$$

where $\ddot{\lambda}$ and $\dot{\lambda}^2$ are the state variables. It has to be solved for each value of λ. The solution of the optimization problem allows to cover also the case in which $a_i(\lambda) = 0$ [31].

Another method to compute the MVC is based on the analysis of the $\dot{\lambda}^2 - \ddot{\lambda}$ phase. It is computed as collection of $\dot{\lambda}$ values, which are the right bound of the polygon on the plot $\dot{\lambda}^2 - \ddot{\lambda}$ for each value of λ [43].

2.6.4 Switching Points method

Thanks to Extended Pontryagin's Maximum Principle, it has been demonstrated the *pseudo-acceleration* takes always the maximum or minimum value. So, finding the optimal time trajectory is equivalent to look for the points in which the changes from acceleration to deceleration happen and viceversa. These points are called *switching points*.

The MVC assumes an important role during the search for the switching points. Excluding the points in which the the bounding joint has $m_i(\lambda) = 0$, the *pseudo-acceleration* is unique on the maximum velocity curve and can point either towards the infeasible region, violating the velocity limit, or towards the feasible region. The former case is called *trajectory sink*, while the latter one is called *trajectory source* [43]. The only points in which the trajectory can touch the maximum velocity curve are those where a switch from *trajectory sink* to *trajectory source* happen, corresponding to a change from acceleration to deceleration. The other switching points, corresponding to a switch from deceleration to acceleration, do not belong to maximum velocity curve and are found through integration, as will be explained later.

The switching points from *sink* to *source* can be divided into three main categories [48]:

- **Discontinuity point**: Points where the maximum velocity curve is discontinuous. It is where the path changes in a discontinuous way [51].
- **Tangent point**: Points where the trajectory is tangent to the maximum velocity curve.
- **Critical point**: Points where the inertia of one joint is equal to zero and

the actuator does not actually contribute to the *pseudo-acceleration*, which is not uniquely determined.

The three types of switching point can be seen on the figure 2.12, respectively as *a, b* and *c* points. The first two categories are easy to handle because the *pseudo-acceleration* remains unique, while the last one needs a further analysis. In fact, the critical points can become *singular points*, which means that the limits of *pseudo-acceleration* point towards or comes from the infeasible region delimited by the maximum velocity curve. So the full interval of accelerations is not really allowed. In Fig. 2.12 it is possible to see the case when a critical point becomes a singular one: *c* and *d* points are an example.

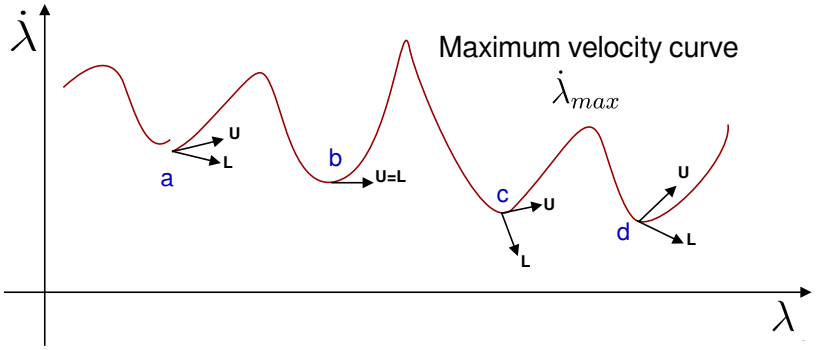

Figure 2.12 – Representation of discontinuous (*a*), tangent(*b*) and singular point (*c,d*) on the MVC

So, in correspondence of a singular point, the maximum feasible *pseudo-acceleration* or minimum *pseudo-acceleration* are chosen such that the maximum velocity constraint is not violated, so tangent to the curve, as follows [48]:

$$\ddot{\lambda}_{max,f} = \dot{\lambda}_{max} \frac{d\dot{\lambda}_{max}(\lambda^+)}{d\lambda} \tag{2.61}$$

if U points to the infeasible region, as in point d, while

$$\ddot{\lambda}_{min,f} = \dot{\lambda}_{max} \frac{d\dot{\lambda}_{max}(\lambda^-)}{d\lambda} \tag{2.62}$$

if L enters into the feasible region from the infeasible one, as in point c.

Once defined the types of the possible switching points, an algorithm to find them has to be designed. First, let us consider a simple problem with only one switching [20]. At the beginning, the manipulator starts with the maximum acceleration, so $\ddot{\lambda} = U$. After a while, it will meet a switching point and it will pass to the maximum deceleration, so $\ddot{\lambda} = L$. In order to find this point, it is possible to integrate forward the trajectory from the beginning with the

maximum acceleration and to integrate backward from the end with the maximum deceleration. The two curves will intersect at a certain point, which becomes the switching point. In the figure 2.13, λ_a is the switching point.

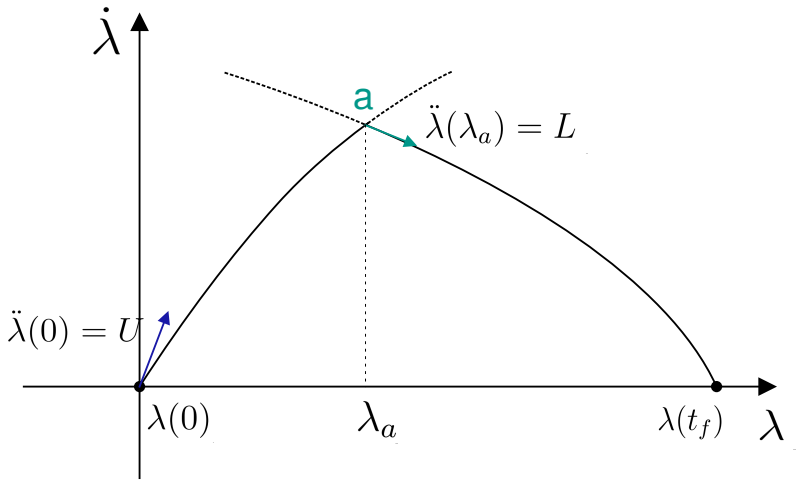

Figure 2.13 – Optimal trajectory with single switching point

When there are multiple switching points, the problem becomes more complex. The method is showed in Fig. 2.14. It is necessary when the switching point found in the previous example lies in the infeasible region of phase plane plot. So, to not overcome the feasible region, more switches are needed. [20] proposes the following algorithm: starting from the initial point, integrate forward with the maximum acceleration until the MVC is met. From this point, the velocity is decreased on a vertical line: from a generic point integrate forward until the MVC or the λ-axis is found. The aim is to find a point where the integration curve meets tangentially the MVC, by using a *trial and error* approach. When this point (*point d*) is found, it is integrated backward to meet the first curve. The crossing point is the first switching point (*point a*). Then, from the point tangent to the MVC (*point b*), it is integrated forward until the MVC is met again (*point c*) or until the backward integration curve from the final point is crossed. The method is iterative until the last switching point is found and the final state is reached. In fig. 2.14, λ_a, λ_b and λ_c are the three switching points.

Other algorithms are proposed by several authors. For example, [48] uses the maximum velocity curve and the critical points to find the switching point, instead of the iterative approach.

It is worth to underline that these integration methods could present some troubles in the neighbourhood of the switching point. In fact, at a switching point, the trajectory in phase plane will present a discontinuity. In general, this does not present an issue, but in the critical points.

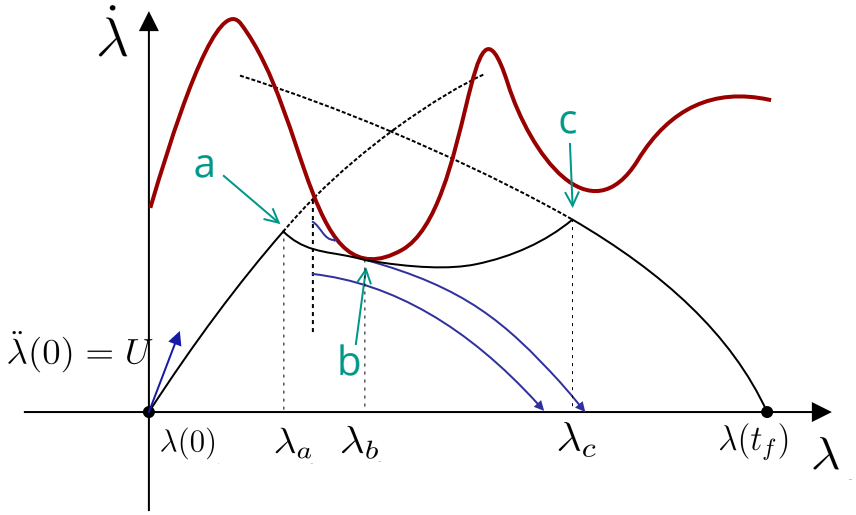

Figure 2.14 – Resolution with multiple switching points: a, b and c are the three switching points, the red curve is the MVC and the blue curves are those obtained through integration during the *trial and error* phase.

Since the phase plane trajectory is obtained through integration, it is possible to analyze the possible trends using the fields generated by the maximum and minimum *pseudo-accelerations*, as it is possible to see in Fig. 2.15. In fact, the optimal trajectory will follow these profiles. In the neighbourhood of the critical points, these fields diverge: so theoretically the trajectory would unlikely reach it. The issue is generated because the described integration methods start from this point to generate the phase plane trajectory, enforcing the passage. This could generate a jitter in the torque profile [45].

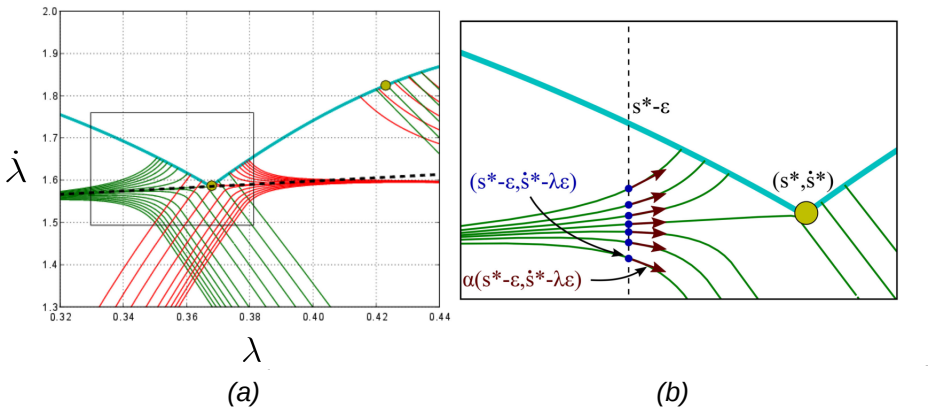

Figure 2.15 – (a) fields of the *pseudo-acceleration* L (green curves) and U (red curves) in the neighbourhood of the switching points; (b) zoom of the black box of (a) [45]

Chapter 3

Resolution with dynamic programming

One among the proposed resolutive techniques for the time optimal planning is the dynamic programming. It has been demonstrated that dynamic programming, or *DP*, easily handles several types of cost functions, allowing also the inclusion of more performance indices, and places few restrictions on the constraints. The aforementioned characteristics makes it flexible and advantageous for the time optimal planning problem. The drawback is that DP demands more memory occupancy and computational effort with respect to the other proposed methods.

Because of its versatility, it covers an extended class of optimization problems, also in the robotic field: an application example is the redundancy resolution. A DP-inspired algorithm for the redundancy resolution has been designed and implemented in ALTEC framework and represents the starting point of the work: indeed, the two modules, i.e. the redundancy resolution and the time optimal planning one, have several features in common, which can be joined to create a single component, representing the high level algorithm, applicable to several semantics.

In this chapter the DP method and the underlying theory will be described, then the redundancy resolution application is analyzed and, finally, the design of the time-optimal planner is described.

3.1 Dynamic programming method

Dynamic programming is an optimization method. It is based on the Bellman principle, called *"principle of optimality"* [19], and it states that:

An optimal policy has the property that whatever the initial state and

initial decisions are, the remaining decision must constitute an optimal policy with regard to the state resulting from the first decisions.

Let's explain this principle in a simple graphical and practical way:

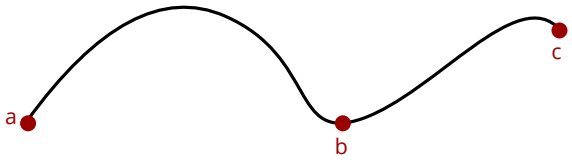

Figure 3.1 – Example of optimal path

The *principle of optimality* states that if the path from point **a** to point **c** is optimal, also the subpath from point **b** to point **c** will be optimal. This is the core of the dynamic programming approach: the original problem is divided into subproblems and the local optimal solutions for each subproblem are found in order to obtain the global one. In fact, the local solutions can be combined to find the one of the original problem due to the principle of optimality.

The dynamic programming method is characterized by three elements [21]:

- **Stage**: The subproblem defined earlier represents the so-called stage. Each stage is characterized by a *policy decision* [35] and it can be solved as an independent optimization problem. In fact, one stage at a time is solved sequentially.

- **State**: Each stage is associated to a group of states. Each state is the set of parameters needed to solve the optimization problem and represents a possible condition in which the system can be at the corresponding stage. The number of states should be as low as possible in order to decrease the computational effort and complexity, but, at the same time, it should be as high as possible to increase the quality of the solution. In fact, the optimality condition depends on the resolution of states. The specification of the states and the state variables resolution is one of the most critical step of the dynamic programming, since a trade-off between complexity and solution quality has to be found.

  For each decision at each stage, the state is updated, becoming the state for the next stage. Finally, each state is independent from the older ones; so each decision is made without taking account the previous decisions and stages. This is a result of *principle of optimality*.

- **Recursion**: The solutions of stages are linked through a recursive relation: in fact each optimization problem can be independently solved, but the global solution must include the optimal decisions taken previously. In fact the

global cost of the n-th *stage* includes all the optimal costs of the $i = 0, ..., n-1$ stages.

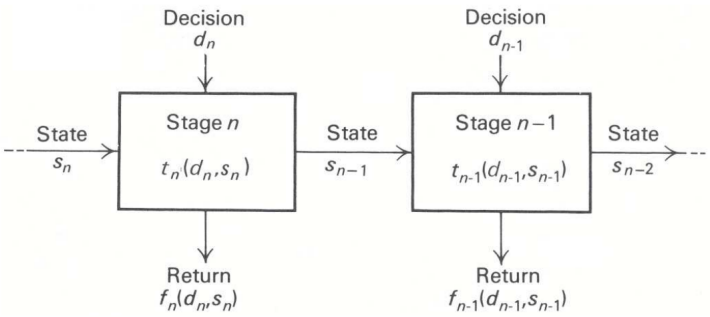

Figure 3.2 – Stage transition process [21]

So the dynamic programming method can be described in the following way. The aim is to minimize or to maximize a defined cost function I: it represents the *cost* of the sequence of the stages, as the sum of local costs. Let us define s_i as the $i-th$ state and d_i as the $i-th$ decision, or input. The return, or local cost, determined by the d_i decision at s_i stage is given by $f_i(d_i, s_i)$; while the next state is given by $t_i(d_i, s_i)$. The transition process is showed in Fig. 3.2.

Assuming we want to reach the n-th state, passing n stages, the optimization problem is equivalent to find the n decisions $d_{n-1}, d_{n-2}, ..., d_0$ to maximize (or minimize) a cost function I at stage n [21]:

$$I(n) = \max(f(d_n, s_n) + f(d_{n-1}, s_{n-1}) + ... + f(d_0, s_0)) \tag{3.1}$$

Since $f(d_n, s_n)$ depends only on (d_n, s_n) and not on the previous variables, (3.1) can be rewritten as:

$$I(n) = \max(f(d_n, s_n) + \max(f(d_{n-1}, s_{n-1}) + ... + f(d_0, s_0))) \tag{3.2}$$

The second term is equal to the cost function $I(n-1)$ evaluated at $n-1$-th stage. So the following recursive relation is obtained:

$$I(n) = \max(f(d_n, s_n) + I(n-1)) \tag{3.3}$$

This equation is the expression of the principle of optimality.

In order to handle the complexity of the problem, a discretization of the state space system is advised, although generally it is not a requirement. This approach transforms the *Bellman principle* into a search on a graph: the optimal solution

is found through a search of the consecutive states in a grid which brings to the best cost [31]. Each cell of the grid is characterized by a state; the cost between two cells at consecutive stages has to be computed if the transition satisfies the defined constraints. The globally-optimal solution can be found evaluating the path, i.e. the sequence of cells, with maximum or minimum cost, depending on the type of optimization problem.

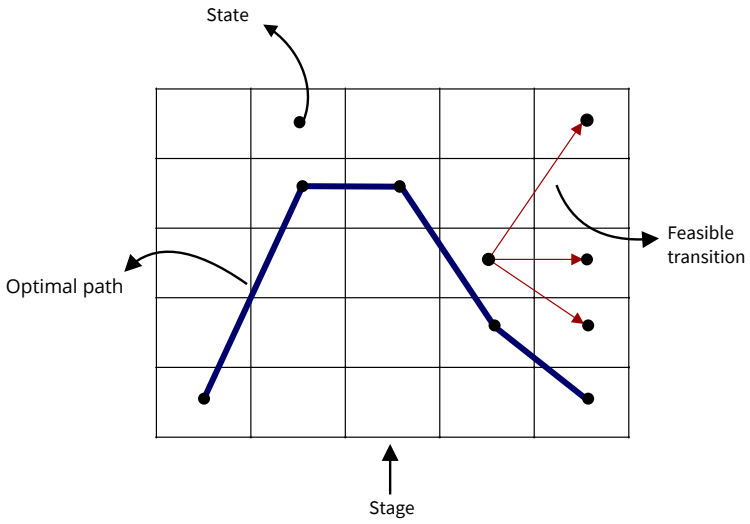

Figure 3.3 – Example of a grid explored by dynamic programming

It is worth to underline that in order for the dynamic programming to be applicable, the problem must be decomposable in sub-problems, whose solutions can be summed to find the global one.

3.2 Redundancy resolution with dynamic programming

The redundancy resolution is defined as the optimization of the inverse kinematic solutions for a redundant robot. Recently, the dynamic programming approach has been preferred among other methods in performing the redundancy resolution, because of the proven flexibility. Indeed, the possibility to optimize several performance indices and the accommodation of generic constraints is an useful characteristic also in redundancy resolution.

