POLITECNICO DI TORINO

Master degree course in Mechatronic Engineering

Master Degree Thesis

Analysis, design and implementation of an optimal planner for redundant robotic applications

Supervisor
prof.ssa Marina Indri

Candidate
Fabio Capasso

ALTEC Advisor
ing. Federico Salvioli

April, 2019

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Space exploration of celestial structures is conducted by both unmanned robotic space probes and human spaceflight. Explorative missions are, generally, very expensive, performed in hostile environment and extremely risky. Such missions are often more suited to robots rather than men, due to lower costs and lower risk factors, but employed systems need to be highly reliable to compensate for human distance.

On the contrary, the investigation of new areas requires robots able to explore unfamiliar territories and capable of dexterous manipulations. Suitable robotic systems for these jobs are those with the ability to proceed on uneven terrains, handle objects and use tools.

However, future space mission scenarios are shifting from robotic exploration towards planetary colonization. These scenarios demand robots, either as human helpers or precursors, capable of supporting construction specific use cases, such as object manipulation, assembling of large structures, material transportation, terrain selection and preparation, etc. In addition, since they require physical interaction with the environment, these tasks are more complex than explorative ones.

In these circumstances, engineers can rely on cooperative robots, which are systems consisting of multiple robots that can collaborate with each other to accomplish specific tasks that individual robotic systems cannot achieve. Although they grant huge flexibility, the combination of two or more robotic units increases the complexity of the whole system and, so, the control of the structure.

Mobile manipulators are used in the present dissertation as an example of representatively complex robotic systems which are not trivial to control. Basically, mobile manipulators are systems composed of a moving platform upon which a robotic arm is installed, so they are very suited for manipulating, transporting and assembling large structures within constructions.

Moreover, due to environment asperity, resources available to space robots are normally very poor. Provided that robots have sufficient autonomy for the mission and in order to satisfy reliability requirements, the ground segment is much more suitable for performing optimal planning than the robot on-board systems, allowing to exploit computational resources at low cost. For this reason, remotely supervised expeditions are often directed with long distance operational communications. These communications are characterized by an off-line transmission, which is a time-delayed data transfer between the ground station and the landed asset. Commonly, time delays for signal transmission vary from a few minutes to several hours, thus, control centers have a predefined visibility window in which all sort of off-line calculations for planning purposes are conducted.

1.2 Focus

When dealing with cooperative robots, and specifically with mobile manipulators, modern robot control approaches assume that the motion of the mobile platform is executed first, and, once the moving base is settled, the manipulator activity starts. From a synergetic point of view, it would be better if the robot acted as a whole automated system while accomplishing a given objective. This way, instead of having an assigned task split into two sub-tasks, the operation would be completed in just one phase, obtaining benefits in terms of performance and execution time.

To make this possible, several problems have to be overcome, such as, how to command the robot in its entirety, how to efficiently exploit the complexity of the structure in order to minimize resources depletion, how to take into account every feasible motion and select the right one, even in case of harsh terrain or large obstacles. These difficulties are only some of the major challenges that companies have to deal with when facing up the exploration topic.

Among the companies supporting planetary exploration missions, the Aerospace Logistics Technology Engineering Company (ALTEC) is the Italian center of excellence for the provision of engineering and logistics services [1] for the aerospace. ALTEC is currently developing advanced technologies for ground control centers to support planning and control of redundant robotic systems, with the aim to improve the next explorative missions and sustain operations on board the International Space Station (ISS).

1.3 Aims and Objectives

The aim of the present dissertation is to contribute to the development of the aforementioned technologies in the measure of producing software deliverables and the corresponding documentation.

The objective of this discussion is to reach the suggested goal through the following steps:

1. build a plugin that performs the inverse kinematics of mobile manipulators, i.e., systems composed of a manipulator mounted on a mobile base
2. generalize a redundancy resolution service and using it in combination with the aforementioned module to test its application in optimal planning of trajectories

Besides space exploration, the newly developed tools could be employed in several fields, such as welding (manipulator mounted on a slider), painting, assembling, or in activities that require the motion in narrow or unsafe areas. Thanks to the exploitation of possible redundant degrees of freedom, these tools will provide globally-optimal ways to perform the assigned operations, offering the possibility to choose the quality metrics (e.g., energy consumption) to be optimized during the execution of the task.

1.4 Outline

Next chapters are arranged as follows.

Chapter 2 introduces the robot planning problem from the control point of view and offers a depiction of the workflow of operations executed during the planning of robotic tasks.

Chapter 3 investigates the most recent stage in the development of redundancy resolution tools and techniques, and explores the inverse kinematics problem presenting two procedures to achieve the globally-optimal solution.

Chapter 4 analyzes the main problems when dealing with the creation of an optimal planner, examining implementation aspects as well.

Chapter 5 shows the methods applied to solve the inverse kinematics of mobile manipulators, and the building of an algorithm that collects information from the previous plugin in order to compute the globally-optimal path for the assigned task.

Chapter 6 discusses the results acquired from tests performed on the software packages implementing inverse kinematics and optimal path planning.

Chapter 7 reviews the objectives of the dissertation and checks if they have been completely attained. At last, some suggestions for future works are provided.

Chapter 2

Robot Planning

2.1 The Planning Problem

In robotic exploration, missions are assigned to robotic systems rather than human spaceflight. Robots can be asked to carry out diverse tasks, such as to reach a location, to handle materials, to utilize tools, to grasp or release objects, to exert forces or torques, etc. These missions share the common problem of planning the robot motion in every single aspect. The planning problem can be decomposed into a number of activities categorized in a hierarchical structure [2].