3.2.1 Redundancy kinematics

A manipulator is said *redundant* if it has more degrees of freedom than those requested by the task [49]. It is equivalent to say that, if the robot has n degrees of freedom and the task requires m degrees of freedom, the robot is redundant if $n > m$ and $r = n - m$ is said the degree of redundancy. So, the redundancy is not an intrinsic property of the robot, but it is related to the task required. In fact, even a robot with three revolute joints can be redundant if the task is defined only in position in a plane, while it becomes non redundant if the orientation is added to the task. If a robot has 7 or more joints, it is intrinsically redundant for a Cartesian pose.

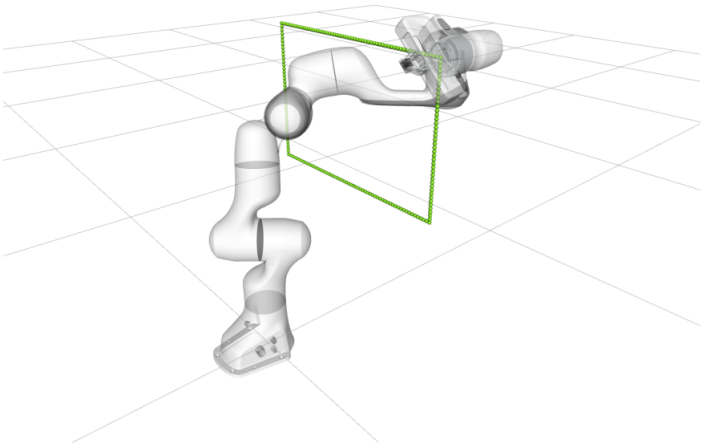

Figure 3.4 – Example of 7-DOF manipulator [22]: Panda by Franka-Emika [1]

The redundancy is useful because it improves the dexterity and the maneuverability. In fact, a redundant robot can reach the same point in the task space in infinite joints configurations, exploiting the additional redundant joints. In this way it becomes easier to avoid near obstacles or joint kinematic limits. It has also drawbacks: the algorithms for inverse kinematics functions and motion control are more complex. Researches have widely studied techniques to solve the redundancy problem.

Let us analyze the dynamic programming approach to redundancy resolution [31, 22].

3.2.2 Problem formulation

Assuming a trajectory $\boldsymbol{x}(t)$ is given in the task space with t belonging to an interval from 0 to T, it is possible to discretize the trajectory in $\boldsymbol{x}(i)$ with $i = 0, \ldots, N_i$ where $N_i = \frac{T}{\tau}$. The joint positions of the trajectory are computed

through the inverse kinematics function $f(q, u)$, obtaining the following discrete time system:

$$\boldsymbol{q}(i+1) = f(\boldsymbol{q}(i), \boldsymbol{u}(i)), \boldsymbol{q}(0) = q_0; \tag{3.4}$$

where $\boldsymbol{q}$ is the *joint vector* and $\boldsymbol{u}$ is the *input vector*. In general, the control inputs are limited: it possible to define the domains A_i and B_i, where $u(i)$ and its derivative $\dot{u}(i)$ must belong to respectively, as follows:

$$\begin{aligned} \boldsymbol{u}(i) &\in A_i \\ \dot{\boldsymbol{u}}(i) &\in B_i(\boldsymbol{u}(i)) \end{aligned} \tag{3.5}$$

The input u can be chosen as the position of one or more joints, which are called *redundant joints*, such that, fixed these values, the other joints positions can be exactly computed at each waypoint i. So the A_i and B_i domains represent respectively the feasible redundant joints positions and velocities.

At each i, it is possible to compute the admissible control input $\boldsymbol{u}(i)$ to reach $\boldsymbol{q}(i+1)$ from $\boldsymbol{q}(i)$, as the inputs belonging to A_i domain plus those ones which satisfy the velocity constraint. If $\dot{u}$ is computed through the forward finite differences, the admissible control inputs must belong to a domain C_i, as follows:

$$\boldsymbol{u}(i) \in C_i \tag{3.6}$$

$$C_i = A_i \cap \left\{ \dot{\boldsymbol{u}}(i) = \frac{\boldsymbol{u}(i+1) - \boldsymbol{u}(i)}{\tau} \in B_i, \boldsymbol{u}(i+1) \in A_{i+1} \right\} \tag{3.7}$$

The aim is to find the optimal sequence of inputs that minimizes or maximizes a defined cost function, using the dynamic programming method. The cost function is assumed to be [31]:

$$I(0) = \psi(\mathbf{q}(N_i)) + \sum_{j=0}^{N_i-1} l(\mathbf{q}(j), \dot{\mathbf{q}}(j), \mathbf{u}(j), \dot{\mathbf{u}}(j)) \tag{3.8}$$

where $\psi(\mathbf{q}(N_i))$ is the final cost. Using the forward finite differences method, the dependence on $\dot{\mathbf{u}}$ and $\dot{\mathbf{q}}$ is substituted by $\boldsymbol{u}(i+1)$ and $\boldsymbol{q}(i+1)$. The cost function can be set in a recursive way as follows:

$$I(i) = I(i+1) + l(\mathbf{q}(i), \mathbf{q}(i+1), \mathbf{u}(i), \mathbf{u}(i+1)) \tag{3.9}$$

This cost function can be used in dynamic programming. Assuming that the objective is to minimize the cost, the cost function becomes:

$$I(i) = \min_{u \in C_i} (I(i+1) + l(\mathbf{q}(i), \mathbf{q}(i+1), \mathbf{u}(i), \mathbf{u}(i+1))) \tag{3.10}$$

where $I(N_i) = \psi(\mathbf{q}(N_i))$ is the initial cost.

In order to transform the dynamic programming approach into a grid search problem, a discretization of the control input and the constraint set is performed, obtaining a discrete state space. The result of such a discretization process is that the optimal solution corresponds to the sequence $\boldsymbol{u}(j)$ which carries the minimum cost from the initial state to the final one [31].

For the sake of simplicity, let us assume to have only one redundancy parameter, that is $r = n - m = 1$. Each stage corresponds to a *waypoint* of the trajectory, whereas the state is the joint configuration corresponding to the redundant joint position, which is the control input. So each cell of the grid is related to a joint positions vector:

$$\mathbf{q}(i,j) = f^{-1}(\mathbf{x}(i), u(j)) = \begin{bmatrix} q_1(i,j) \\ q_2(i,j) \\ \dots \\ q_n(i,j) \end{bmatrix} \tag{3.11}$$

where i is the index for the stage, while j is the index for the state at the corresponding stage. A transition between two states can be done if the constraints are satisfied and the local cost can be computed.

In a redundant robot, once the redundant joints positions are set, the inverse kinematic function (3.11) admits a finite set of solutions. Each solution of the kinematic functions belongs to a subspace of the joint configuration space: the joint subspace in which it is possible to find *one and only one* solution of the inverse kinematics, fixed the redundant joint positions, is called *extended aspect* [42]. The number of extended aspects depends on the mechanical structure of the robot and it is fixed: 2 for planar, 2 for spherical, 4 for regional, 16 for spatial [31].

In order to find a globally optimal solution, which considers the total configuration space of the robot, each inverse kinematics solution of the corresponding extended aspect has to be included in the problem. So, multiple grids must be built and each grid deals with one extended aspect. Now the transitions could happen also between states of different grids; the constraints to be respected are not changed. So, the globally-optimal solution is found looking for the best path from the initial state to the final one across the entire set of grids. The globally-optimal solution corresponds to the sequence of joint positions which minimizes or maximizes a cost function along the desired trajectory. The explanation of the entire algorithm can be found in [31].

3.3 Design of a time-optimal planner using dynamic programming

The dynamic programming approach for time optimal planning has been already dealt by different authors [37, 50, 28].

The great advantage of DP approach is its versatility: in fact it can deal with different performance indices in planning, such as energy, time or also an union, and it can cover a big class of constraints, such as the jerk ones. Besides, if the path is parametrized, the dimension of the problem passes from $2n$ for a n-links manipulator, to 2, avoiding the so-called *curse of dimensionality*, which affects systems with high number of states, increasing exponentially the computational effort [37].

It is not necessary to use the parametrized path as state variable: [50] proposes an algorithm which uses joint positions instead of λ, starting from the assumption that for each waypoint of the trajectory, given the value of one non-stationary joint, it is possible to obtain the entire joints configuration through the inverse kinematics function. In this way it is possible to apply the method also to paths which may not be parametrized and at the same moment *the curse of dimensionality* is avoided. [37] proposes a method in which the state variables are $(\lambda, \dot{\lambda})$, the same of the *switching point method*. Our algorithm is inspired to that one described in [37].

3.3.1 Grid representation

Let us consider $(\lambda, \dot{\lambda})$ as the state variable. In order to apply the DP as a grid-search, a discretization of state space is required.

The maximum value of the *pseudo-velocity* $\dot{\lambda}_{MAX}$ is defined as the maximum value in the Maximum Velocity Curve or the upper limit of any feasible trajectory, as follows:

$$\dot{\lambda}_{MAX} = \max_{\lambda} \dot{\lambda}_{max}(\lambda) \tag{3.12}$$

The MVC often assumes relevant peaks, so the $\dot{\lambda}_{MAX}$ could be very high with respect to the optimal time trajectory. Therefore, the resolution of the $\dot{\lambda}$-domain could be not dense enough in the interested interval [32]. The $\dot{\lambda}_{MAX}$ could be taken to a lower value than the true maximum one and it could be increased if, after planning, the limit is hit one or more times. The same could be done if the MVC is not known a priori: using a *trial-and-error* approach, an adequate value of $\dot{\lambda}_{MAX}$ could be found.

Assuming that:

- λ is comprised in the interval $\lambda \in [0, \Lambda]$ and Δ_λ is the sampling size for λ-domain. So N_i is the total number of λ samples:

$$N_i = \frac{\Lambda}{\Delta_\lambda}$$

- $\dot{\lambda}$ is comprised in the interval $\dot{\lambda} \in [0, \dot{\lambda}_{MAX}]$ and Δ_λ is the sampling size for $\dot{\lambda}$-domain. So N_j is the total number of $\dot{\lambda}$ samples:

$$N_j = \frac{\dot{\lambda}_{MAX}}{\Delta_{\dot{\lambda}}}$$

It is possible to define the λ and $\dot{\lambda}$ samples as:

$$\begin{aligned} \lambda(i) &= i\Delta_\lambda \text{ with } i = 0,1,2,\dots,N_i \\ \dot{\lambda}(j) &= j\Delta_{\dot{\lambda}} \text{ with } i = 0,1,2,\dots,N_j \end{aligned} \tag{3.13}$$

So the grid for the dynamic programming is composed by N_i columns and N_j rows and each cell is characterized by the state $(\lambda(i), \dot{\lambda}(j))$.

The input control of our system is the *pseudo-acceleration* $\ddot{\lambda}$, which corresponds to the so-called *decision*. The transitions between states are limited by constraints on $\ddot{\lambda}$. In fact, for each cell of the grid, defined $\lambda(i)$ and $\dot{\lambda}(j)$, it is possible to identify the lower limit $L(\lambda(i), \dot{\lambda}(j))$ and upper limit $U(\lambda(i), \dot{\lambda}(j))$ of the *pseudo-acceleration* from (2.39) and (2.40).

The set of constraints on states and input controls have to be defined. The stage-dependent domains A_i and B_i, where the admissible *pseudo-velocities* $\dot{\lambda}(i)$ and admissible the *pseudo-accelerations* $\ddot{\lambda}(i,j)$ must belong to, respectively, can be defined as follows:

$$\begin{aligned} \dot{\lambda}(i) &\in A_i \\ \ddot{\lambda}(i,j) &\in B_i(\dot{\lambda}(j)) \end{aligned} \tag{3.14}$$

The domain A_i is composed by the $\dot{\lambda}$ values below the MVC for each λ, while the domain B_i is composed by the $\ddot{\lambda}$ values inside the interval of feasible *pseudo-accelerations*, defined as $[L(i,j), U(i,j)]$. It is worth to underline that the maximum velocity curve condition can be checked through the following condition:

$$L(i,j) \leq U(i,j) \tag{3.15}$$

It derives from the definition of maximum velocity curve, explained in Sec. 2.6.3. If the inequality is true, $\dot{\lambda}(i,j)$ is below the MVC and belongs to A_i domain.

As it is done for redundancy resolution, using an approximation to compute $\ddot{\lambda}$ from $\dot{\lambda}$, it is possible to define a domain C_i, which includes both the constraints and $\dot{\lambda}(i)$ must belong to. In fact, it represents the domain of states that can be reached from at least one state of the previous stage without violating the constraint of *pseudo-acceleration*. It will be defined later, when the approximations will be analyzed.

Assuming to use the forward recursive procedure and defining $\boldsymbol{s}$ as the state, the general cost function is defined as:

$$I(N_i) = \psi(s_0) + \sum_{k=1}^{N_i} l(\boldsymbol{s}(k-1), \boldsymbol{s}(k))) \tag{3.16}$$

where $\psi(s_0)$ is the cost associated to the initial state and $l(\boldsymbol{s}(k-1), \boldsymbol{s}(k)))$ is the local cost computed between the states $\boldsymbol{s}(k-1)$ and $\boldsymbol{s}(k)$ of two consecutive stages.

Since the aim is to minimize the time, the local cost is defined as the time traveled between the states, called $\Delta_t(k)$. It can be defined as function of λ and $\dot{\lambda}$, as we will define later. Besides, we assume that the initial state is equal to the rest condition, so $\dot{\lambda}(0)$.

So the cost function is equal to:

$$t(N_i) = t(0) + \sum_{k=1}^{N_i} \Delta_t(k-1) \text{ where } \Delta_t(k-1) = t(k) - t(k-1) \tag{3.17}$$

which can be rewritten in the following recursive way:

$$\begin{aligned} t(0) &= 0 \\ t(i) &= t(i-1) + \Delta_t(i-1) \end{aligned} \tag{3.18}$$

It is worth to say that the backward recursive procedure could be used too.

At each stage of the algorithm, the optimal cost at stage i is found as:

$$\begin{aligned} t(0) &= 0 \\ t_{opt}(i) &= \min_{\dot{\lambda}(i) \in C_i} (t(i-1) + \Delta_t(i-1)) \end{aligned} \tag{3.19}$$

The globally optimized function will be $t_{opt}(N_i)$.

3.3.2 Discrete approximation of local cost function

The local cost corresponds to the traveled time between two consecutive states, defined as Δ_t. Different methods could be used to approximate it.

Forward finite differences approximation

Using the forward finite differences approximation to compute $\lambda(i+1)$:

$$\lambda(i+1) = \lambda(i) + \Delta_t \dot{\lambda}(i) \tag{3.20}$$

the traveled time Δ_t can be computed as:

$$\Delta_t = \frac{\lambda(i+1) - \lambda(i)}{\dot{\lambda}(i)} \tag{3.21}$$

The cost would be infinite when $\dot{\lambda}(i) = 0$. The same would happen if the backward finite differences approximation was used.

Trapezoidal approximation or Crank-Nicholson Method

Using the trapezoidal approximation to compute $\lambda(i+1)$:

$$\lambda(i+1) = \lambda(i) + \frac{\Delta_t}{2}(\dot{\lambda}(i+1) + \dot{\lambda}(i)) \tag{3.22}$$

the traveled time Δ_t can be computed as:

$$\Delta_t = 2\frac{\lambda(i+1) - \lambda(i)}{\dot{\lambda}(i) + \dot{\lambda}(i+1)} \tag{3.23}$$

In this way, Δ_t is always a finite value, even when $\dot{\lambda}(i) = 0$.

This approximation is used in planning.

3.3.3 Discrete approximation of *pseudo-accelerations*

In order to verify the constraints on *pseudo-accelerations*, $\ddot{\lambda}$ must be computed. Different methods could be used to approximate $\ddot{\lambda}$.

Chain derivative rule

The *pseudo-acceleration* $\ddot{\lambda}$ is the derivative of $\dot{\lambda}$ with respect to time. It is defined as:

$$\ddot{\lambda} = \frac{d\dot{\lambda}}{dt} \tag{3.24}$$

Using the chain derivative rule, it can be rewritten as:

$$\ddot{\lambda} = \frac{d\dot{\lambda}}{dt} = \frac{d\dot{\lambda}}{dt}\frac{d\lambda}{d\lambda} = \frac{\dot{\lambda}}{d\lambda}\frac{d\lambda}{dt} = \frac{d\dot{\lambda}}{d\lambda}\dot{\lambda} \tag{3.25}$$

Using the forward finite differences approximation, it becomes:

$$\ddot{\lambda}(i) = \frac{\dot{\lambda}(i+1) - \dot{\lambda}(i)}{\lambda(i+1) - \lambda(i)}\dot{\lambda}(i) \tag{3.26}$$

It would become equal to zero when $\dot{\lambda}(i) = 0$. To avoid this condition, it could be applied the approximation proposed by [20], which is:

$$\ddot{\lambda} = \frac{d\dot{\lambda}}{d\lambda} \tag{3.27}$$

which becomes:

$$\ddot{\lambda}(i) = \frac{\dot{\lambda}(i+1) - \dot{\lambda}(i)}{\lambda(i+1) - \lambda(i)} \tag{3.28}$$

It is possible to demonstrate that this approximation could generate additional inaccuracies during planning.

Forward finite differences with time computation

Using the forward finite differences approximation and (3.23), it is possible to compute $\ddot{\lambda}(i)$ as:

$$\begin{aligned} \ddot{\lambda}(i) &= \frac{\dot{\lambda}(i+1) - \dot{\lambda}(i)}{\Delta_t} = \\ &= \frac{\dot{\lambda}(i+1) - \dot{\lambda}(i)}{\frac{2(\lambda(i+1)-\lambda(i))}{\dot{\lambda}(i)+\dot{\lambda}(i+1)}} = \\ &= \frac{\dot{\lambda}^2(i+1) - \dot{\lambda}^2(i)}{2(\lambda(i+1) - \lambda(i))} \end{aligned} \tag{3.29}$$

It is possible to demonstrate that, using this approximation, more accurate results in planning phase are obtained.

This approximation is used in planning. So, the C_i domain becomes:

$$C_{i+1} = A_{i+1} \cap \left\{ \dot{\lambda}(i) : \ddot{\lambda}(i+1) = \frac{\dot{\lambda}(i+1) - \dot{\lambda}(i)}{\Delta_t} \in B_{i+1}(\dot{\lambda}(i)), \dot{\lambda}(i) \in A_i \right\} \tag{3.30}$$

3.3.4 Time discretization of joint variables

Since the algorithm is based on a discrete formulation of the problem, the joint variable discretization assumes an important role in the design of the planner. Indeed, it could affect drastically the planning results. This problem occurs since, at the beginning of the planning, the joints' velocities $\dot{\boldsymbol{q}}(i)$ and accelerations $\ddot{\boldsymbol{q}}(i)$ are computed through the discrete values of the parametric variables, which are

the inputs of the problem: if a wrong expression is used to compute $\ddot{\boldsymbol{q}}(i)$, it may happen that the velocity at the next sample $\dot{\boldsymbol{q}}(i+1)$, given as integration of $\ddot{\boldsymbol{q}}(i)$ from initial condition $\dot{\boldsymbol{q}}(i)$, does not correspond to the same assumed expression. This difference could have negative effects on the validation of the planning, because the joints' velocities and accelerations used in planning are not longer related through a derivative/integrative relationship. Since the joints' velocities and accelerations values are used to compute the parametrized dynamics, on which the algorithm is based, errors are generated affecting the correctness of the results. This concept will be explained better in the following paragraphs.

Two different formulations to compute the discrete joints' velocities and accelerations are described.

First formulation

A natural choice to discretize Eqs. (2.18) and (2.19) would be the following:

$$\begin{aligned} \dot{\boldsymbol{q}}(i) &= \boldsymbol{q}'(i)\dot{\lambda}(i) \\ \ddot{\boldsymbol{q}}(i) &= \boldsymbol{q}'(i)\ddot{\lambda}(i) + \boldsymbol{q}''(i)\dot{\lambda}^2(i) \end{aligned} \tag{3.31}$$

where the joints' velocities and accelerations at sample i are directly computed from the related variables evaluated at the same sample. The vectors of parametrized dynamics are:

$$\begin{aligned} \boldsymbol{a}(i) &= \boldsymbol{q}'(i)\boldsymbol{H}(i) \\ \boldsymbol{b}(i) &= \boldsymbol{q}''(i)\boldsymbol{H}(i) \end{aligned} \tag{3.32}$$

If we compute the velocity at the sample $i+1$:

$$\begin{aligned} \dot{\boldsymbol{q}}(i+1) &= \dot{\boldsymbol{q}}(i) + \Delta_t \ddot{\boldsymbol{q}}(i) \\ &= \dot{\boldsymbol{q}}(i) + \Delta_t(\boldsymbol{q}'(i)\ddot{\lambda}(i) + \boldsymbol{q}''(i)\dot{\lambda}^2(i)) = \\ &= \dot{\boldsymbol{q}}(i) + \Delta_t\Big(\boldsymbol{q}'(i)\Big(\frac{\dot{\lambda}(i+1) - \dot{\lambda}(i)}{\Delta_t}\Big) + \Big(\frac{\boldsymbol{q}'(i+1) - \boldsymbol{q}'(i)}{\lambda(i+1) - \lambda(i)}\frac{\dot{\lambda}(i+1) - \dot{\lambda}(i)}{\Delta_t}\Big)\dot{\lambda}(i)\Big) = \\ &= \dot{\boldsymbol{q}}(i) + \boldsymbol{q}'(i)\dot{\lambda}(i+1) - \boldsymbol{q}'(i+1)\dot{\lambda}(i) + \boldsymbol{q}'(i+1)\dot{\lambda}(i) - \boldsymbol{q}'(i)\dot{\lambda}(i) = \\ &= \dot{\boldsymbol{q}}(i) + \boldsymbol{q}'(i)\dot{\lambda}(i+1) - \boldsymbol{q}'(i+1)\dot{\lambda} + q'(i+1)\dot{\lambda}(i) - \boldsymbol{q}'(i)\dot{\lambda}(i) = \\ &= \boldsymbol{q}'(i)\dot{\lambda}(i+1) \end{aligned} \tag{3.33}$$

which is incoherent with the expression computed previously in (3.31). So, the velocity sample at $i+1$ will be not equal to the pre-computed velocity sample.

The discretized variables from this formulation are very simple to obtain, but it

has been demonstrated that they don't return the right expressions of parametric velocity in (3.33). It could possibly be used when this inaccuracy is negligible for the final purpose of the computation. Since the dynamical model of a manipulator is very complex and nonlinear, this formulation should not be used for this task.

Second formulation

The joints' velocities and acceleration could be computed using the forward finite differences: in this way the discretization is implicitly included in the derivative.

The following discrete variables are the input of the planning:

- $\boldsymbol{q}(i)$ with $i = 0, ..., n$
- $\lambda(i)$ with $i = 0, ..., n$
- $\dot{\lambda}(i)$ with $i = 0, ..., m$

where n represents the number of waypoints, or equivalently the resolution of λ-domain, and m the number of the samples of $\dot{\lambda}$. The forward finite differences method is defined as follows:

$$\dot{x}(k) = \frac{x(k+1) - x(k)}{t(k+1) - t(k)} \tag{3.34}$$

Therefore, the derivative $\dot{q}$ at sample i can be computed as the derivative of q with respect to time, using the forward finite differences, as follows:

$$\dot{\boldsymbol{q}}(i) = \frac{\boldsymbol{q}(i+1) - \boldsymbol{q}(i)}{t(i+1) - t(i)} = \frac{\boldsymbol{q}(i+1) - \boldsymbol{q}(i)}{t(i+1) - t(i)} \frac{\lambda(i+1) - \lambda(i)}{\lambda(i+1) - \lambda(i)} = \boldsymbol{q}'(i)\dot{\lambda}(i) \tag{3.35}$$

where $\boldsymbol{q}'(i) = \frac{\boldsymbol{q}(i+1) - \boldsymbol{q}(i)}{\lambda(i+1) - \lambda(i)}$ is the partial derivative of q with respect to λ.

The same considerations could be done for $\ddot{\boldsymbol{q}}(i)$:

$$\begin{aligned}
\ddot{\boldsymbol{q}}(i) =& \frac{\dot{\boldsymbol{q}}(i+1) - \dot{\boldsymbol{q}}(i)}{t(i+1) - t(i)} = \frac{\boldsymbol{q}'(i+1)\dot{\lambda}(i+1) - \boldsymbol{q}'(i)\dot{\lambda}(i)}{t(i+1) - t(i)} = \\
=& \frac{\boldsymbol{q}'(i+1)\dot{\lambda}(i+1) - \boldsymbol{q}'(i)\dot{\lambda}(i) + \boldsymbol{q}'(i+1)\dot{\lambda}(i) - \boldsymbol{q}'(i+1)\dot{\lambda}(i)}{t(i+1) - t(i)} = \\
=& \frac{\boldsymbol{q}'(i+1)(\dot{\lambda}(i+1) - \dot{\lambda}(i))}{t(i+1) - t(i)} + \frac{\dot{\lambda}(i)(\boldsymbol{q}'(i+1) - \boldsymbol{q}'(i))}{t(i+1) - t(i)} = \\
=& \boldsymbol{q}'(i+1)\ddot{\lambda}(i) + \boldsymbol{q}''(i)\dot{\lambda}(i)^2
\end{aligned} \tag{3.36}$$

Now the parametrized dynamics becomes:

$$\begin{aligned} \boldsymbol{a}(i) &= \boldsymbol{q}'(i+1)H(\boldsymbol{q}(\lambda(i)) \\ \boldsymbol{b}(i) &= \boldsymbol{q}''(i)H(\boldsymbol{q}(\lambda(i)) \end{aligned} \tag{3.37}$$

If we compute the velocity at the sample $i+1$:

$$\begin{aligned} \dot{\boldsymbol{q}}(i+1) &= \dot{\boldsymbol{q}}(i) + \Delta_t \ddot{\boldsymbol{q}}(i) \\ &= \dot{\boldsymbol{q}}(i) + \Delta_t(\boldsymbol{q}'(i+1)\ddot{\lambda}(i) + \boldsymbol{q}''(i)\dot{\lambda}^2(i)) = \\ &= \dot{\boldsymbol{q}}(i) + \Delta_t\Big(\boldsymbol{q}'(i+1)\frac{\dot{\lambda}((i+1)-\dot{\lambda}(i)}{\Delta_t} + \frac{\boldsymbol{q}'(i+1)-\boldsymbol{q}'(i)}{\lambda(i+1)-\lambda(i)}\frac{\dot{\lambda}(i+1)-\dot{\lambda}(i)}{\Delta_t}\dot{\lambda}(i))\Big) = \\ &= \dot{\boldsymbol{q}}(i) + \boldsymbol{q}'(i+1)\dot{\lambda}(i+1) - \boldsymbol{q}'(i+1)\dot{\lambda}(i) + \boldsymbol{q}'(i+1)\dot{\lambda}(i) - \boldsymbol{q}'(i)\dot{\lambda}(i) = \\ &= \dot{\boldsymbol{q}}(i) + \boldsymbol{q}'(i+1)\dot{\lambda}(i+1) - \boldsymbol{q}'(i+1)\dot{\lambda} + \boldsymbol{q}'(i+1)\dot{\lambda}(i) - \boldsymbol{q}'(i)\dot{\lambda}(i) = \\ &= \boldsymbol{q}'(i+1)\dot{\lambda}(i+1) \end{aligned} \tag{3.38}$$

which is coherent with the parametric velocity computed previously in (3.35). So, using this formulation, the expected errors are due to the numerical ones derived from the use of approximated expressions for derivatives and integrals and not from a wrong expression of the discretization.

Since the dynamic equations have parameters ($\boldsymbol{H}$ and $\boldsymbol{h}$) which depend non-linearly on the position and velocity of joints, the number of samples chosen for discretization changes the accuracy of the planning. The higher the number of samples is, more accurate the planning is because the dynamics of these intermediate waypoints is included in the planning.

This formulation will be used in the design of the planner.

3.3.5 Algorithm design

Let's describe the algorithm to find the optimal time trajectory using the DP approach.

The aim is to build a grid in which the optimal path is found through a research of the best sequence of cells with the lower cost, from the initial to the final one.

The algorithm could be divided in the following steps:

1. Given the path input parametrized in λ, discretize the state-space in N_i λ samples and in N_j $\dot{\lambda}$ samples, creating a grid with $N_i \times N_j$ cells.

2. Compute the parametrized dynamics variables $\boldsymbol{a}(i)$ and $\boldsymbol{b}(i)$, as (3.37), for each sample of λ and $\dot{\lambda}$ in domains. In order to compute the parametrized dynamics, the joint space matrix $\boldsymbol{H}(i)$ and the non-linear coefficients $\boldsymbol{h}(i,j)$

for each state are needed: it is possible to compute them from the analytic formulas of robot dynamics or using some inverse dynamics algorithm, as explained in Sect. 2.1.3.

3. Compute the *pseudo-acceleration* limits for each cell of the grid and the maximum velocity curve for each stage. Finally, build the sets A_i and B_i which satisfy the constraints.

4. Initialize the cost of the entire grid to infinite, but for the initial state, which is set to zero.

5. Starting from the initial state, analyze sequentially each stage at a time, in the following way: for each feasible state j at stage i, if the constraints are respected, compute the local cost for each state k at stage $i+1$ which belongs to A_{i+1} domain. A state at stage i is said feasible if it is belongs to C_i.

   For each next state, we have to store the information about the best cost and the correspondent predecessor. This passage deserves a detailed explanation. Let us take into account a generic feasible state k at stage $i+1$. At the end of iterations for the stage i, we will have analyzed the local costs between the cell in consideration and the cells at the stage i. Since we are using a forward procedure, we have to store the best cost and the correspondent predecessor among all the computed costs. So, at each iteration, if the new computed cost is lower than the previous one, it is saved and substitute the old one. The predecessor must be saved in order to build the final optimal path. When the stage i is concluded, we will have that, for each feasible state at stage $i+1$, a cost and a predecessor pointer are saved. If one state does not have a cost or a predecessor, it means that no states at previous stage can't reach it and it will not be taken in account at the next iterations. Instead, if the state is reached, it is added to domain C_{i+1}, such that at the next step it is taken into account.

6. Once all the stages are analyzed, it is possible to obtain the best path, starting from the last desired state and retracing the path given by all the predecessor pointers of each state.

Chapter 4

Implementation in ROS

One of the aims of this thesis is to implement a time optimal planner in ROS and MoveIt! architecture and integrate it in the robotic framework available in ALTEC.

The proposed integration does not consist of a basic insertion of a new module. The aim is to refactor the existing architecture to develop a new one capable to offer users the potentialities of the dynamic programming.

The idea is to develop an independent component, encapsulating the algorithm, and an abstract layer, representing the underlying generic structure in which the optimal solution is found. The specific applications are integrated through the creation of specialized components, which define the context and the semantic of the optimization problem. Currently, two specific components are present in the framework: the redundancy resolution module, refactored from the already existing one, and the time optimal trajectory generation modules, the purpose of the presented work.

The user-friendliness is a requirement of the new architecture: users must easily employ the algorithm exploiting the application interface offered by the aforementioned components.

Finally, the environment of simulation available in ALTEC is expanded: the setup of a 3D dynamic simulator is performed in order to support demonstration, testing and validation of the developed technologies.

4.1 ROS and MoveIt!

ROS, which means *Robot Operating System*, is defined as meta-operating system and provides a software platform to develop robotic applications [6]. It offers a collection of tools, such as hardware abstraction, low-level control, navigation, planning and others, all intended to simplify and to help the creation of programs to

control robots. It is open source, so users can freely contribute to its development and to analyze the source code in order to understand and to improve it.

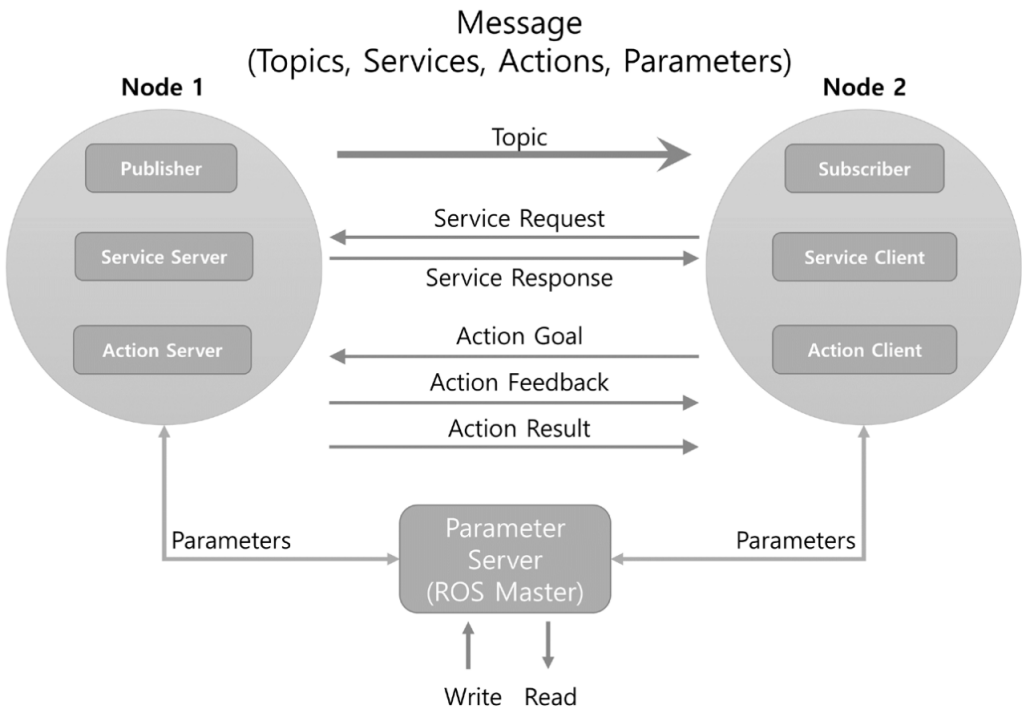

Figure 4.1 – Message communication scheme [46]

It is organized in basic units, called packages, which constitute the applications. Each package contains nodes, which run in dedicated processes. The packages are created in order to provide a specific functionality. Besides, they are created such a way that they can be reused to develop some user-defined functionalities.

The entire framework is based on communication: in fact, nodes can work together through the use of messages. There are three message communication methods [46, 6]:

- **Topic**: it is an unidirectional communication and is used to exchange data in a continuous way between a publisher node and a subscriber one. The former writes a message on a topic, while the latter receives the information published in the topic which it is connected to. When a subscriber receives a message, it usually performs an action, called *CallbackFunction*.

- **Service**: it is a bidirectional communication and is used to exchange request/response messages between a service server node and a service client node. The former offers a service under a string name, which can be called

by the latter using a request message. When the service is concluded, the server sends a response message to the client.

- **Action**: it is a bidirectional communication and is used to exchange goal/result /feedback messages between an action server node and an action client node. The former offers an action, which can be called by the latter through a goal message. During the action, the server sends periodically feedbacks to the client. When the action is finished, the server sends a result message to the client.

The message communication is described in Figure 4.1.

In order to store some data and retrieve them at runtime, nodes can use the *parameter server*. It is a centralized server which stores data identified in a key-value store with keys of type strings [41]. It is often used to store some configuration parameters which do not change at runtime.

MoveIt! is the motion planning framework of ROS and collects libraries to mainly control manipulators [7]. It provides several functions for motion planning, manipulation, navigation, control and others. Besides, it allows the visualization and the simulation on RViz [8] and Gazebo [2] platforms. The former is a 3D visualizer able to display the robot model, the sensor data or robot state information from ROS; the latter is a 3D dynamic simulator and will be described in detail in Sec. 4.2.

The main node provided by MoveIt! is the MoveGroup node which interacts with the user through ROS services and messages. It is a ROS node and can be configured by using the parameter server, from which receives some important information, such as the structure of robot, the dynamic and kinematic limits or the type of planner which is desired by the user. It can communicate with the robot through ROS actions and topics, in order to get some information, such as the current state or actual positions of joints from sensors.

The MoveGroup node offers several features, such as planning, inverse kinematics, execution of trajectories, through *capabilities*. They are basic functionalities of the MoveGroup node and are also extensible through a plugin architecture, by using the *pluginlib* library of ROS [9]. Generally, the MoveIt! framework is based on the plugin architecture because it allows to avoid recompiling without core modifications as well [7].

MoveIt! basically works with position kinematic planning: it plans only joint or end effector positions, without taking into account the velocity and acceleration values. There are several planners already available in MoveIt!, such as OMPL [10] and STOMP [11], which are tipically used in a point-to-point planning.