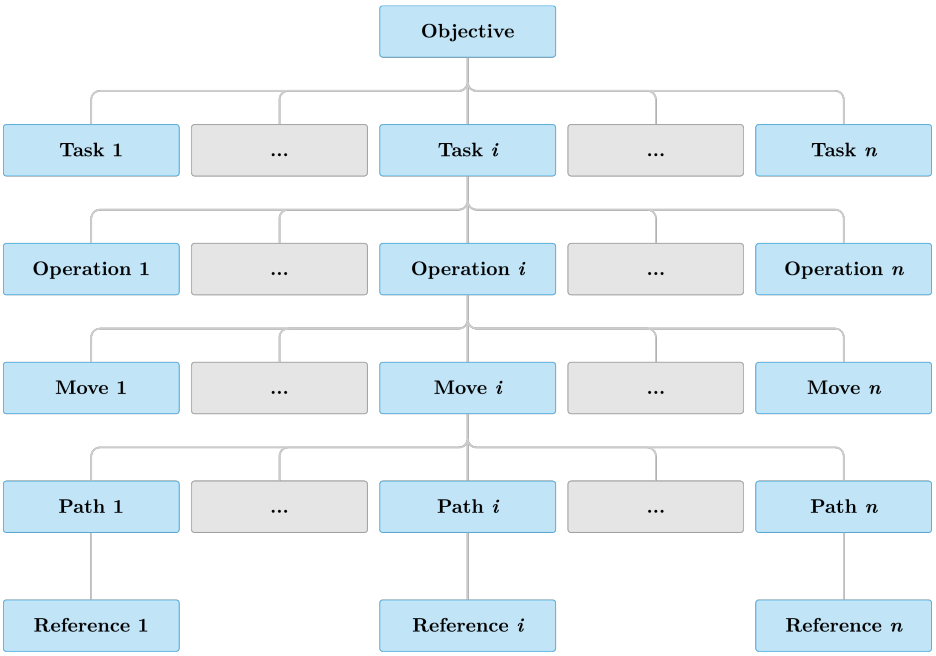


Figure 2.1: Diagram depicting the activity hierarchy in a planning problem.

Objective

At the highest level in the hierarchy, there is the objective. Once a mission is designated, then a goal is established and the objectives to achieve that goal are defined. For instance, if the appointed mission is the exploration of new areas, one goal might be the discovery of buildings or constructions and the objective would be the reaching of a specified location using a robotic system. Thereby, for a given goal, one or more objectives are delineated.

Task

Each objective is characterized by at least one task that has to be executed. A task defines a group of actions or operations to be accomplished for the attainment of the objective. For instance, some tasks corresponding to the previously defined objective would be the conduction of the robot to the location or the acquisition and analysis of data received by the payloads mounted on it.

Operation

For a given task, the operation defines one of the single activities the task is composed of. For instance, if the assigned task is to lead the robot to a location, the corresponding operations might be the motion of a platform on a particular terrain, or the manipulation of an arm to remove possible obstacles placed on the route.

Move

Often, operations require the motion of some part of the considered system. The move action defines a single motion that must be executed to perform an operation. For instance, manipulation operations may request the motion of parts of the arm, the attaining of a pose, or the grasping act by means of the manipulator's hand.

Path

The elementary move can be decomposed in one or more geometrical paths, if no time law is defined, or trajectories, if a time law and kinematic constraints are defined. For instance, in order to move a robotic arm end-effector a trajectory must be first determined, so that the manipulator TCP can be moved along the discretized path with certain velocity and acceleration constraints.

Reference

At the lowest level in the hierarchy, there is the reference. It consists of the data vector obtained from the kinematic inversion once the path or trajectory is sampled. The reference signal is, at last, supplied to motors for their control.

2.2 Path Planning and Trajectory Planning

One of the most common robotics tasks is the motion of the robot end-effector along a prescribed path. The fulfillment of this action involves the creation of a trajectory planner that computes, at each time sample, the reference needed by the controller to regulate the motion of the system. The trajectory planner is a software "node" that, given the desired path, computes the joint reference values (for the controller block), according to the robot kinematic constraints (max velocity, etc.) and dynamic constraints (max accelerations, max torques, etc.) [2].

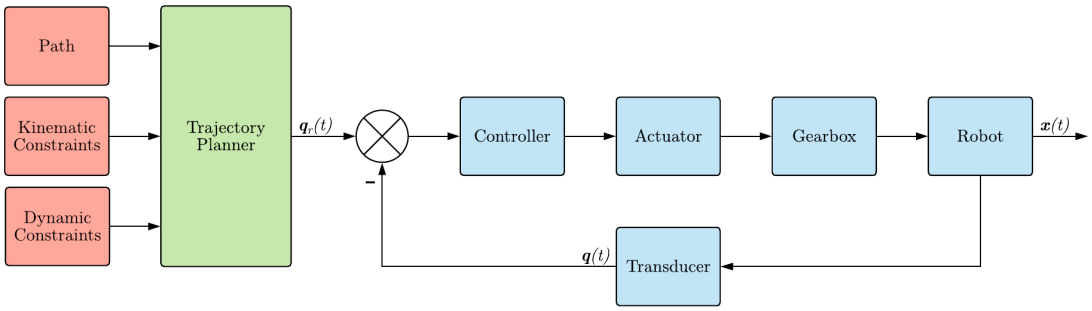


Figure 2.2: Scheme of a trajectory planner followed by a control system.

Trajectory planners make use of different algorithms to compute the reference vector. Typically, there are two kinds of algorithms used for building a trajectory planner: path planning algorithms and trajectory planning algorithms.

For the sake of clarity, it is worth making a distinction among path planning and trajectory planning, but first, it is necessary to define the configuration space.

Given a robot with n-links, a *configuration* is a complete specification of the robot location, generally, by means of a set of joint coordinates $\boldsymbol{q}$. The set of all possible configurations is known as the *configuration space*, or *joint space*. Given a robot with n-links and its configuration space, the subset of the special Euclidian group $SE(3) = \mathbb{R}^3 \times SO(3)$, where the robot motion takes place, is called *workspace* of the robot, or *task space*. The workspace might contain obstacles.

Path planning is the task of finding a path in the configuration space, that is, finding a function which connects an initial configuration and a final configuration, that does not collide any obstacle as the robot traverses the path.

Trajectory planning is the task of finding a trajectory in the configuration space, i.e., a path parametrized with time. As a result, trajectory planning implies the possibility to make specifications on the kinematic constraints, such as velocity or acceleration.