In order to parameterize the trajectory, adding the planning of velocity, acceleration and timestamps, the framework offers a post-processing of the path. In fact, if a trajectory with some constraints is desired, the *Motion Plan Request* is used: it essentially specifies what the motion planner has to do. In order to provide additional motion planning functionalities, MoveIt! allows the user to configure the motion planning capability via the definition of a pipeline, called *planning pipeline* which includes a planner and optionally a list of *Plan Request Adapters*, which precede or follow the default planner. These adapters aim to make planner more robust and to add some features.

MoveIt! already provides some tools to execute the time parametrization of a path. There are three available algorithms implemented in MoveIt!:

- Iterative Parabolic Time Parameterization
- Iterative Spline Parameterization
- Time-optimal Trajectory Generation

The last algorithm produces a time optimal trajectory, as in the aim of this thesis.

4.1.1 Time optimal trajectory generation in MoveIt!

MoveIt! already provides an algorithm which generates a time-optimal trajectory. A brief description of the algorithm is given.

Introduction

It is developed by Humanoid Robotics Lab of Georgia Institute of Technology. [39] explains their work: an integration method is used to find the switching points with a pre-processing of the trajectory in order to make it differentiable in every point. In fact, in a non-differentiable trajectory, at a waypoint, the direction changes immediately, forcing the robot to stop. The result is that the robot movement results slow and unnatural. To avoid this behaviour, they process the trajectory, adding circular blends in correspondence of waypoints, as showed in Fig. 4.2. In this way, the trajectory is continuous and differentiable and the robot has a smooth movement. It is worth to underline that the points where the path switches between a circular segment and linear ones represent discontinuous switching points. A detailed explanation of the algorithm could be found at [39].

They assume only acceleration and velocity joint limits, resulting in a simpler problem than the derived one from the torque limits. In fact, if only these bounds are assumed, the robot dynamic equations are not taken into account during the planning. So, the proposed algorithm is limited and not suitable to our problem formulation. For this reason, it is not used as term of comparison of our results.

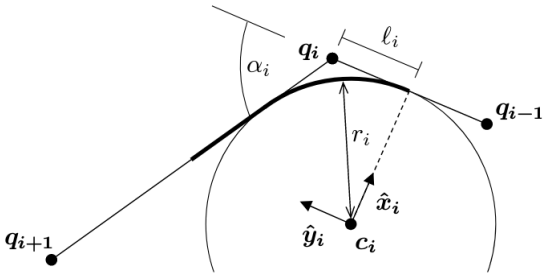

Figure 4.2 – Circular blends at waypoint $\mathbf{q_i}$. [39]

Architecture

The software architecture is composed mainly by three components:

- **Path**: it performs the pre-processing, adding the circular blends. So, the path is actually composed by a sequence of linear and circular segments and the conjunction points are saved as switching points candidates.
- **Trajectory**: it is responsible for executing the integration algorithm. In fact, it receives as input the pre-processed path and performs the forward and backward integration, the search for the switching points, the computation of the pseudo-accelerations and others.
- **TimeOptimalTrajectoryGeneration**: it represents the interface with the external architecture. It is responsible for creating the needed structures for generating the optimal trajectory and it controls and commands the entire process, from the pre-processing to the creation of the output variable.

  If the generation is successful, it clears the input trajectory and fills it again using the values obtained by the algorithm, adding positions, velocities and acceleration for each sample.

This trajectory processing could be added to the motion planning as a *Planning Request Adapter*, which is called `AddTimeOptimalParameterization`.

4.2 Gazebo and ROS control

Gazebo is a 3D dynamics simulator and provides models, worlds and tools to test and simulate robots in a realistic way through a physics engine [46, 2]. Several physics engines are available, depending on the user requirements: ODE, Symbody, DART and Bullet. It is characterized by a 3D graphics environment, which provides high quality simulations in indoor and outdoor simulations.

Gazebo is completely compatible with ROS. To obtain a complete integration, the set of packages *gazebo_ros_pkgs* are added to the framework. They provide the interface to communicate between ROS and Gazebo, through topics and services, and provide other functionalities. For example, in order to spawn the model in the simulator, it translates the URDF (Unified Robot Description Format) [12], a configuration file to describe robots adopted by ROS, to the SDF (Simulation Description Format) [13], the configuration file used by Gazebo.

In order to a simulate a robot in Gazebo using ROS, it is necessary to add two interfaces: an *hardware interface*, which represents the bridge with the simulated or real robot, and a *controller*, which is responsible for communicating with the hardware and sends the commands. Let us describe in detail the two components, defined both in *ros_control* packages [14].

In order to communicate with a real robot, ROS uses an Hardware Abstraction Layer, represented by the `RobotHW` interface, which has the aim to send commands to the real embedded controller and to read the state from the encoders on the robot [26]. This interface should be inherited for specific robot implementations. Besides, the `RobotHW` configures the specific hardware interfaces for joint, defining how the joint is commanded. The most common default hardware interfaces are:

- **EffortJointInterface**: used for commanding effort-based joints.
- **PositionJointInterface**: used for commanding position-based joints.
- **VelocityJointInterface**: used for commanding velocity-based joints.

If the robot is simulated, the Hardware Abstract Layer is defined by `RobotHWSim`, representing the simulated `RobotHW`. It has the same tasks: in fact, it provides functions to read states and command joints in Gazebo simulator.

The communication between the hardware interfaces and the ROS interface is handled by the controllers. Controllers in ROS have a more extended meaning than the classical one: they do not necessarily implement a feedback loop or act to minimize the error between the reference and the actual values. In general, their task is to receive a desired reference and to send the command to the hardware interface. This process can be done either in a forward way or using a feedback loop, receiving the actual joint state from the hardware interface. The controllers are characterized by two features: the type of command, which has to correspond to the hardware interface type, and the type of reference. If the reference type is equal to the command one, the signal is forwardly sent to the joint; otherwise, a feedback loop is implemented. For example, the **effort_controllers** collect the controllers which send effort commands to an EffortJointInterface. They

can accept several types of reference, defining different types of controller; some examples are:

- **effort_controllers::JointEffortController**: it forwards the command reference to joints.
- **effort_controller:JointPositionController**: it receives as input the desired position and implements a PID loop to compute the effort.
- **effort_controllers::JointTrajectoryController**: it adds extra functionalities for splining an entire trajectory as reference. It will be further analyzed later.

In general, the namespace indicates the type of command, while the name represents the type of reference used. A particular type of controller is the `joint_state_controller`: it is has the task to read and publish the joint state.

Since ROS does not support a real-time communication natively, ROS control adds also the *realtime_tools* package, which contains a set of tools that could be used from a hard realtime thread, necessary to control real robots.

The entire system is shown in Figure 4.3.

4.3 Software architecture

The aim of the planner is to compute the time-optimal trajectory as follows: it receives a task space or joint space path input, composed by a sequence of intermediate poses or joints configurations respectively, and computes the best sequence of *pseudo-velocities* using a search algorithm based on dynamic programming, as explained in Sect. 3.3. The best *pseudo-velocities* return the minimum time. Then, in order to simulate the robot behaviour, the optimal solution is translated and sent to controllers as a series of optimal efforts or as an optimal trajectory, composed by the vector of optimal joints positions, velocities and accelerations at each time instant. The entire planner is integrated in ROS and MoveIt! framework and uses Gazebo as simulator.

The developing planner has to be completely integrated in the framework available in ALTEC [29]. The existing software architecture is dedicated to the redundancy resolution, using dynamic programming [33].

Our starting point consists of extending the existing architecture in order to generalize the grids for dynamic programming and to create a module to integrate the algorithm for time-optimal planning.

First, the existing redundancy resolution module is introduced, summarizing its characteristics, in particular the common features with the time optimal planning

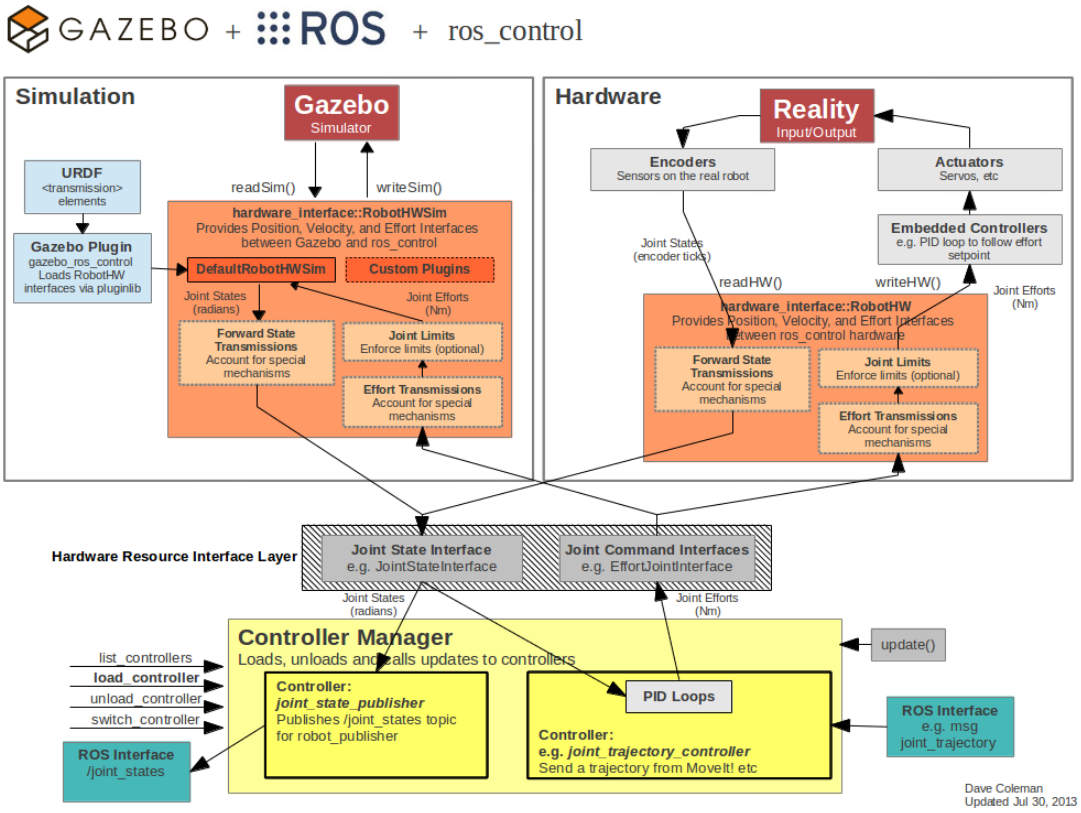

Figure 4.3 – Scheme representing the architecture overview on the simulated robot in Gazebo, the real robot and the controllers in ROS [2]

algorithm, then the final software architecture is described.

4.3.1 Redundancy resolution module

The redundancy resolution module has the task to find the globally-optimal joint-space solution of a redundant robot along a a prescribed path using a search algorithm based on dynamic programming, as described in Sec. 3.2. The aim is to extend the MoveGroup capabilities with a planner able to optimize a given performance index in an integral way, thus achieving globally-optimal inverse kinematics. The module has been already integrated in the framework and tested on a variety of robots, including the Panda [1], the Mico6 [15] and mobile manipulators (arms mounted on mobile bases for space applications).

Architecture

It is mainly composed by three parts:

- **Generation of grids**: The `StateSpaceGrid` is the class which represents all the discretized joint space solutions along the trajectory. Each cell of the grid, in fact, contains information about the joint vector solution of the IK. The code supports also the generation of a set of grids, in order to take into account the multiple solutions of IK and to generate a global solution.

- **Execution of algorithm:** The `DynamicProgrammingSolver` class is responsible for finding the optimal solution. When the solver is called, it computes all the costs in the grids and it stores, for each cell, the best cost and the best predecessor. The cost function can be defined by the user through the *ObjectiveFunction* class. At the end of the process, it is able to compute the minimum path, starting from the last node and going backward until the first one.

- **The MoveGroup capability:** The `MoveGroupDPRedundancyResolutionService` is the additional MoveGroup capability. It can be called through the `ros::ServiceServer` and it performs the entire planning process. It recovers the inputs from the parameter server and the request message, such as the robot type, the trajectory and the objective function and, depending on the type of robot, it creates an appropriate number of grids, computing the inverse kinematics solutions. Finally, it calls the dynamic programming solver function, which computes the optimal trajectory and sends it to Rviz visualizer.

4.3.2 Architecture design

Since the underlying resolution technique of redundancy resolution module is the same of the time optimal trajectory planning one and there are commonalities between the state space representation, the software architecture can be generalized to build a complete independent interface which could be used by the dynamic programming solver, moving all the specific characteristics of the semantics into specialized components.

The system are both discretized in order to create a state space, such that the dynamic programming can be exploited as a search algorithm in a grid. Although the states, the stages and the control inputs are represented by different quantities, the decision process underlying the transitions is not modified. For both, it is possible to define the set of constraints for the control input and for the states, limiting the feasible transitions. In order for the solver to find the best solution, each cell, or node, of the grid must store the cost and predecessor cell. Only the local decision depends on the semantics and the chosen cost function. Once

defined the cost and the predecessor, the semantics does not longer matter because the solver can build the final sequence of best nodes through the pointers of predecessors.

The cell represents the most critical component of the architecture: it is the bridge between the abstract layer and the semantics. It has to contain both the generic information for the solver, such as the cost and predecessor, and the specific data, i.e. the actual state and other, for checking the constraints and for computing the local cost. Nodes are given the autonomy to check the constraints and to compute the cost between their neighbours, but they are not able to *self-compute*, i.e. computing the semantics-specific state. They needs an external component, called manager, which knows the problem definition: it handles all the operations related to the problem, but not strictly to the optimization algorithm. These are the contour operations which could be required to accomplish the entire process. One example could be the generation of the robot trajectory variable, starting from the output of the optimization process.

In order for the manager to work correctly, it needs to be contextualized and to have access some robot and semantics specific information: a component, representing the context, is created for this purpose. It contains all the useful information needed by the manager and the other components of the architecture. It has to be created externally and allows to maintain the user-friendliness of the software, because it encloses all the needed information. Finally, it allows also to reduce the memory occupation: indeed, through the context, some information has not be replicated in the several components.

So, both the DP-inspired algorithms can be divided in the following common five phases:

1. **Construction of the grid**: It represents the structure of state space, on which the algorithm works and looks for the optimal solution. The grid is composed by a set of cells, representing the states of the system.

2. **Computation of cells**: Each cell has to contain the information relative to the state.

3. **Check of constraints**: In order for a transition to be feasible, the constraints must be satisfied.

4. **Computation of local costs**: If the constraints are satisfied, the local cost can be computed. The cost and the precessors are stored in this phase.

5. **Construction of the sequence of best cells**: Through the pointers of predecessors, it i possible to rebuild the sequence of states with minimum or maximum cost.

Once defined the main commonalities and differences, the software architecture can be described.

4.3.3 Context

As it is done in redundancy resolution module, the time-optimal trajectory planning is performed through an additional MoveGroup capability, called `MoveGroupDPTimeOptimalParametrizationService`. To execute the capability, a service interface is implemented: a ROS node calls a `ros::serviceClient` and exchanges a `GetTimeOptimalTrajectory` message with the `ros::serviceServer` executed in the MoveGroup node context. The former will send a request message, whereas the latter will send a response message at the end of the time parametrization process.

The capability accepts either a task-space path or a joint-space path, which are stored in the the `GetOptimalTimeTrajectoryRequest` message. If a task-space is given, the capability has also the task to perform the inverse kinematics in order to obtain the joints configurations corresponding to the input task poses. As explained in Sec. 3.3, this is necessary in order to compute the parametrized dynamics.

In order to simulate and visualize the optimal trajectory, the results are published through a `ros::Publisher` or a `ros::ServiceClient`, depending on the type of controller implemented, as will be explained later. Then, these controllers communicate with Gazebo in order to send the commands to the simulated joints. Finally, to analyze the results and plots, the solution can be exported to *bag* files through the *rosbag* package. These files can be then interpreted by MATLAB software [16].

The context is explained in the Fig. 4.4, through the UML component diagram.

4.3.4 Grid generalization and dynamic programming solver

The redundancy resolution module is taken as starting point to create a component which implements the dynamic programming without any connection to semantics and the context in which it is used.

The first step is to analyze and identify all the functions of the `StateSpaceNode`, `StateSpaceGrid` and`DynamicProgrammingSolver` classes that could be suitable to perform the operations described in Sec. 4.3.2. These functions are refactored in order to extract the semantics-specific features, which will be collected in the specialized components, and to adapt the generic features to realize the abstract layer. Finally, the novel components, i.e. the manager and the context, as

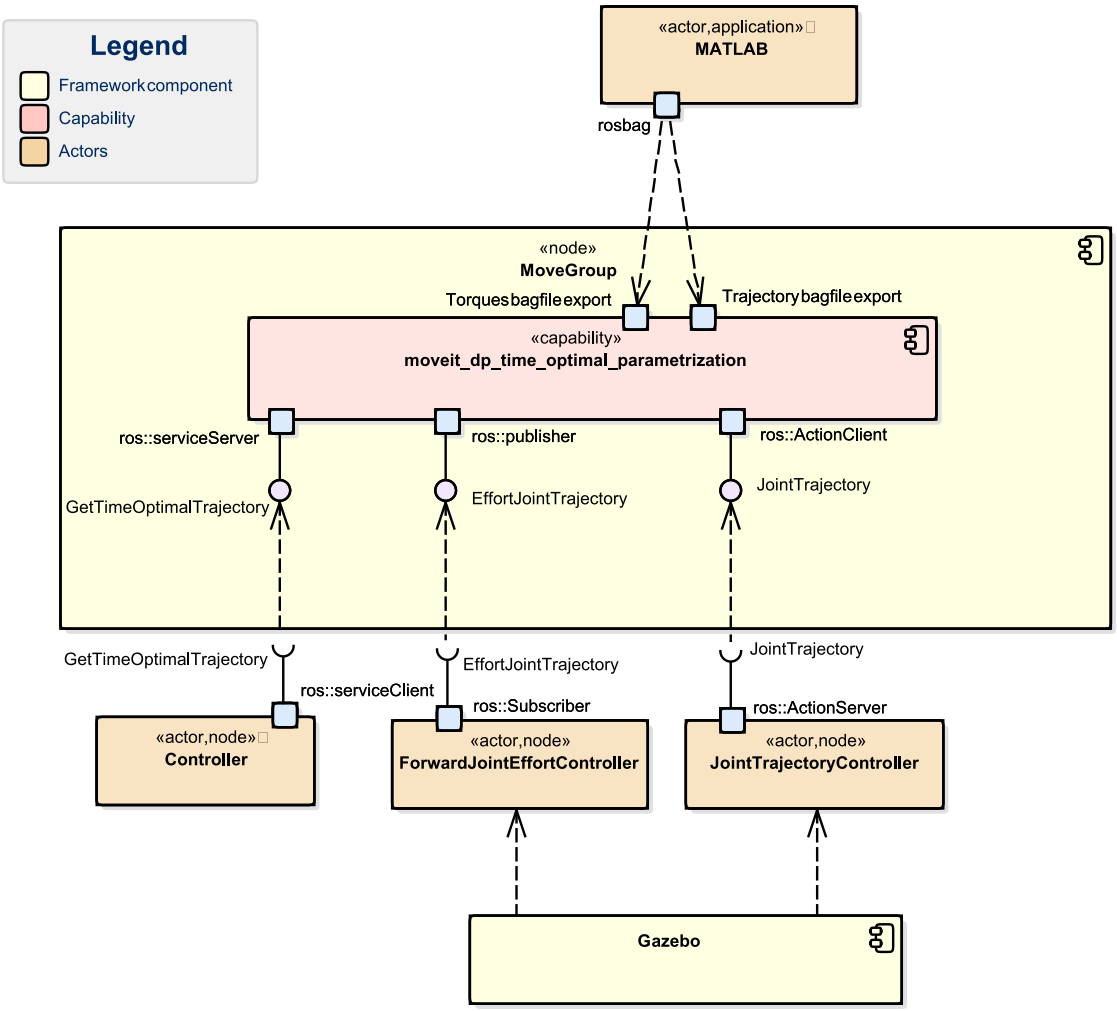

Figure 4.4 – UML component diagram representing the connections between the capability and the other components. The link with Gazebo is better explained in Fig. 4.3.

described in Sec. 4.3.2, are created. So, the new architecture will be composed as follows:

1. The components which realize the dynamic programming;
2. The abstract layer which interacts with the dynamic programming components. It has to be specialized to define the specific semantics.

It is worth to underline that the dynamic programming components communicates only with the abstract layer and never know the problem definition.

Let us describe the main components of the architecture:

- **StateSpaceNode**: it is the smallest unit of the architecture and it represents one grid cell, or, equivalently, one state of the system. It stores basic

information such as the cost and the predecessor, useful to build the final sequence of best nodes.

- **StateSpaceGrid**: it is the entity representing the grid and contains the collections of `StateSpaceNode`.
- **DynamicProgrammingSolver**: it is the entity responsible for executing the search algorithm along the set of `StateSpaceNode` of the grid and for building the sequence of best states.
- **StateSpaceManager**: it is the entity which manages the grids, such that they can be generated and computed, and handles some utility functions, such as the export to bags or the creation of trajectory variable.
- **StateSpaceContext**: it is a data structure which contains all the useful information to perform the algorithm.

The `StateSpaceNode`, `StateSpaceManager` and `StateSpaceContext` are the entities of the abstract layer and must be specialized to apply the dynamic programming to different problems, whereas the `StateSpaceGrid` and `DynamicProgrammingSolver` are completely independent from the semantic, since they represent the components to realize the search algorithm.

The specialized components have to define all the semantics-specific functions and attributes, which are necessary for the execution of the algorithm. First, the `StateSpaceNode` must be able to check the fulfilment of the constraints and to compute the local cost of the transition towards an adjacent node. So, it has to own all the needed information to execute these operations. The nodes are not able to compute autonomously these information: this is an high-level operation which would require too effort and too much knowledge for a single node. The *computation of the node* is the main task of the `StateSpaceManager`, which has the capability to access to all the needed information. So, it has to know perfectly the problem definition and how to compute the data of nodes.

The architecture of the entire module can be analyzed on the UML decomposition diagram in Fig. 4.5.

4.3.5 Implementation of time optimal trajectory generation module

The module for optimal trajectory generations needs to inherit the abstract classes previously defined.

The `StateSpaceNodeTimeOptimalTrajectory` is the class which represents the state of the problem, composed by the pair $(\lambda, \dot{\lambda})$. Besides, in order to check

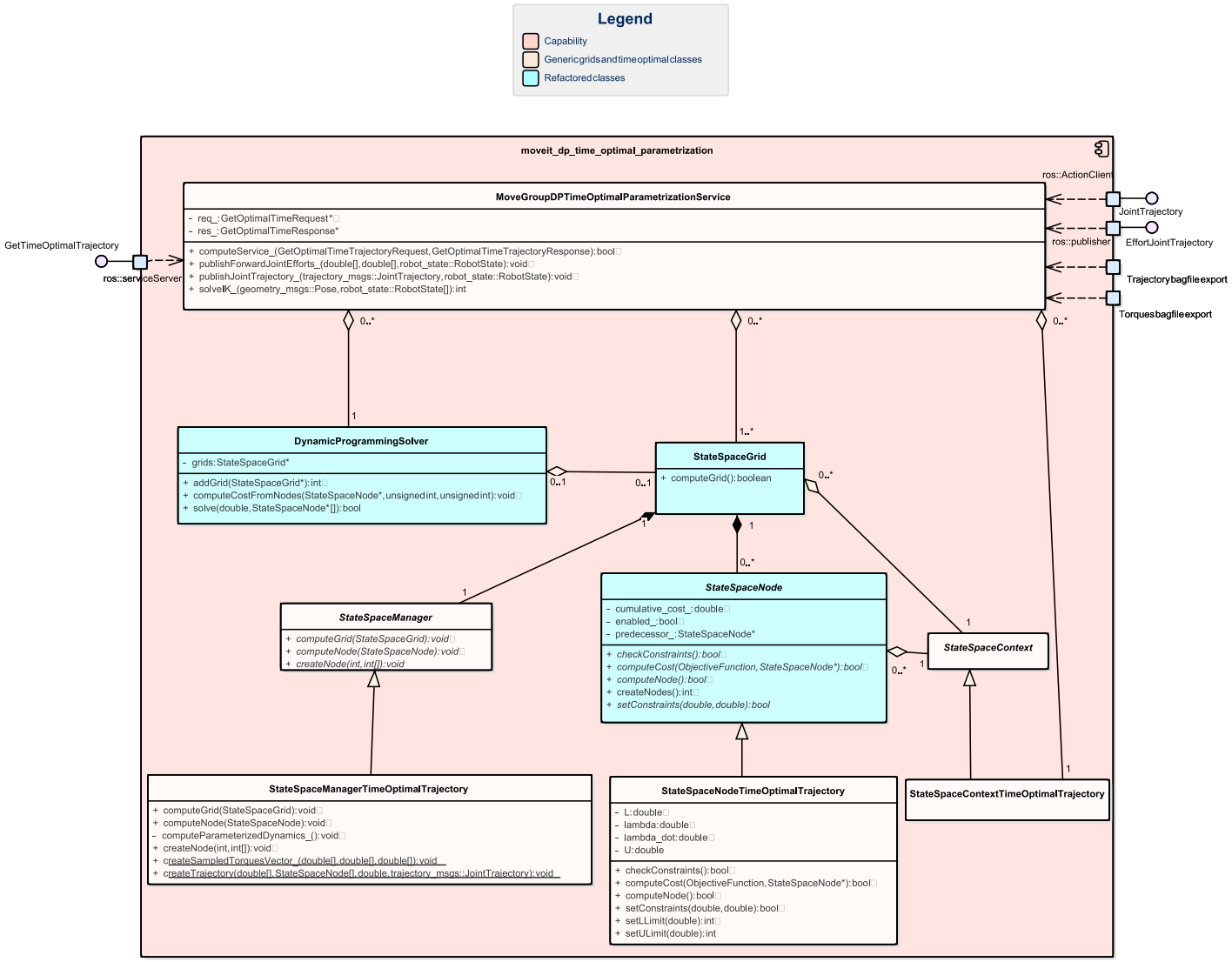

Figure 4.5 – UML decomposition/class diagram representing the classes which compose the time optimal trajectory generation module.

constraints during transitions, it contains the information about the *pseudo-acceleration* limits, or rather the couple (L, U).

In order to compute the data of `StateSpaceNodeTimeOptimalTrajectory`, the `StateSpaceManagerTimeOptimalTrajectory` has to compute the parametrized dynamics. So it has access to all the robot characteristics, through the *kdl_parser* library which allows to parse the URDF [17]. Then, it is able to compute the parameters of dynamics equation for each state of system, again through the KDL library [4]. Once the parametrized dynamics is obtained, the limits of *pseudo-accelerations* of the entire grid can be computed from Eqs. 2.39, 2.40.

Now the grid is ready and the `DynamicProgrammingSolver` can be called. As described in Sec. 3.3, it is assumed that the initial state is the rest condition. So,

the DP solver starts from the initial state, the only one which is actually active and belonging to C_0, and, in an iterative way, it begins to compute the local costs for each couple of states at successive stages. At each iteration, it checks if the next stage and the transition are feasible and, if successful, it computes the cost and compares it with the previous one. If it is lower, it saves the new cost and the new predecessor. When all the stages are analyzed, it builds the sequence of best `StateSpaceNodeTimeOptimalTrajectory` instances.

The sequence of nodes represents the best trajectory in $(\lambda, \dot{\lambda})$ plane. It can be translated into a sequence of efforts through Eq. (2.27) or into a sequence of joint positions, velocities and accelerations through (3.35) and (3.36).

The software behaviour is illustrated in the UML sequence diagram in Fig. 4.6.

4.3.6 Trajectory control

As explained in Sec. 4.2, in order to simulate a robot in Gazebo, the hardware interface and the controllers must be defined. First, we assume that the joints are torque-driven, so the EffortJointInterface is chosen. As far as concerns the controllers, we use two types: the `JointTrajectoryController`, which is a default one and it is often used in combination with MoveIt!, and the `ForwardJointEffortController`, a custom controller, created to forward directly sampled efforts to joints.

Let us describe these controllers and how it is possible to communicate with them.

Joint Trajectory Controller

It is a default controller provided by ROS control [18], whose aim is to execute joint-space trajectories on a group of joints. The trajectories are specified as a sequence of waypoints, which consist of positions and possibly of velocities and accelerations at each time instant. Depending on the type of parameters provided, it executes respectively a linear, cubic or quintic interpolation between given points. The desired trajectory is specified in the `trajectory_msgs::JointTrajectory`.

The controller can work with different hardware types: effort, position and velocity - controlled joints. If the joints are actuated in position, or equivalently they have a `PositionJointInterface`, the desired positions from trajectory are forwarded directly to the joints. While if the joints are actuated in velocity or in effort, a PID feedback control is implemented. Since we assume an `EffortJointInterface`, a PID is added to the control loop.

In order to communicate with the controller, there are two available mechanisms: the *topic interface* or *action interface*. We use the *action interface*, because it

allows to monitor the execution. For example, it is possible to know if the goal is reached, under some pre-defined tolerances.

So, the capability `MoveGroupTimeOptimalDPTrajectoryService` has the task to send the filled `trajectory_msgs::JointTrajectory` message to the controller. When the `DynamicProgrammingSolver` has terminated its work, the `StateSpaceManagerTimeOptimalTrajectory` translates the sequence of best nodes into a sequence of joints positions, velocities and acceleration with a defined time instant. Each waypoint is saved into a `JointTrajectoryPoint` variable, which fills the `trajectory_msgs::JointTrajectory` message. When the message is ready, the capability uses the `ActionClient` to send the trajectory message. The controller receives it and executes the control action on the robot.

Forward Joint Effort Controller

It is a custom controller created for this specific application. In fact, when the robot model is perfectly known, it is expected that the torques directly computed from the planning and dynamics equation, are sufficient to move the robot as desired. Obviously, the actual path performed by the robot will be not perfectly coincident with the desired one because of the numerical and discretization errors, but it is possible to demonstrate that they can be very similar choosing an high number of samples.

Starting from these considerations, we have created a custom controller, whose task is to receive the optimal efforts vector and to forward it to joints. In order to make move robot as desired, it is necessary to send efforts at the specific time instant required by the planning. Since ROS is not a real-time architecture, the classical `JointEffortController` provided by *ROS control* is not adequate for our application. Indeed, it receives one message at a time, containing the efforts to be sent to joints, but it is not ensured that the published messages are received and executed at the same moment. So, to avoid this problem, we have created a controller which receives as input a vector of efforts for the entire trajectory. It stores them in a buffer and at each cycle update, it sends a sequence of efforts to joints. Besides, since the frequency loop is fixed, the efforts are sampled with the same frequency of the controller with a kind of zero-order hold model. In this way, the probability that the effort is not sent at the right time instant is decreased a lot and depends only on the execution of the controller in time. The higher the frequency is, the smaller the error due to this problem is.

The `ForwardJointEffortController` receives an `EffortJointTrajectory` message, which contains the vector of sampled joint efforts.

The capability `MoveGroupTimeOptimalDPTrajectoryService` has the task to

send the filled `EffortJointTrajectory` message to the controller. When the `DynamicProgrammingSolver` has terminated its work, the `StateSpaceManagerTimeOptimalTrajectory` translates the sequence of best nodes into a sequence of joints efforts. Then it re-sample them at the desired control loop frequency.

When the message is ready, the capability uses a `ros::Publisher` to send the efforts. The controller receives it and at each cycle update it sends the sequence of efforts to joints, which must have an `EffortJointInterface`.

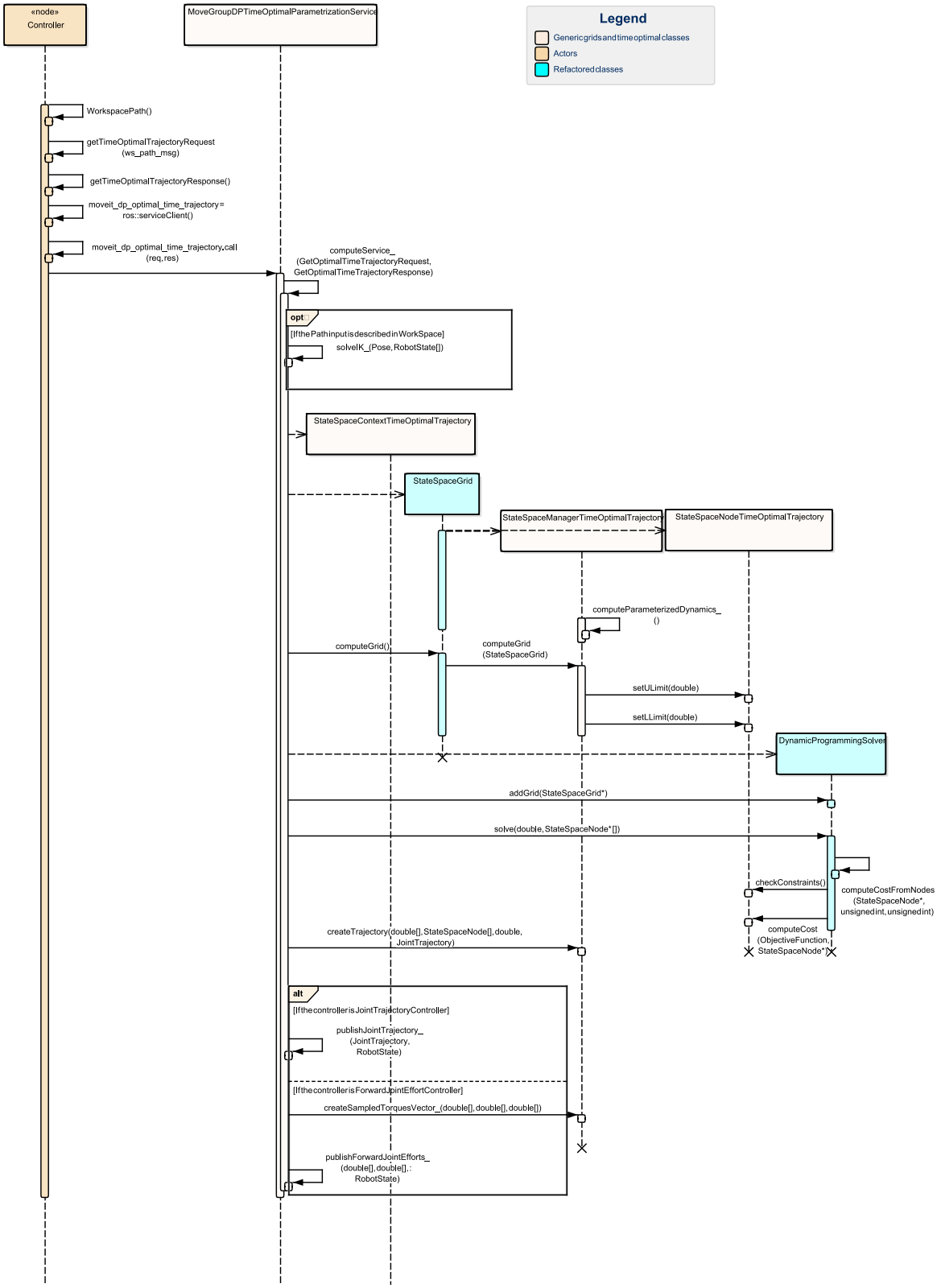

Figure 4.6 – UML sequence diagram illustrating the high level sequence to perform the planning.

Chapter 5

Simulation and results

In order to validate the algorithm, the generated references from the dynamic programming planner are tested through several simulations on Gazebo, using a two revolute joints planar manipulator, which is the same of the use case proposed by *Shiller* in [47]. It is also used to give proof of some expected behaviours derived from the algorithm design.

5.1 Application to a 2R planar robot

Shiller proposes the application of the switching points method on a 2R planar manipulator [47]. In order to validate the results of the DP-inspired algorithm, the same robot and desired path are used.

Shiller supposes to have monodimensional links, so only the lengths of links and the inertia around *z-axis* are defined. Unfortunately, this kind of description is not enough for a dynamic simulation in Gazebo, thus it is necessary to extend it to the three-dimensional case. It is modeled with 3D links and the inertia on all the principal axes and is showed in Fig. 5.1. The main mechanical characteristics of the robot are described in Table 5.1.

Dimensions $[m]$	Mass $[kg]$	Inertia $[kg \cdot m^2]$	Maximum torque $[N \cdot m]$
$l_1 = 1 \times 0.05 \times 0.01$	$m_1 = 1$	$I_{1,\{xx,yy,zz\}} = 0.08$ $I_{1,\{xy,xz,yz\}} = 0$	$\tau_1 = \pm 20$
$l_2 = 1 \times 0.05 \times 0.01$	$m_2 = 1$	$I_{2,\{xx,yy,zz\}} = 0.08$ $I_{2,\{xy,xz,uz\}} = 0$	$\tau_2 = \pm 10$

Table 5.1 – Mechanical characteristics of 2R planar manipulator

The trajectory is supposed to be the same as [47] and it is shown in Fig. 5.1.