Path planning algorithms generate collision-free paths through configuration space. Obstacles are converted (with the aid of a collision checker) from the workspace to configuration space, and geometric algorithms are used to search through configuration space for a path, from start to goal, that does not collide with any obstacles (including robot self-collisions). Once a geometric path is found, it has no timing element. Since the robot could move along the path with different speed patterns, there are infinite trajectories per path. Trajectories can be seen as paths endowed with time information, made up of a set of *waypoints*, i.e., discrete points spread across the path, often at a fixed time interval. The scope of trajectory planning algorithms is to generate trajectories segments between two consecutive waypoints, taking into account physical constraints such as collision avoidance, joint limits, velocity limits, acceleration limits, jerk limits, torque bounds, etc.

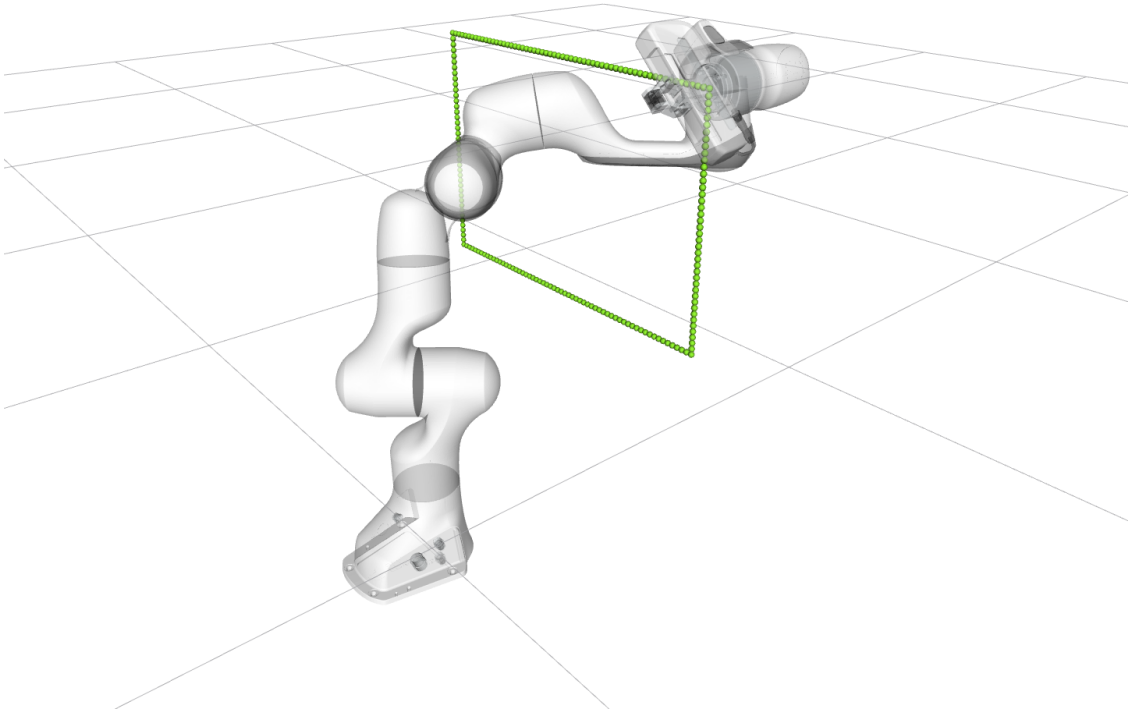


Figure 2.3: Panda manipulator TCP proceeding through trajectory waypoints.

Kinematic Inversion

Robot planning for space exploration is executed by ground control centers and robotic systems are remotely controlled from afar. The executed trajectory $\boldsymbol{x}(t)$ is computed on board the robot and sent to the ground control center. There, the robot planning begins with the computation of an optimal path, starting from the last position received from the robot. The resulting geometric path $\boldsymbol{x}_r(\lambda)$, calculated by the path planner, does not depend on time. Indeed, λ is a variable used to parametrize the path $\boldsymbol{x}$ independently from its variation with time. The following step is to feed in the path $\boldsymbol{x}_r(\lambda)$ to the trajectory planner, which returns a discrete trajectory in the time variable $\boldsymbol{x}_r(t)$. Ultimately, the inverse kinematics for that robotic system is computed from $\boldsymbol{x}_r(t)$ and the calculated configuration $\boldsymbol{q}_r(t)$ is sent back to the robot.

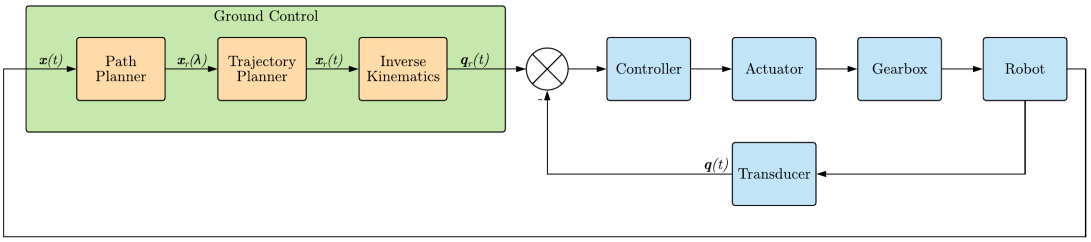


Figure 2.4: Scheme of a closed-loop control system with trajectory planner.

An alternative scheme to the previous one has the blocks *Trajectory Planner* and *Inverse Kinematics* swapped. While with scheme in Figure 2.4 the trajectory planning derives from a conservative process, the alternative grants the satisfaction of kinematic and dynamic constraints after the planning is performed.

One big challenge in this context is the resolution of the inverse kinematics problem. When performing kinematic inversion on compound robots, the evaluation of the reference vector $\boldsymbol{q}_r(t)$ becomes a hard process. Sometimes, due to the complexity of the problem, heavy computational loads are required to solve the inverse kinematics, implying a lot of time consumed for this operation. Provided with the most advanced technologies, ground centers take care of robot planning, minimizing the time spent in numerical calculations thanks to a huge computational power availability. Moreover, in space exploration missions, a priori information about the environment can be obtained, which is another reason in favor of offline planning and the usage of offline planners.

Chapter 3

State of the Art on Redundancy Resolution

3.1 Robotic Manipulators

A *robotic manipulator*, or robotic arm, is a machine capable of automatically carrying out a series of actions. Many robotic arms are designed with shape that resembles a human arm, consisting of a shoulder, an elbow and a final wrist. Connected to the wrist there is an *end-effector*, or robotic hand, that is a tool employed in the handling of devices or instruments.

Robotic manipulators have a mechanical structure, modeled by means of links or bars, driven by joints such as electrical actuators. This structure is described by geometric entities, called *kinematic chains*, that are characterized by a number of links connected by joints. Kinematic chains are either open chains, if there is only one link between any two joints, or closed chains, if there is more than one link between two joints, and manipulators typically fall into the first category.

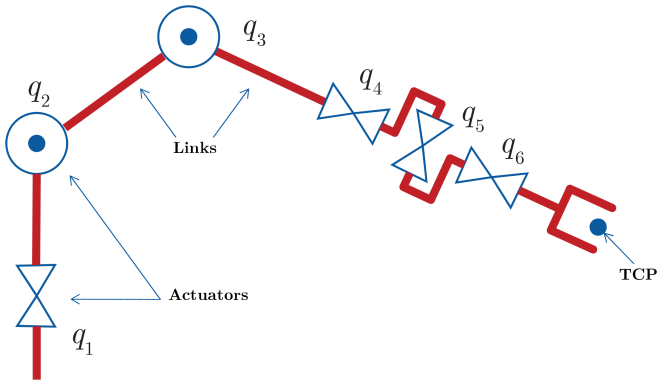


Figure 3.1: Kinematics chain of a 6-DOF manipulator.

Joints, or kinematic pairs, represent connections between two bodies that impose constraints on their relative motion. Depending on the number of constraints imposed by the joint, they are able to grant one or more *degrees of freedom* (DOF) to the paired bodies. In general, supposing that only 1-DOF kinematic pairs are employed, the number of *degrees of motion* (DOM) corresponds to the number of joints of the robot. As this number grows, the robot acquires a greater ability to move freely in the space, i.e., there are multiple ways to fulfill a given task.

The tasks assigned to a robotic arm can be disparate. For instance, many ordinary tasks involve the 'pick and place' action; basically, the robotic hand, usually provided with a gripper, grabs an object, which is taken from a location and brought to another one. Other common tasks are those required in assembly operations, welding, painting, drilling, handling of equipment, which can be more or less complicated. Whatever task will be assigned to a robot, a number of degrees of freedom can always be determined for that task. This quantity represents the number of independent variables that characterize or are required by the task reference frame.

Every task requires the motion of the robot. To this purpose commands are sent to the robotic system, but before giving any instruction to a real-world robot, virtual simulations are performed. Using simulation tools to design robots is a simple and cheap method to build complex robots [3], at least virtually. A robot model that emulates real-world processes is built into a virtual environment so to test any action that the actual robot will execute.

The *tool center point* (TCP) is adopted as a reference to move the robot end-effector inside the simulation environment. It is an ideal point chosen on the manipulator end-effector, usually in the middle of it, that is also used to set a target point in the *task space*, or Cartesian space. The number of independent variables in the joint space that describe the TCP reference frame in the task space determines the number of degrees of freedom of the TCP.

It is comprehensible that, while the number of DOM of the robot can be as large as desired, the number of DOF of the task can only assume fixed values. For instance, a task for which it is required to fix the position and orientation of a manipulator end-effector, in the 2-dimensional plane, is characterized by a number of DOF equal to 3; while, the same task executed in the 3-dimensional space is characterized by a number of DOF equal to 6. Other examples of typical manipulator tasks are: positioning the end-effector in the 2D plane ($DOF = 2$), orienting the end-effector in the 2D plane ($DOF = 1$), positioning the end-effector in the 3D space ($DOF = 3$), orienting the end-effector in the 3D space ($DOF = 3$). Frequently, more complex tasks are assigned to a manipulator such as to follow a task space trajectory. A common case is that of a trajectory in the 3D space with constrained roll/pitch orientation, for which the number of DOF is equal to 5 instead of 6. Ordinary examples are laser cutting, painting, welding, etc.

3.2 Redundant Robots

When a task with DOF m is assigned to a robot having DOM n, three main conditions can be identified.

Case 1	$n < m$
Case 2	$n = m$
Case 3	$n > m$

Table 3.1: Redundancy table

The first case deals with robots whose DOM does not allow to perform the specified task, consequently, it is not relevant to any application.

The second case, the most common one, is associated with *non-redundant* robots, that is, robots having a robot DOM that matches the task DOF. The number of variables required for the task is equal to the number of variables the TCP is provided. This situation happens, for instance, when a 6-revolute-joints (6R) robotic manipulator is employed to draw some 3-dimensional task space trajectory with constrained orientation.

The third case shows the condition where the DOM of the robot is higher than the task DOF. It represents a condition for which the robot is said to be *redundant* with respect to the assigned task: the robotic system is able to select a unique solution from a set of infinite alternatives to accomplish the given task. An example of this condition is what happens when a 7-revolute-joints (7R) manipulator is employed to draw some 3-dimensional task space trajectory with constrained orientation. This is the case of utmost relevance for real applications, since, once fixed the task space as a 3D 6-DOF space, robots having a DOM higher than 6 are able to exploit their redundancy capability in order to achieve better performance according to some cost functional.

The measure of the redundancy capability of a robot requires the introduction of a new concept: the *degree of redundancy*. Given a prescribed task with DOF m, the degree of redundancy r is the number of extra degrees of freedom the robot is provided with respect to those necessary to fulfill that task:

$$r = n - m \tag{3.1}$$

3.3 Kinematic Inversion

The manipulator forward kinematic function $\boldsymbol{f}$ [4] is a non-linear vector function which relates a set of n joints coordinates $\boldsymbol{q}$ to a set of m end-effector coordinates:

$$\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{q}) \tag{3.2}$$

One of the primary issues of practical interest in manipulator kinematics is determining the inverse kinematic function $\boldsymbol{f}^{-1}$. This function computes one or more sets of joint variables $\boldsymbol{q}$ (angles if revolute joints, linear displacements if prismatic joints) that place the manipulator end-effector in a desired position and orientation $\boldsymbol{x}$, also known as *pose*:

$$\boldsymbol{q} = \boldsymbol{f}^{-1}(\boldsymbol{x}) \tag{3.3}$$

The process that aims at finding such a point in the joint space is termed *kinematic inversion*. For non-redundant manipulators, there is a bounded set of distinct configurations which satisfy (3.3). For redundant manipulators, there is an infinite number of configurations which satisfy the relation in (3.3).