Figure 5.1 – Shiller robot model and desired trajectory

The following aspects will be demonstrated:

1. The validity of the algorithm, comparing the obtained optimal trajectory in phase plane with the one obtained through the switching points method, described in [47] and available in the framework of ALTEC.

2. The errors caused by the discretization, as explained in Sect. 3.3, comparing the simulated joints positions and velocities with the ideal ones obtained from planning.

3. The effects on the solution quality and computational time when the grid resolution increases.

In order to show these aspects, various simulations with different configuration parameters are performed.

5.1.1 First simulation: comparison with the switching point method

The aim is to validate the algorithm, comparing the optimal trajectory in phase plane obtained using dynamic programming with the one obtained using the switching points method described by *Shiller* [47]. The results of the switching point method were available in the framework of ALTEC.

The following parameters are set for the planner:

- Number of samples for λ domain: $N_i = 150$
- Number of samples for $\dot{\lambda}$ domain: $N_u = 1000$

- 2^{nd} formulation, as explained in Par. 3.3.4.

It is worth to underline that the choice of the grid resolution, e.g. the number of samples of λ and $\dot{\lambda}$ domains, is a relevant aspect during the design of the planner because it affects both the quality of solution and the computational effort. This aspect will be analyzed in the third simulation, with some examples to give proof of this statement.

The algorithm is executed in ~11 s on an Intel Core i7-6700 CPU @ 3.40GHz x 8, without parallel execution. The optimal trajectory time obtained from the algorithm is equal to 0.8487 s.

In Fig. 5.2 both the optimal trajectories are reported. In addition, the MVC is showed too, in order to verify that the trajectories do not violate the constraints. In general, the trend of the two trajectories is very similar, but in the neighbourhood of the critical point. In fact, it is worth to underline that the dynamic programming solution behave better near to the critical point; in fact, as explained in Sec. 2.6.4 the integration method presents a torque jitter in the neighbourhood of the critical point. In dynamic programming optimal trajectory, this may not happen because the phase plane trajectory does not depend on the integration fields of the *pseudo-acceleration*.

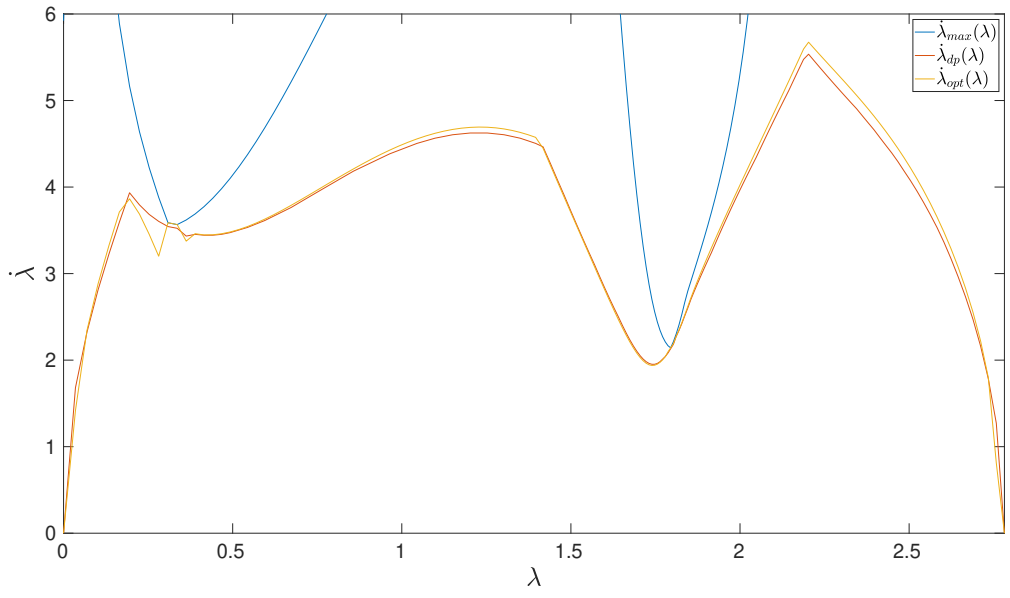

Figure 5.2 – First simulation. The optimal trajectories in phase plane: in red the trajectory obtained through dynamic programming, in yellow the trajectory obtained through switching points method and in blue the MVC.

In Fig. 5.3, the comparison between planned torques of DP solution and

switching point one are reported. The *bang-bang* profile of torques is proven: in fact, one actuator is always saturated in both cases. In DP torques, during saturation, the torques present some jitters. They are caused by the resolution of the grid, which affects the resolution of feasible *pseudo-accelerations*. The jitter could be decreased if an higher grid resolution is chosen. Besides, at 0.12 s s, it is possible to notice the torque jitter of the switching point method: it presents a very fast variation. As previously said, the same behaviour is not present in the DP-torques.

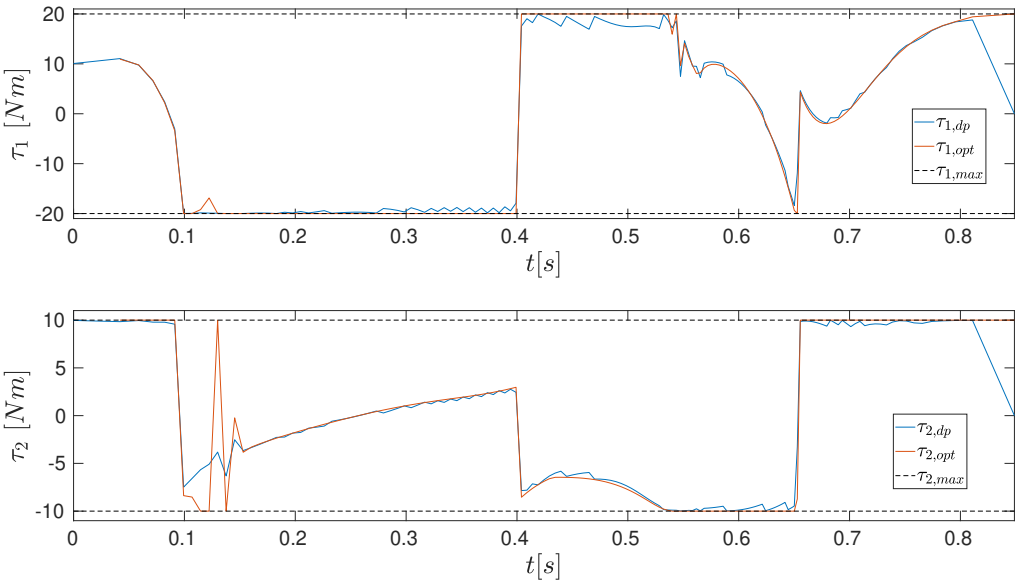

Figure 5.3 – First simulation. Comparison between planned torques obtained through dynamic programming (blue) and switching points method (red).

5.1.2 Second simulation: comparison between different discretization formulations

The objective is to compare the results between the different types of discretization explained in Sect. 3.3 and to show how the choice could affect the solution quality.

The following parameters are set for the planner:

- Number of samples for λ domain: $N_i = 300$
- Number of samples for $\dot{\lambda}$ domain: $N_u = 3000$

The main characteristics of the two formulations are defined as follows:

1. **First formulation**: The *pseudo-acceleration* is computed through the chain derivative rule and considering the approximation (3.28) when $\dot{\lambda}(i) = 0$, as explained in Par. 3.3.3. The joints' velocities and accelerations are computed using the first formulation explained in Par. 3.3.4.

2. **Second formulation**: The *pseudo-acceleration* is computed through the expression of the traveled time between two points, as in (3.29). The joints' velocities and accelerations are computed using the second formulation explained in Par. 3.3.4.

The algorithms for both formulations are executed in ~2.4 *min* on Intel Core i7-6700 CPU @ 3.40GHz x 8, without parallel execution. The optimal trajectory time obtained from the first formulation algorithm is equal to 0.8862 s, while it is equal to 0.8448 s for the second formulation.

In Fig. 5.4 the optimal trajectories in phase plane are reported with the MVC curve and the switching point trajectory.

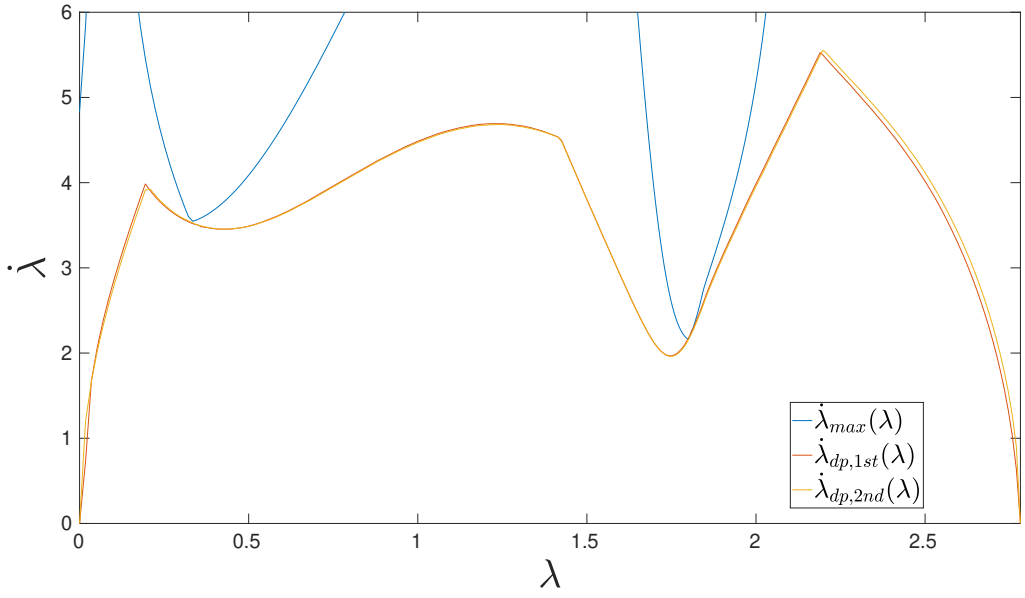

Figure 5.4 – Second simulation. The optimal trajectories in phase plane: in red the trajectory obtained through dynamic programming with 1st formulation, in yellow the trajectory with 2nd formulation and in blue the MVC, corresponding to $N_i = 300$.

In order to show the different results, the simulations on Gazebo are performed. The ForwardJointEffortController is chosen such that to analyze the robot behaviour in response to the planned torques. In fact, let us recall that the ForwardJointEffortController sends the planned torques in open loop.

In Fig. 5.5, the sampled planned torques are reported for both formulation. They are sampled with a frequency of 10 KHz. So, the controller sends joints commands every 0.1 ms. It is possible to see a temporal shift between the curves, which is due to the different discretization: recalling Eq. (2.27), the joint torque depends directly on $a(i)$ and $b(i)$ which are computed differently for the two formulation, from Eq. (3.32) for 1^{st} formulation and from Eq. (3.37) for the 2^{nd} formulation. Besides, the planning results are different too, again due to the chosen discretization.

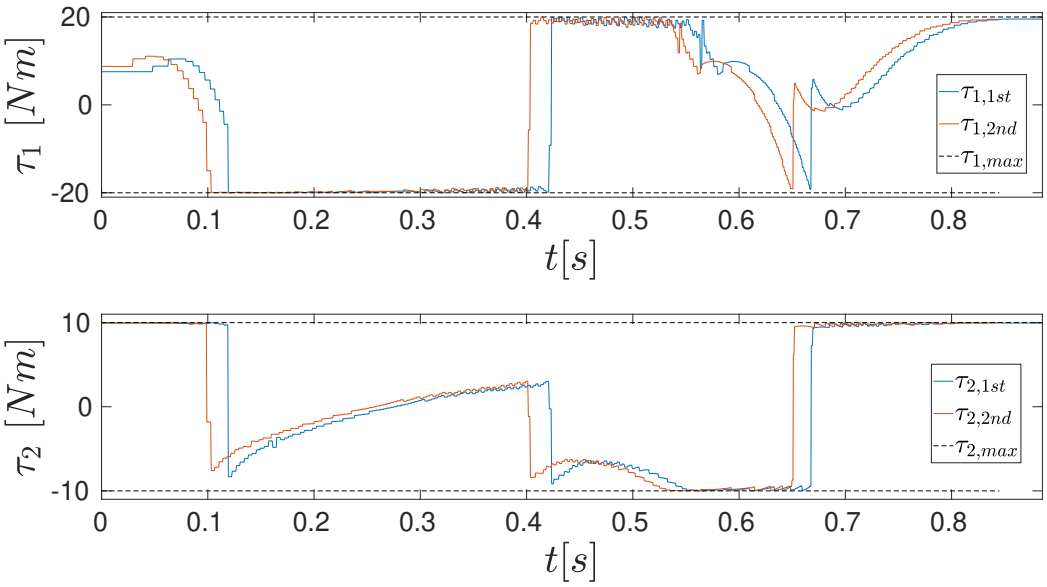

Figure 5.5 – Second simulation: The comparison between planned torques in 1^{st} formulation (blue) and 2^{nd} formulation (red).

In Fig. 5.6 and 5.7, the simulated joints' positions and velocities are showed. They are compared with the ideal joints' positions, derived from the IK, and velocities, from Eq. 3.31 for 1^{st} formulation and from Eq. 3.35 for 2^{nd} formulation. It is evident that, in first formulation, the model is ill-defined: in fact, the ideal and simulated curves are distant since the initial time instants. Whereas, in second formulation, the curves are coincident at the beginning and they detach along the trajectory due to the numerical errors. It is worth to underline that the sampled planned torques are able to move the robot as desired, unless small errors.

5.1.3 Third simulation: increase of grid resolution

The aim is to demonstrate that an higher grid resolution determines a better solution. As already said in the previous paragraphs, the grid resolution affects

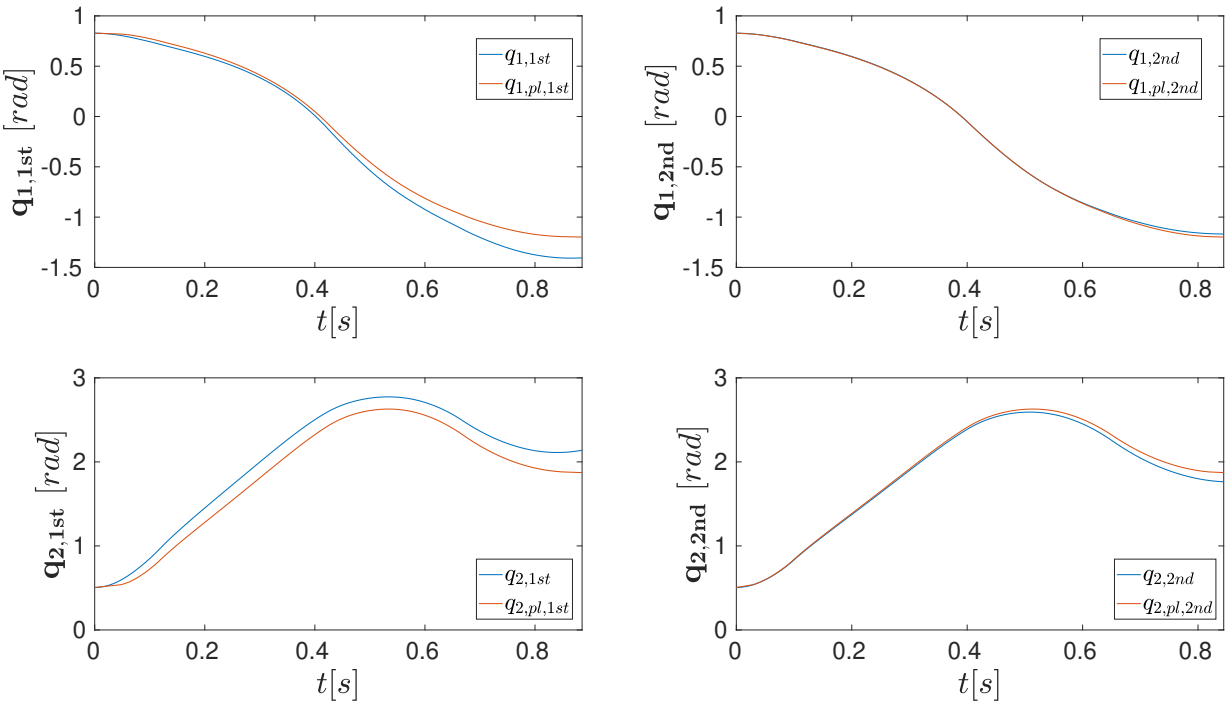

Figure 5.6 – Second simulation. The comparison between simulated joints' position (blue) and ideal ones (red), obtained from planning. On the left, the results from 1^{st} formulation, on the right from 2^{nd} formulation.

both the computational effort and the solution quality. A trade-off has to be found: the grid has to be dense enough to obtain more accurate results, but not too much because the computational time increases very fast. The number of columns, or N_i samples of λ domain, determines the accuracy of the dynamic model at waypoint, while the number of rows, or N_u samples of $\dot{\lambda}$ domain, affects the accuracy of cost computation [37]. Besides, also the ratio between N_i and N_u affects the quality: in fact, in the worst case, if N_u is very small and N_i is very big, it may happen that no *pseudo-acceleration* exists to reach the next stage [37]. In general, the ratio affects the resolution of the allowable *pseudo-accelerations*.