3.3.1 Resolution Methods

From (3.3), it is clear that the inverse kinematics problem for a manipulator requires the resolution of a set of nonlinear equations [5]. For a solution to exist, the desired position and orientation of the end-effector must lie in the workspace of the manipulator. In cases where solutions do exist, they often cannot be determined in a closed form, so numerical methods are required. Several techniques [6] or a combination of them can be used to solve the inverse kinematics of a manipulator. Basically, they are divided into *analytical* methods and *numerical* methods.

Analytical Methods

An analytic solution to the inverse kinematics problem is a closed-form expression. Closed-form solutions are desirable because they are, in general, faster than numerical solutions and immediately identify all possible results. For instance, when manipulators are not provided of a spherical wrist, that is, when the three wrist axes of rotations do not intersect at one point, the inverse kinematics solutions for the shoulder (and the elbow, if any) can be easily obtained through a simple analytic formulation. The drawback of this method is that it does not guarantee a solution for generic structures, since it is robot dependent. Also, even if a closed-form solution existed, it would get more and more complex with the growth of the number of DOM. In general, analytic solutions can only be obtained for 6-DOM systems with specific kinematic structures [5], so most industrial manipulators have such structures because they permit more efficient software coordination.

Numerical Methods

Unlike analytical methods, numerical methods are not robot dependent, and can be applied to any kinematic structure. The disadvantage of numerical solutions is that they can be slower than analytical solutions and, in some cases, they do not allow the computation of all the possible solutions. Among numerical methods, there are iterative approaches that solve the kinematics problem by successive approximations so to obtain progrssively better solutions. Some worth mentioning iterative methods [6] are the following:

- Jacobian inversion technique
  - + computationally fast
  - + works fine with simple or complex structures
  - − singular configurations cannot be handled as the inversion of the Jacobian is undefined
  - − unpredictable joint configurations
- Jacobian transpose technique
  - + simpler calculations with respect to the previous method, since matrix inversion is not required
  - − needs many iterations until convergence in certain configurations
  - − unpredictable joint configurations
- Optimization based techniques
  - + explicit optimization criterion provides control over manipulator configurations
  - − numerical problems at singularities
  - − non conservative
- Cyclic coordinate descent (CCD)
  - + singularity-free and suitable for real-time applications
  - − does not work well with complex structures
- Genetic programming
  - + useful for complex and independent motion control
  - − very slow, in general

Each of them has their own advantages and disadvantages, but, being numerical methods, their solution is just an approximation of the analytic one. Furthermore, in some cases the analytic solution is not the true inverse kinematics solution, but an approximation as well. That is why, if available and computationally efficient, an analytic solution is always preferred over a numerical solution.

3.3.2 Inverse Kinematics of Redundant Manipulators

Over the last decades, a large body of literature has appeared concerning the inverse kinematics resolution of redundant robots. Many papers have been published discussing local and global approaches for the optimal resolution of the problem.

Some papers cover the problem from an engineering point of view, by means of locally-optimal techniques.
Carignan [7] used the energy consumption as cost function to search for a solution to the problem.
Nedungadi [8] developed a locally-optimal algorithm focusing on drive-forces.
Kazerounian [9] showed the difference between locally-optimal and globally-optimal control approaches.

Some other papers cover the problem applying globally-optimal techniques, exploiting procedures proper of the calculus of variations.
Nakamura [10] has solved the globally-optimal control problem by minimizing the norm of the joint velocities vector over the entire trajectory.
Hanafusa [11] extends the previous work to obstacle avoidance.
Kyriakopoulos [12] tries to solve the problem by minimizing the jerks, the time derivative of the joint accelerations.

3.3.3 Inverse Kinematics of Non-Redundant Manipulators

For 6-DOM manipulators to which a 6-DOF task is assigned, the system of equations (3.3) to solve is square, that is, it has a number of equations equal to the number of unknowns. This implies that, for non-redundant manipulators, none, one or multiple solutions can be found. Literature related to the inverse kinematic problem of non-redundant robots is, now, consolidated from both the mathematical and engineering point of view.

Papers that cover the problem from a mathematical point of view, investigate the maximum number of achievable kinematics solutions.
The problem of finding all possible solutions has been addressed by Tsai and Morgan [13], who applied homotopy map techniques to the kinematic inversion of 6R and 5R arms and developed a numerical method guaranteed to find all isolated solutions. An application example of this homotopy map method proved that the number of real inverse kinematics solutions could be as high as 12.
Duffy and Crane [14] have shown that the inverse kinematic problem of a 6-DOM arm can be expressed in terms of a 32nd-degree polynomial in one joint variable.
Some work by Lee and Liang [15] reduces the degree of the polynomial to 16. This result puts an upper bound of 16 on the number of distinct configurations by which a 6-DOM general manipulator can achieve a given end-effector pose.
Thanks to Manseur and Doty [16] the previous theoretical analysis has been practically verified on a 6R manipulator, thereby closing the proof that 16 is indeed the maximum number of inverse kinematics solutions for 6-DOM manipulators.

3.4 Redundancy Resolution

The redundant degrees of freedom can be used to optimize manipulator properties or perform additional tasks. As a result, much previous work in redundant manipulator inverse kinematic analysis has been closely tied to the redundancy resolution process, where algorithms are developed to determine the motion of the joints in order to achieve end-effector trajectory control. The common thread is that redundancy should be resolved in such a way that the robot optimizes some performance index while carrying out its given task.

At the most basic level, researchers have proposed redundancy resolution methods involving optimized solutions to the kinematic inversion. The complexity of this problem is highly dependent on the geometry of the manipulator, and the inverse kinematics solution is, usually, not unique. The exact number of solutions depends on the manipulator architecture as well as the desired end-effector trajectory.