To prove these statements, two simulations are performed: one with a low resolution grid and one with an high resolution grid. For the first simulation, the following parameters are set:

- Number of samples for λ domain: $N_i = 150$
- Number of samples for $\dot{\lambda}$ domain: $N_u = 1000$
- 2^{nd} formulation.

while for the second one:

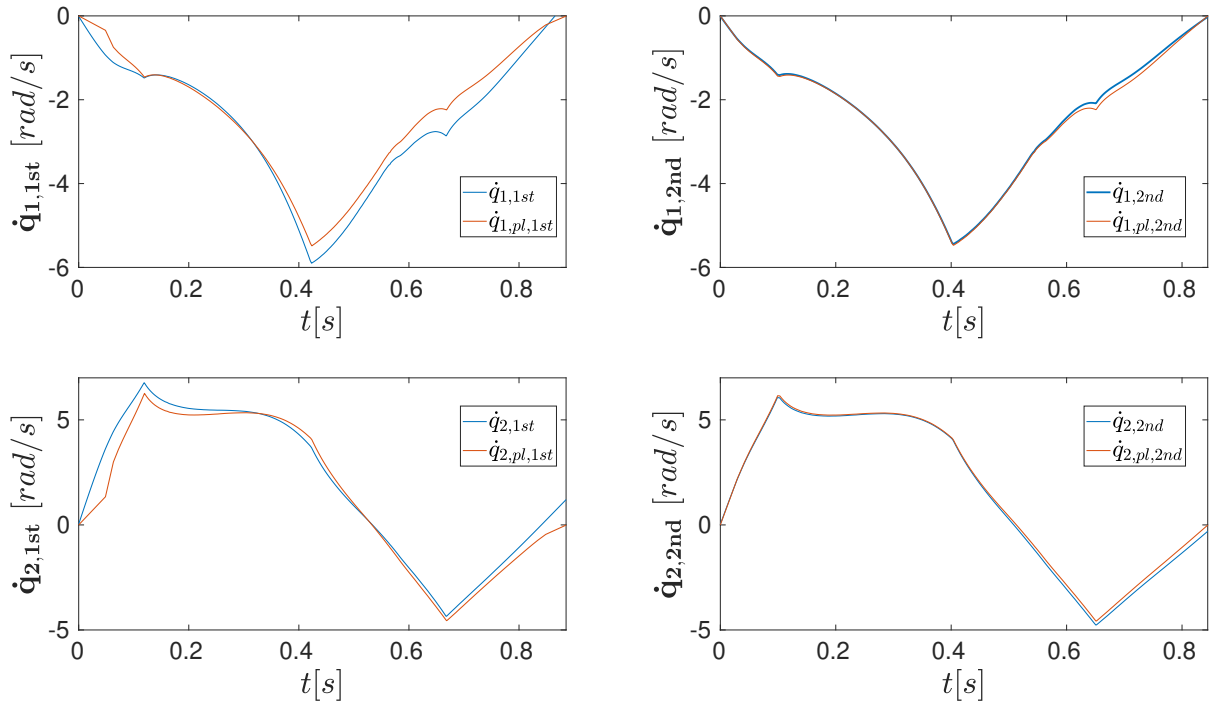

Figure 5.7 – Second simulation: The comparison between simulated joints' velocities (blue), and ideal ones (red) obtained from planning. On the left, the results from 1^{st} formulation, on the right from 2^{nd} formulation.

- Number of samples for λ domain: $N_i = 300$
- Number of samples for $\dot{\lambda}$ domain: $N_u = 10000$
- 2^{nd} formulation.

The algorithm with the first set of parameters is executed in ~11 s, while the second is executed in ~30 min on Intel Core i7-6700 CPU @ 3.40GHz x 8, without parallel execution. For the high resolution grid, the computational time has drastically increased. The optimal trajectory time obtained from the first simulation algorithm is equal to 0.8487 s, while it is equal to 0.8409 s for the second simulation. In the second simulation, the optimal cost is slightly decreased, so the solution quality is improved.

In Fig. 5.8 the optimal trajectories in phase plane are reported with the MVC curve. It is possible to notice that in the 2^{nd} simulation optimal phase plane trajectory, the red curve, is above the 1^{st} one, the yellow curve; this is equivalent to a faster trajectory. Besides, the red curve is nearer to the critical point than the blue one: this is due to the increased resolution of the *pseudo-accelerations*.

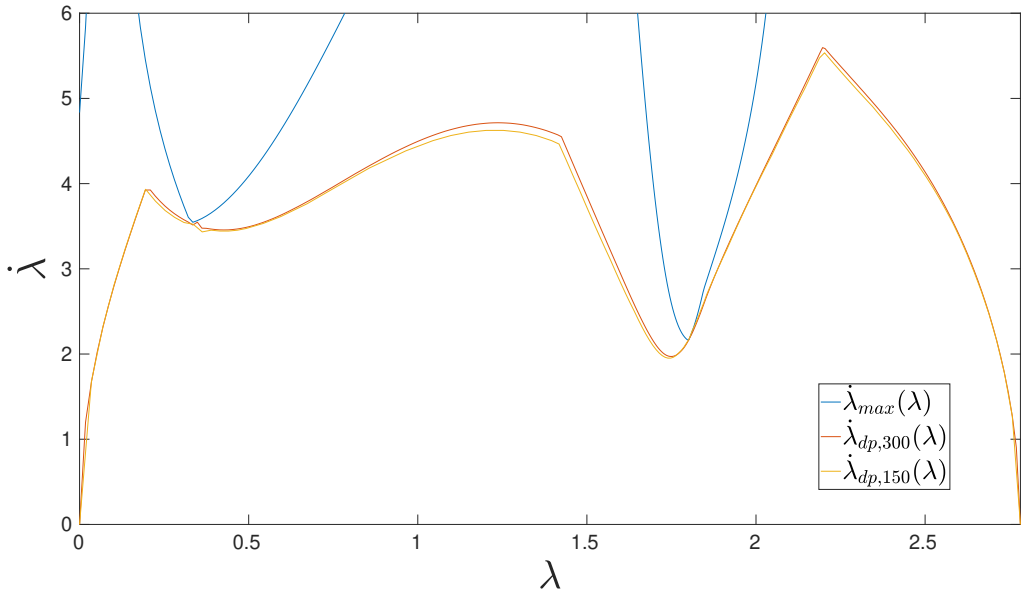

Figure 5.8 – Third simulation. The comparison between phase plane trajectories: in yellow the trajectory from 1^{st} simulation with low grid resolution, in red from 2^{nd} simulation with high grid resolution.

In order to show the different results, the simulations on Gazebo are performed. The ForwardJointEffortController is chosen such that to analyze the robot behaviours in response to the planned torques.

In Fig 5.9, the sampled planned torques are reported. They are sampled with a frequency of 10 KHz. So, the controller sends joints command every $0.1ms$. The trend is very similar, but it is possible to notice that in the 2^{nd} simulation, i.e. the blue curve, the torque jitter at saturation is decreased. Besides, in the neighbourhood of the critical point, the red curve presents a bigger torque jitter: this is due to the vicinity of the curve to the critical point, as happens in the switching point method and explained in Sec. 2.6.4.

In Fig. 5.10 and 5.11 the simulated joints' positions and velocities are showed, compared with the respective ideal joints' positions and velocities, computed during planning phase. In the second simulation the error between ideal and simulated curves for both positions and velocities is lower. This improvement is due to the decrease of numerical errors caused by the discretization. In particular, the higher the number of intermediate waypoints is, the more accurate the dynamic model is.

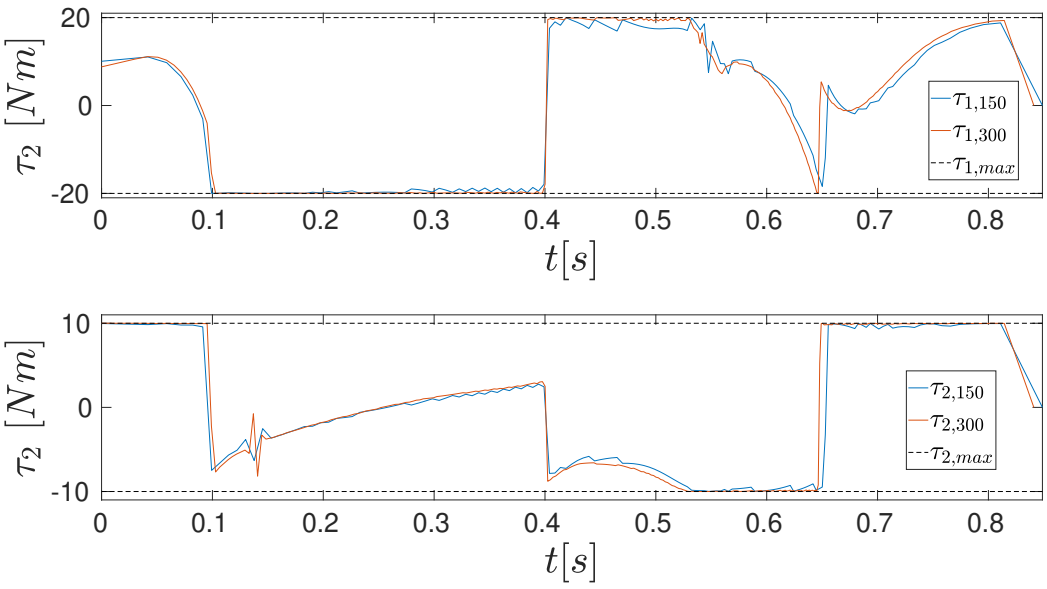

Figure 5.9 – Third simulation. The comparison between planned torques in 1^{st} simulation with low grid resolution (blue) and 2^{nd} simulation with high grid resolution (red).

5.1.4 Fourth simulation: JointTrajectoryController

The aim is to test the JointTrajectoryController and to analyze the results that could be obtained if a feedback controller is used.

Using a forward controller, such as the JointForwardEffortController, the error cannot be controlled. In fact, it is interesting to notice that, in previous simulations, the final joints' velocities are not equal to zero, so it would continue to move, even if the desired trajectory is concluded. So, the errors accumulated during planning will affect the simulation, also in the case the robot model is very simple and perfectly known; besides, in real robots, the models becomes much more complex and very difficult to obtain, so the errors drastically increase. If the model is perfectly known, it is expected that an infinite grid resolution, which corresponds to a continuous one, could solve the problem. Another solution is to add a feedback loop in the system: so the actual state of the robot is compared to the desired one and the torques are generated by controller to minimize this error.

The set of parameters for planner is:

- Number of samples for λ domain: $N_i = 150$
- Number of samples for $\dot{\lambda}$ domain: $N_u = 1000$
- 2^{nd} formulation.

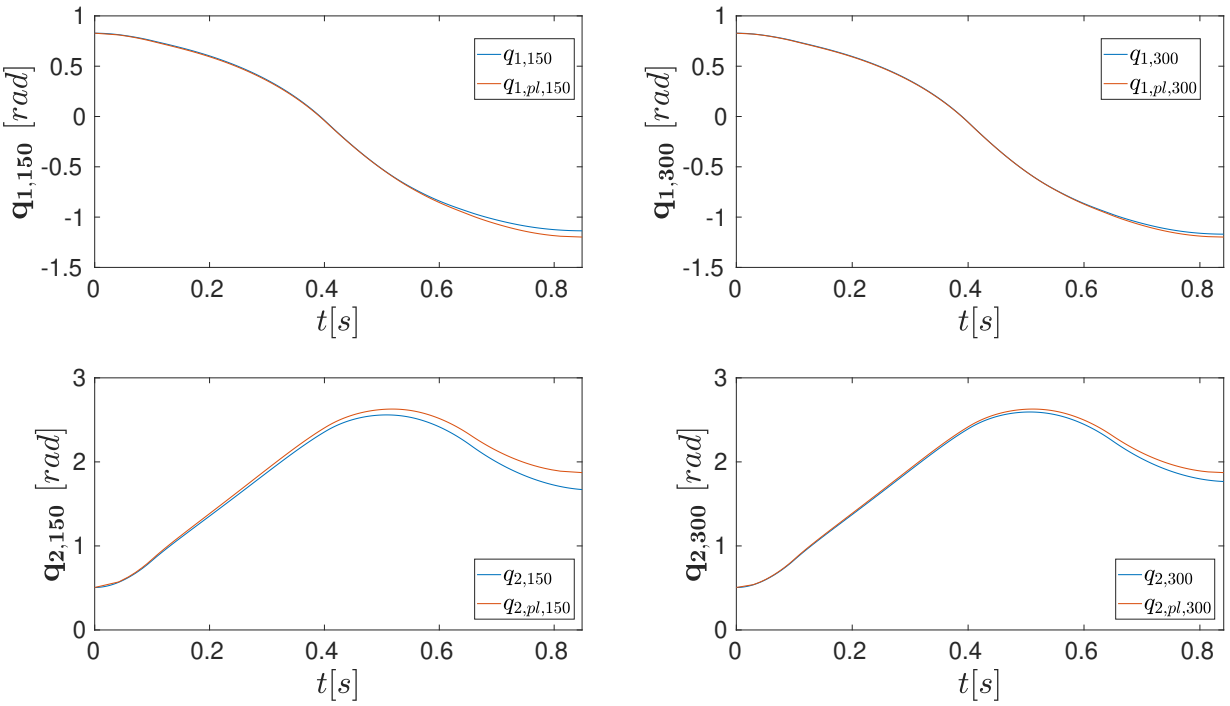

Figure 5.10 – Third simulation. The comparison between simulated joints' positions (blue) and ideal ones (red) obtained from planning. On the left, the results from 1^{st} simulation with low grid resolution, on the right from 2^{nd} simulation with high grid resolution.

- PID gains: $K_p = 200$, $K_d = 200$, $K_i = 50$

The computational time, trajectories in phase plane and optimal time solution are the same as the previous simulation because the chosen parameters are equal.

The JointTrajectoryController receives as input a message containing the positions, velocities and acceleration desired at each time instant. Once received the desired trajectory, it applies a quintic interpolation in order to guarantee the continuity at acceleration order and, finally, the interpolated values of positions and velocities are used as references during the simulation. It is interesting to notice how the planning velocities and accelerations are modified because of the interpolation process in Fig. 5.12. The acceleration are so oscillating in order to guarantee the continuity at all the three levels, e.g. positions, velocities and accelerations, while the velocities are affected by minor oscillations. The positions are not really altered from the interpolation; for this reason, they are not reported. These effects are caused by the chosen approximation to discretize the system, i.e. the forward finite differences.

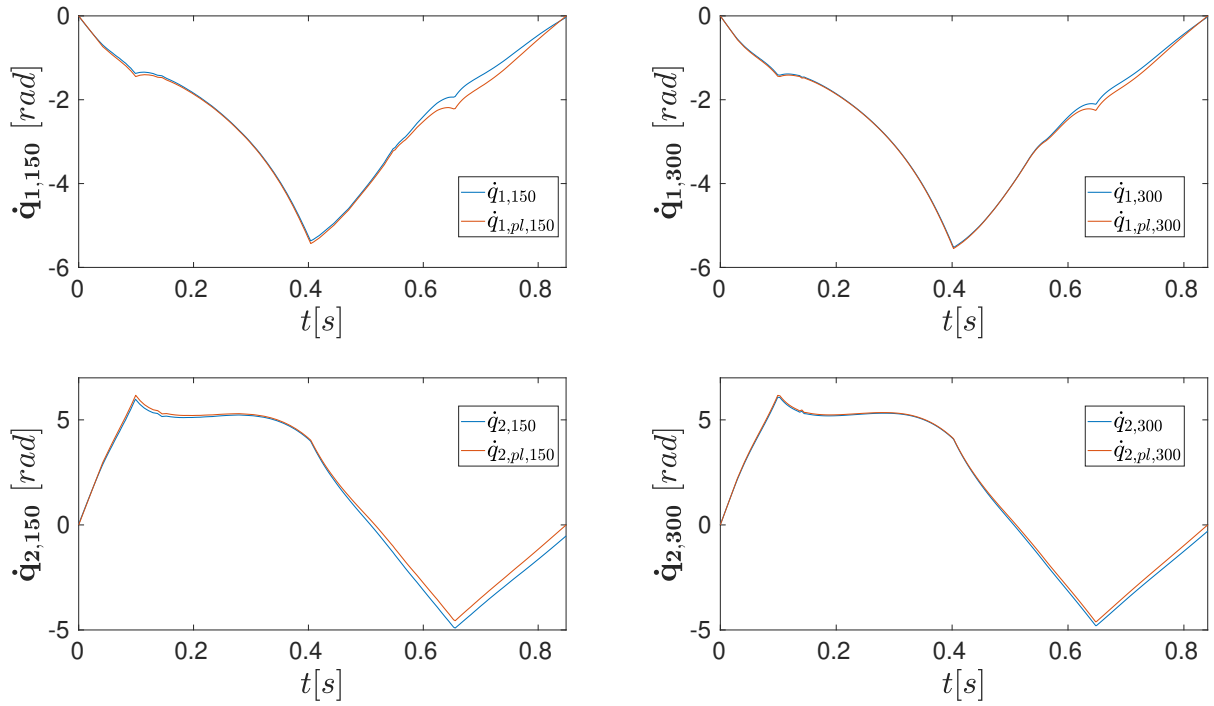

Figure 5.11 – Third simulation. The comparison between simulated joints' velocities (blue) and ideal ones (red) obtained from planning. On the left, the results from 1^{st} simulation with low grid resolution, on the right from 2^{nd} simulation with high grid resolution.

In Fig. 5.13 and 5.14 it is possible to compare the results obtained with JointEffortController, showing the simulated joints' positions and velocities, compared to the reference ones. It is worth to notice that the robot tracks the desired trajectory, unless small errors mostly in velocities, although the oscillations introduced by interpolation. The errors are due also to the planned bang-bang trajectory: it may be incompatible to a controller because of the actuators' saturation, which could have negative effects on controller performances. A solution could be to scale a bit the desired velocity, such that the actuators does not saturate, or to consider in planning a torque limit lower than the true one, but losing the globally-optimal solution. Also in this case, a trade-off should be found.

In Fig. 5.15 the planned and applied torques are reported. It is interesting to notice that applied torques are very oscillating because the controller needs to be very reactive in order to act so fast and to follow the trajectory. So, it is very sensible to small error variations. Since the desired velocity reference is not linear, but oscillating because of the interpolation, the showed behaviour is caused. In order to solve this problem, a better interpolation should be provided.