3.4.1 Locally-Optimal Inverse Kinematics

Many previous investigations [17, 18, 11, 19] have focused on the linearized first order instantaneous kinematic relation between end-effector velocities and joint velocities:

$$\dot{\boldsymbol{x}} = \boldsymbol{J}(\boldsymbol{q})\ \dot{\boldsymbol{q}} \tag{3.4}$$

where $\dot{\boldsymbol{x}} \in \mathbb{R}^m$ and $\dot{\boldsymbol{q}} \in \mathbb{R}^n$ are, respectively, the *task space* and *joint space* velocity vectors, while $\boldsymbol{J}(\boldsymbol{q}) = \partial \boldsymbol{f}/\partial \boldsymbol{q} \in \mathbb{R}^{m \times n}$ is the analytic Jacobian of the manipulator. Once joint velocities $\dot{\boldsymbol{q}}$ are known, it is possible to compute the corresponding joint positions $\boldsymbol{q}$ via numerical integration. Eventually, appropriate algorithms are used to minimize the errors introduced by the numerical integration [20].

Since a redundant manipulator is employed, system (3.4) is not square, i.e., it has more unknowns than equations, or, more rows than columns. Thus, set $\dot{\boldsymbol{x}}$, the aforementioned system admits infinite solutions in $\dot{\boldsymbol{q}}$.

Despite the complexity, a solution to this problem can be obtained if, besides the system of equations (3.4), an additional requirement is considered: a common choice is the optimization of a cost function g.

Given $\dot{\boldsymbol{x}}$, the question is how to find the solution vector $\dot{\boldsymbol{q}}$ that, at the same time, satisfies (3.4) and minimizes the following quadratic cost function:

$$\begin{aligned} \min_{\boldsymbol{q}(t)} g(\dot{\boldsymbol{q}}) &= \frac{1}{2}\ \dot{\boldsymbol{q}}^{\intercal} \boldsymbol{W} \dot{\boldsymbol{q}} \\ s.t. \quad \dot{\boldsymbol{x}} &= \boldsymbol{J}(\boldsymbol{q})\ \dot{\boldsymbol{q}} \end{aligned} \tag{3.5}$$

where $\boldsymbol{W} \in \mathbb{R}^{n \times n}$ is a weight matrix, symmetric and positive definite, whose elements reflect the relative cost of utilizing each joint, for instance, in terms of energy or time. The choice of a quadratic cost function grants a closed-form solution to the problem.

One way to solve (3.5) is by means of the *Lagrange multipliers method*: a strategy for finding the local maxima and minima of a function $g(\dot{\boldsymbol{q}})$ subject to equality constraints $\boldsymbol{h}(\dot{\boldsymbol{q}}) = 0$, i.e., subject to the condition that one or more equations have to be satisfied exactly by the chosen values of the variables.

Introducing the *Lagrange multipliers vector* $\boldsymbol{\lambda}$, which allows to incorporate constraint (3.4) into a wider cost function, it is possible to study the *Lagrange function* (or Lagrangian) defined by:

$$\mathcal{L}(\dot{\boldsymbol{q}}, \boldsymbol{\lambda}) = g(\dot{\boldsymbol{q}}) + \boldsymbol{\lambda} \cdot \boldsymbol{h}(\dot{\boldsymbol{q}}) \tag{3.6}$$

which, in this specific case, becomes:

$$\mathcal{L}(\dot{\boldsymbol{q}}, \boldsymbol{\lambda}) = \frac{1}{2}\, \dot{\boldsymbol{q}}^{\intercal} \boldsymbol{W} \dot{\boldsymbol{q}} + \boldsymbol{\lambda}^{\intercal}(\dot{\boldsymbol{x}} - \boldsymbol{J}(\boldsymbol{q})\, \dot{\boldsymbol{q}}) \tag{3.7}$$

As a consequence, the sought solution has to satisfy the following two conditions:

$$\begin{aligned} \frac{\partial \mathcal{L}^{\intercal}}{\partial \dot{\boldsymbol{q}}} &= 0 \\ \frac{\partial \mathcal{L}^{\intercal}}{\partial \boldsymbol{\lambda}} &= 0 \end{aligned} \tag{3.8}$$

By merging their results, a new relation is obtained:

$$\dot{\boldsymbol{q}} = \boldsymbol{W}^{-1} \boldsymbol{J}^{\intercal}(\boldsymbol{q}) \boldsymbol{\lambda} \tag{3.9}$$

which, combined with (3.4), gives:

$$\dot{\boldsymbol{x}} = \boldsymbol{J}(\boldsymbol{q}) \boldsymbol{W}^{-1} \boldsymbol{J}^{\intercal}(\boldsymbol{q}) \boldsymbol{\lambda} \tag{3.10}$$

Assuming $\boldsymbol{J}(\boldsymbol{q})$ full rank, then $\boldsymbol{J}(\boldsymbol{q}) \boldsymbol{W}^{-1} \boldsymbol{J}^{\intercal}(\boldsymbol{q})$ is a square full rank matrix and, thus, invertible. Solving the previous expression in $\boldsymbol{\lambda}$:

$$\boldsymbol{\lambda} = (\boldsymbol{J}(\boldsymbol{q}) \boldsymbol{W}^{-1} \boldsymbol{J}^{\intercal}(\boldsymbol{q}))^{-1} \dot{\boldsymbol{x}} \tag{3.11}$$

and substituting in (3.9), it yields to:

$$\dot{\boldsymbol{q}} = \boldsymbol{W}^{-1} \boldsymbol{J}^{\intercal}(\boldsymbol{q}) (\boldsymbol{J}(\boldsymbol{q}) \boldsymbol{W}^{-1} \boldsymbol{J}^{\intercal}(\boldsymbol{q}))^{-1} \dot{\boldsymbol{x}} \tag{3.12}$$

The previous equivalence is also written as:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}_W^{\dagger}(\boldsymbol{q}) \dot{\boldsymbol{x}} \tag{3.13}$$

where $\boldsymbol{J}_W^{\dagger}(\boldsymbol{q})$ is the weighted pseudoinverse of the manipulator Jacobian $\boldsymbol{J}(\boldsymbol{q})$, which instantaneously minimizes the quadratic form $\dot{\boldsymbol{q}}^{\intercal} \boldsymbol{W} \dot{\boldsymbol{q}}$ at configuration $\boldsymbol{q}$.

The above solution can be generalized by adding another quantity to the right-hand side of the equality, as stated by the fundamental theorem of linear algebra [21]: given a matrix $\boldsymbol{A}$, any vector $\boldsymbol{v}$ can be expressed as the sum of two orthogonal vectors, one in the range of $\boldsymbol{A}^\intercal$, and one in the null-space of $\boldsymbol{A}$.
On the trail of this, the general form of the solution to problem (3.5) is:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}_W^\dagger(\boldsymbol{q})\dot{\boldsymbol{x}} + (\boldsymbol{I} - \boldsymbol{J}_W^\dagger(\boldsymbol{q})\boldsymbol{J}(\boldsymbol{q}))\ \boldsymbol{\nu} \tag{3.14}$$

or, in a simpler form:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}_W^\dagger(\boldsymbol{q})\dot{x} + \boldsymbol{P}_W\boldsymbol{\nu} \tag{3.15}$$

The term $\boldsymbol{P}_W = (\boldsymbol{I} - \boldsymbol{J}_W^\dagger(\boldsymbol{q})\boldsymbol{J}(\boldsymbol{q}))$ projects an arbitrary vector $\boldsymbol{\nu} \in \mathbb{R}^n$ onto the null-space of the manipulator Jacobian matrix $\boldsymbol{J}(\boldsymbol{q})$. Physically, any motion in the null-space is an instantaneous motion of the manipulator joints which causes no movement of the end-effector. Many redundancy resolution objective functions can be developed as potential functions, and $\boldsymbol{\nu}$ might be the gradient of that function. In general, vector $\boldsymbol{\nu}$ is conveniently chosen to achieve further objectives, such as obstacle avoidance or to prevent kinematic singularities.

Analogously to previous calculations, it is possible to show that equation (3.14) is the solution of the more general formulation of (3.5):

$$\begin{aligned} \min_{\boldsymbol{q}(t)} g(\dot{\boldsymbol{q}}) &= \frac{1}{2}\ (\dot{\boldsymbol{q}} - \boldsymbol{\nu})^\intercal \boldsymbol{W}(\dot{\boldsymbol{q}} - \boldsymbol{\nu}) \\ s.t. \quad \dot{\boldsymbol{x}} &= \boldsymbol{J}(\boldsymbol{q})\ \dot{\boldsymbol{q}} \end{aligned} \tag{3.16}$$

It is significant to note that the particular (3.13) and generalized (3.14) solutions of the optimization problem bring to a *local minimization* of the norm of $\dot{\boldsymbol{q}}$, because they do not take into account the time evolution of $\boldsymbol{q}$ along the task space trajectory. A locally-optimal solution may not be suitable for some kinds of tasks: minimizing the norm of the joint velocities vector at a given configuration does not guarantee its minimal value along the whole path.

3.4.2 Globally-Optimal Inverse Kinematics

Another approach for the resolution of redundant manipulator inverse kinematics is through global optimization of the cost function [22, 10, 23, 9]. Such a procedure exploits the redundancy of the robot optimizing an integral performance index. Additional constraints assures the fulfillment of forward kinematics as well.
The problem can be summarized in the following way:

$$\begin{aligned} \min_{\boldsymbol{q}(t)} G &= \int_{t_0}^{t_1} g(t, \boldsymbol{q}, \dot{\boldsymbol{q}}) dt \\ s.t. \quad & \boldsymbol{x} = \boldsymbol{f}(\boldsymbol{q}) \end{aligned} \tag{3.17}$$

where G is the objective function, t_0 and t_1 are, respectively, the initial and final values of the time interval under observation, and $g(t, \boldsymbol{q}, \dot{\boldsymbol{q}})$ is a generic cost function to be minimized. In (3.17), constraints are represented by algebraic equations, so it can be solved by exploiting the *Euler-Lagrange equation*.

A different formulation of (3.17) can be provided, if forward kinematics (3.2), differential kinematics (3.4), and equation (3.14) are considered, supposing $\boldsymbol{W}$ equal to the identity matrix $\boldsymbol{I}$ and substituting $\boldsymbol{\nu} = \boldsymbol{u}$:

$$\begin{aligned} \min_{\boldsymbol{q}(t)} G &= \int_{t_0}^{t_1} g(t, \boldsymbol{q}, \dot{\boldsymbol{q}}) dt \\ s.t. \quad & \dot{\boldsymbol{q}} = \boldsymbol{J}^{\dagger}(\boldsymbol{q}) \dot{\boldsymbol{x}} + \boldsymbol{P}_W \boldsymbol{u} \end{aligned} \tag{3.18}$$

The reason behind this choice is justified by the fact that $g(t, \boldsymbol{q}, \dot{\boldsymbol{q}})$ is a generic cost function, so any possible pseudoinverse of $\boldsymbol{J}$ can be used, as well as a non-weighted pseudoinverse given by setting the weight matrix $\boldsymbol{W} = \boldsymbol{I}$.

One remarkable difference between the previous systems is the constraints formulation: system (3.17) is composed by algebraic equations, while system (3.18) is constructed with differential equations. It is interesting to notice how constraints in (3.18) can be read as the equations composing a non-linear time-varying dynamical system, where $\boldsymbol{q}$ is the state vector and $\boldsymbol{u}$ is the input (or control) vector.

Keeping in mind that the selection of vector $\boldsymbol{u}$ is arbitrary, it is possible to use this quantity as decision variable for the minimization of performance index G:

$$\begin{aligned} \min_{\boldsymbol{u}(t)} G &= \int_{t_0}^{t_1} g(t, \boldsymbol{q}, \dot{\boldsymbol{q}}) dt \\ s.t. \quad & \dot{\boldsymbol{q}} = \boldsymbol{J}^{\dagger}(\boldsymbol{q}) \dot{\boldsymbol{x}} + \boldsymbol{P}_W \boldsymbol{u} \end{aligned} \tag{3.19}$$

This latter case can be worked out with the help of *Pontryagin's maximum principle*.