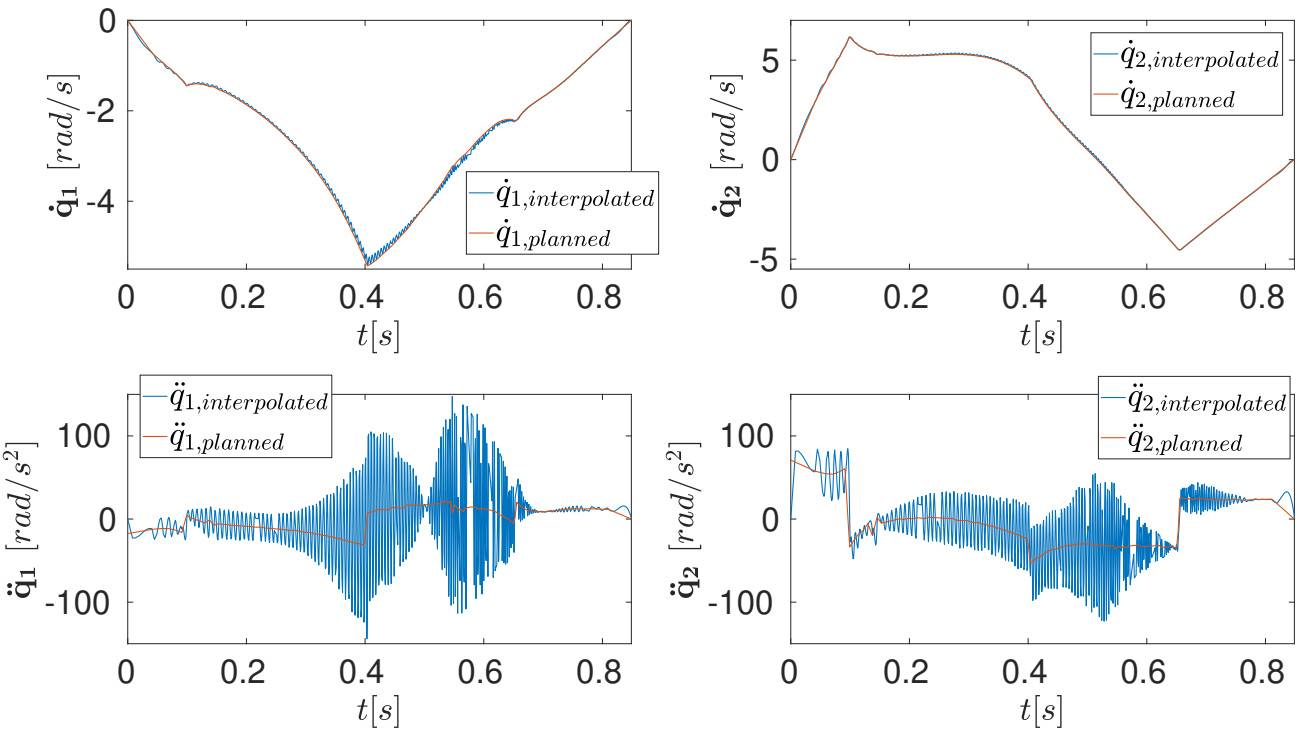

Figure 5.12 – Fourth simulation. The comparison between planned (red) and interpolated (blue) joints' velocities and accelerations.

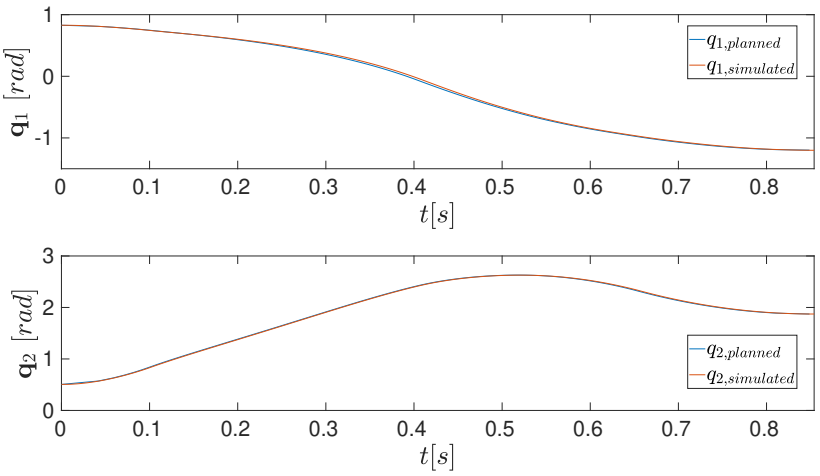

Figure 5.13 – Fourth simulation. The comparison between simulated (red) joints' positions and planned ones (blue).

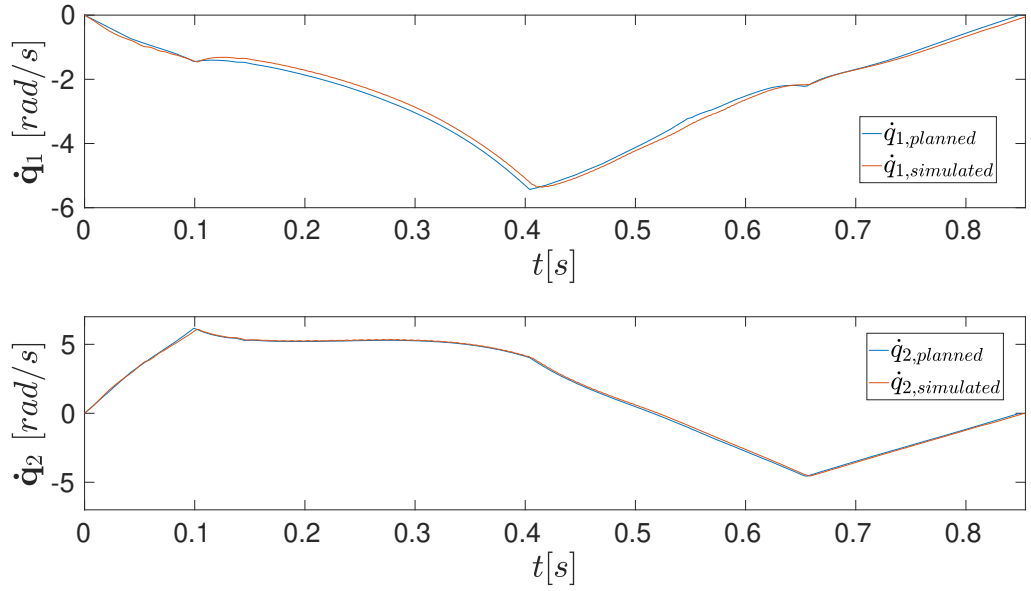

Figure 5.14 – Fourth simulation. The comparison between simulated (red) joints' velocities and planned ones (blue).

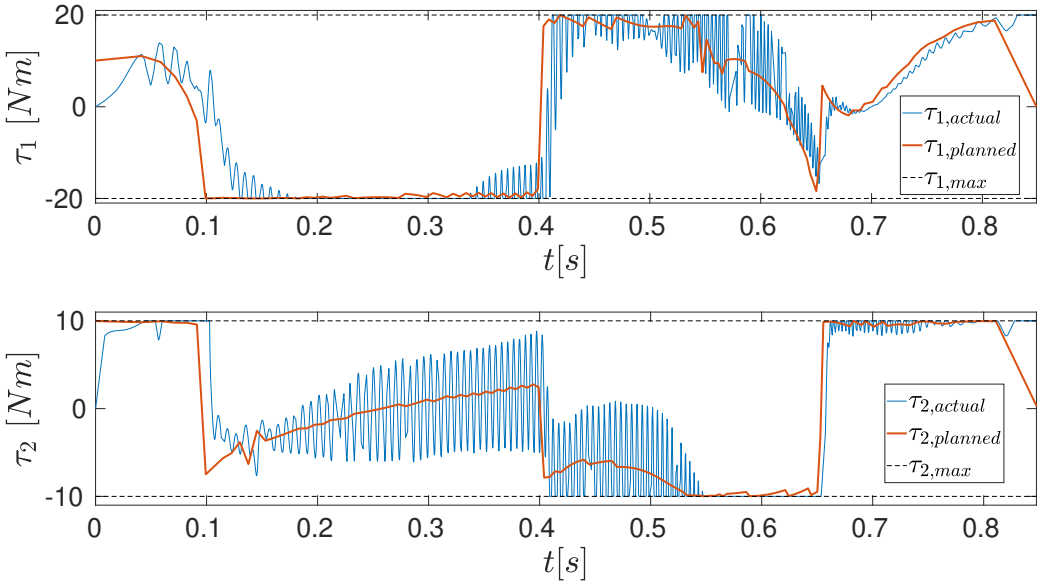

Figure 5.15 – Fourth simulation. The comparison between planned torques (red) and the applied one (blue).

Chapter 6

Conclusions

6.1 Results

Recently robotics is assuming a key role in space missions: it is capable to replace humans in dangerous and complex activities. In order for a robot to perform the required tasks, complex and advanced control architectures are required. One of the main issues in robotics is the motion planning: the robot must be capable to move to accomplish the desired activities. In some specific applications, it could be required to optimize the time execution of the motion in order to maximize the productivity and to minimize the spent resources.

The aim of this dissertation is to contribute to the development of technologies to implement the minimum time parametrization of trajectories. The following objectives has been defined:

- To design and to implement a time optimal planner.
- To integrate the planner in the available framework at ALTEC.
- To demonstrate the validity and to show the obtained results through the setup of a 3D dynamic simulator.

The predefined objectives has been reached, with the following contributions:

1. the development of a time optimal planner based on the dynamic programming approach and the definition of a new formulation for variables discretization.
2. The refactoring of the available framework separating problem specific details from the dynamic programming algorithm, effectively allowing to use dynamic programming for a generic constrained optimization problem via the implementation of a problem specific extension module.

3. the development of the robot-agnostic extension module responsible for the time optimal trajectory generation, able to parametrize path in both task and joint space.

4. the setup of the interface for communicating with the 3D dynamic simulator and the experimental validation and analysis of the planner through simulations.

6.2 Future works

The presented work offers several starting points for future researches, both to improve the efficiency of the algorithm and to develop new technologies.

Some suggestions for future works are:

- In order to improve the planning results, a more accurate approximation could be used to discretize the state space and the related variables. This could decrease the numerical errors affecting the planning variables. Besides, a different approximation could be most suitable to a continuous interpolation, such that the controller action is not affected by the injected oscillations.

- In order to speed up the execution of algorithm, some optimization factors could be found, such that the grid is not entirely explored, but only the more probable sections. Besides, the computational time could be decreased if parallel processing is used.

- A conjunction of the switching point method and the dynamic programming one could be used in order to exploit their advantages. In particular the dynamic programming could be used in the neighbourhood of the switching point, in order to avoid the jitter of the torque profile.

- Additional constraints could be considered in the planning algorithm. For example, the jerk constraints could allow to obtain torque profile more suitable to hardware implementations.

- New extension modules could be created to expand the capabilities of the dynamic programming algorithm. For example it could be used for generating time optimal trajectories for redundant manipulators.

Bibliography

[1] https://www.franka.de/.

[2] http://gazebosim.org/.

[3] https://www.altecspace.it/.

[4] https://orocos.org/.

[5] http://openrave.org/docs/0.8.2/openravepy/ikfast/.

[6] http://wiki.ros.org/.

[7] https://moveit.ros.org/.

[8] http://wiki.ros.org/rviz.

[9] http://wiki.ros.org/pluginlib.

[10] http://ompl.kavrakilab.org/.

[11] http://wiki.ros.org/stomp_motion_planner.

[12] http://wiki.ros.org/urdf.

[13] http://sdformat.org/.

[14] http://wiki.ros.org/ros_control.

[15] https://www.kinovarobotics.com/.

[16] https://it.mathworks.com/products/matlab.html.

[17] http://wiki.ros.org/kdl_parser.

[18] http://wiki.ros.org/joint_trajectory_controller.

[19] R. Bellman. The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60(6):503–515, 11 1954. URL https://projecteuclid.org:443/euclid.bams/1183519147.

[20] J. Bobrow, S. Dubowsky, and J. Gibson. Time-optimal control of robotic manipulators along specified paths. *The International Journal of Robotics Research*, 4(3):3–17, 1985. doi:10.1177/027836498500400301.

[21] A. C. H. Bradley, Stephen P. and T. L. Magnanti. *Applied Mathematical Programming*. Reading, MA: Addison-Wesley Publishing Company, 1977.

[22] F. Capasso. Analysis, design and implementation of an optimal planner for redundant robotic applications. Master's thesis, Politecnico di Torino, 2019.

[23] A. Casalino, A. M. Zanchettin, and P. Rocco. Online planning of optimal

trajectories on assigned paths with dynamic constraints for robot manipulators. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 979–985, Oct 2016. doi:10.1109/IROS.2016.7759168.

[24] Y. Chen and S. Y. . Chien. General structure of time-optimal control of robotic manipulators moving along prescribed paths. In *1992 American Control Conference*, pages 1510–1514, June 1992. doi:10.23919/ACC.1992.4792360.

[25] Y. Chen and A. A. Desrochers. Structure of minimum-time control law for robotic manipulators with constrained paths. In *Proceedings, 1989 International Conference on Robotics and Automation*, pages 971–976 vol.2, May 1989. doi:10.1109/ROBOT.1989.100107.

[26] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, and E. Fernández Perdomo. ros_control: A generic and simple control framework for ROS. *The Journal of Open Source Software*, 2017. doi:10.21105/joss.00456.

[27] D. Constantinescu and E. Croft. Smooth and time-optimal trajectory planning for industrial manipulators along specified path. *Journal of Robotic Systems - J ROBOTIC SYST*, 17:233–249, 05 2000. doi:10.1002/(SICI)1097-4563(200005)17:53.0.CO;2-Y.

[28] T. Dierks and S. Jagannathan. Online optimal control of nonlinear discrete-time systems using approximate dynamic programming. *Journal of Control Theory and Applications*, 9:361–369, 2011.

[29] E. Ferrentino and F. Salvioli. ROS/MoveIt! extension for redundancy resolution with dynamic programming. doi:10.5281/zenodo.3236880.

[30] R. Featherstone. *Robot Dynamics Algorithm*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[31] E. Ferrentino and P. Chiacchio. *A Topological Approach to Globally-Optimal Redundancy Resolution with Dynamic Programming*, pages 77–85. 01 2019. doi:10.1007/978-3-319-78963-7_11.

[32] E. Ferrentino, A. D. Cioppa, A. Marcelli, and P. Chiacchio. An evolutionary approach to time-optimal control of robotic manipulators. Unpublished.

[33] E. Ferrentino, F. Salvioli, and P. Chiacchio. Globally-optimal redundancy resolution with dynamic programming for robot planning: a ROS implementation. Unpublished.

[34] A. Gasparetto, P. Boscariol, A. Lanzutti, and R. Vidoni. Path planning and trajectory planning algorithms: A general overview. *Mechanisms and Machine Science*, 29:3–27, 03 2015. doi:10.1007/978-3-319-14705-5_1.

[35] F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research, 4th Ed.* Holden-Day, Inc., San Francisco, CA, USA, 1986.

[36] Kang G. Shin and N. McKay. Minimum-time control of robotic manipulators with geometric path constraints. *IEEE Transactions on Automatic Control*, 30(6):531–541, June 1985. doi:10.1109/TAC.1985.1104009.

[37] Kang G. Shin and N. McKay. A dynamic programming approach to trajectory planning of robotic manipulators. *IEEE Transactions on Automatic Control*, 31(6):491–500, June 1986. doi:10.1109/TAC.1986.1104317.

[38] Kim Doang Nguyen, I-Ming Chen, and Teck-Chew Ng. Planning algorithms for s-curve trajectories. In *2007 IEEE/ASME international conference on advanced intelligent mechatronics*, pages 1–6, Sep. 2007. doi:10.1109/AIM.2007.4412440.

[39] T. Kunz and M. Stilman. Time-optimal trajectory generation for path following with bounded acceleration and velocity. In *Robotics: Science and Systems*, pages 09–13, July 2012.

[40] Y. Nakamura and H. Hanafusa. Inverse Kinematic Solutions With Singularity Robustness for Robot Manipulator Control. *Journal of Dynamic Systems, Measurement, and Control*, 108(3):163–171, 09 1986, https://asmedigitalcollection.asme.org/dynamicsystems/article-pdf/108/3/163/5065441/163_1.pdf. doi:10.1115/1.3143764.

[41] J. M. O'Kane. *A Gentle Introduction to ROS*. Independently published, Oct. 2013. Available at `http://www.cse.sc.edu/~jokane/agitr/`.

[42] A. Pamanes, P. Wenger, and J. Zapata. Motion planning of redundant manipulators for specified trajectory tasks. *Advances in Robot Kinematics*, pages 203–212, 01 2002.

[43] F. Pfeiffer and R. Johanni. A concept for manipulator trajectory planning. *IEEE Journal on Robotics and Automation*, 3(2):115–123, April 1987. doi:10.1109/JRA.1987.1087090.

[44] H. Pham and Q. Pham. On the structure of the time-optimal path parameterization problem with third-order constraints. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 679–686, May 2017. doi:10.1109/ICRA.2017.7989084.

[45] Q. Pham. A general, fast, and robust implementation of the time-optimal path parameterization algorithm. *IEEE Transactions on Robotics*, 30(6):1533–1540, Dec 2014. doi:10.1109/TRO.2014.2351113.

[46] Y. Pyo, H. Cho, L. Jung, and D. Lim. *ROS Robot Programming*. ROBOTIS, 12 2017.

[47] Z. Shiller. On singular time-optimal control along specified paths. *IEEE Transactions on Robotics and Automation*, 10(4):561–566, Aug 1994. doi:10.1109/70.313107.

[48] Z. Shiller and H.-H. Lu. Computation of path constrained time optimal motions with dynamic singularities. *Journal of Dynamic Systems Measurement and Control-transactions of The Asme - J DYN SYST MEAS CONTR*, 114, 03 1992. doi:10.1115/1.2896505.

[49] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[50] S. Singh and M. C. Leu. Optimal Trajectory Generation for Robotic Manipulators Using Dynamic Programming. *Journal of Dynamic Systems, Measurement, and Control*, 109(2):88–96, 06 1987, https://asmedigitalcollection.asme.org/dynamicsystems/article-pdf/109/2/88/5006019/88_1.pdf. doi:10.1115/1.3143842.

[51] J. . E. Slotine and H. S. Yang. Improving the efficiency of time-optimal path-following algorithms. *IEEE Transactions on Robotics and Automation*, 5(1):118–124, Feb 1989. doi:10.1109/70.88024.

[52] M. Tarkiainen and Z. Shiller. Time optimal motions of manipulators with actuator dynamics. volume 2, pages 725 – 730 vol.2, 06 1993. doi:10.1109/ROBOT.1993.291873.

[53] G. Visentin. Space robotics. In M. O. Tokhi, G. S. Virk, and M. A. Hossain, editors, *Climbing and Walking Robots*, pages 27–37, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[54] M. W. Walker and D. E. Orin. Efficient Dynamic Computer Simulation of Robotic Mechanisms. *Journal of Dynamic Systems, Measurement, and Control*, 104(3):205–211, 09 1982, https://asmedigitalcollection.asme.org/dynamicsystems/article-pdf/104/3/205/4802773/205_1.pdf. doi:10.1115/1.3139699.

[55] Q. Zhang, S. Li, and X. Gao. Practical smooth minimum time trajectory planning for path following robotic manipulators. In *2013 American Control Conference*, pages 2778–2783, June 2013. doi:10.1109/ACC.2013.6580255.