Besides the different formalization of (3.17) and (3.19), they conceptualize the same optimization problem: the former involves the minimization of performance index G with respect to all the possible joint space trajectories $\boldsymbol{q}$, while the latter minimizes G about all the possible velocity null-space trajectories $\boldsymbol{u}$.

3.5 Redundancy Resolution with Euler-Lagrange Equation

The Euler-Lagrange equation is a second-order partial differential equation whose solutions are the functions that make a given functional stationary. Because a differentiable functional is stationary at its local maxima and minima, the Euler–Lagrange equation is useful for solving optimization problems in which, given some functional, one seeks the function minimizing or maximizing it.

As long as system (3.17) is concerned, it is worth remarking that it represents an n-DOM redundant manipulator inverse kinematic problem, subject to algebraic equality constraints along a prescribed path specified by the forward kinematics $\boldsymbol{f}$, with the goal to minimize some integral functional G for all the possible joint space trajectories $\boldsymbol{q}$.

3.5.1 Euler-Lagrange Equation

In a similar way to what has already been done for locally-optimal inverse kinematics resolution procedure, a Lagrangian function is introduced:

$$\mathcal{L}(t, \boldsymbol{\lambda}, \boldsymbol{q}, \dot{\boldsymbol{q}}) = g(t, \boldsymbol{q}, \dot{\boldsymbol{q}}) + \boldsymbol{\lambda}^{\intercal}(\boldsymbol{x} - \boldsymbol{f}(\boldsymbol{q})) \tag{3.20}$$

Thanks to $\boldsymbol{\lambda} \in \mathbb{R}^m$, the Lagrange multipliers vector, it is possible to extend the cost function g comprising the forward kinematic constraint. Therefore, by calling G^* the *augmented objective function*, the problem assumes the compact form:

$$\min_{\boldsymbol{q}(t)} G^* = \int_{t_0}^{t_1} \mathcal{L}(t, \boldsymbol{\lambda}, \boldsymbol{q}, \dot{\boldsymbol{q}}) dt \tag{3.21}$$

and Euler-Lagrange first-order necessary condition can be applied:

$$\mathcal{L}_{\boldsymbol{q}}(t, \boldsymbol{\lambda}, \boldsymbol{q}, \dot{\boldsymbol{q}}) - \frac{d}{dt}\mathcal{L}_{\dot{\boldsymbol{q}}}(t, \boldsymbol{\lambda}, \boldsymbol{q}, \dot{\boldsymbol{q}}) = 0 \tag{3.22}$$

in which $\mathcal{L}_{\boldsymbol{q}}(t, \boldsymbol{\lambda}, \boldsymbol{q}, \dot{\boldsymbol{q}})$ and $\mathcal{L}_{\dot{\boldsymbol{q}}}(t, \boldsymbol{\lambda}, \boldsymbol{q}, \dot{\boldsymbol{q}})$ are the partial derivatives of $\mathcal{L}(t, \boldsymbol{\lambda}, \boldsymbol{q}, \dot{\boldsymbol{q}})$ with respect to $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$.

3.5.2 Analysis with Euler-Lagrange Equation

In order to let the non-constrained optimization problem (3.21) be equivalent to the constrained optimization problem (3.17), forward kinematic constraints have to be considered besides equation (3.22). Hence, the optimal solution has to satisfy the set of differential algebraic equations (DAE):

$$\begin{aligned} \mathcal{L}_{\boldsymbol{q}}(t, \boldsymbol{\lambda}, \boldsymbol{q}, \dot{\boldsymbol{q}}) - \frac{d}{dt}\mathcal{L}_{\dot{\boldsymbol{q}}}(t, \boldsymbol{\lambda}, \boldsymbol{q}, \dot{\boldsymbol{q}}) &= 0 \\ \boldsymbol{x}(t) &= \boldsymbol{f}(\boldsymbol{q}(t)) \end{aligned} \tag{3.23}$$

This set of DAEs is composed of $n+m$ equations (characterized by (3.22) and (3.2), respectively) in $n+m$ unknowns (represented by $\boldsymbol{q} \in \mathbb{R}^n$ and $\boldsymbol{\lambda} \in \mathbb{R}^m$). Moreover, this system of equations can be reconducted to a set of only algebraic equations. Taking into account equation (3.22), applying definition (3.20) and simplifying the results, the following equation is obtained:

$$\frac{\partial g}{\partial \boldsymbol{q}} - \boldsymbol{\lambda}^\intercal \frac{\partial \boldsymbol{f}(\boldsymbol{q})}{\partial \boldsymbol{q}} - \frac{d}{dt}\left(\frac{\partial g}{\partial \dot{\boldsymbol{q}}}\right) = 0 \tag{3.24}$$

Bearing in mind the definition of manipulator analytic Jacobian and transposing both members of the previous equation, it follows that:

$$\left(\frac{\partial g}{\partial \boldsymbol{q}}\right)^\intercal - \boldsymbol{J}^\intercal \boldsymbol{\lambda} - \frac{d}{dt}\left(\frac{\partial g}{\partial \dot{\boldsymbol{q}}}\right)^\intercal = 0 \tag{3.25}$$

Eventually, setting:

$$\begin{aligned} g_{\boldsymbol{q}} &= \left(\frac{\partial g}{\partial \boldsymbol{q}}\right)^\intercal \\ g_{\dot{\boldsymbol{q}}} &= \left(\frac{\partial g}{\partial \dot{\boldsymbol{q}}}\right)^\intercal \end{aligned} \tag{3.26}$$

equation (3.25) becomes:

$$\frac{dg_{\dot{\boldsymbol{q}}}}{dt} - g_{\boldsymbol{q}} + \boldsymbol{J}^\intercal \boldsymbol{\lambda} = 0 \tag{3.27}$$

At this point, adopting as cost function a simple quadratic law, e.g., the weighted norm of the joint velocities vector $\dot{\boldsymbol{q}}$:

$$g(t, \boldsymbol{q}, \dot{\boldsymbol{q}}) = \frac{1}{2}\dot{\boldsymbol{q}}^\intercal \boldsymbol{W} \dot{\boldsymbol{q}} \tag{3.28}$$

equations (3.26) turn into:

$$\begin{aligned} g_{\boldsymbol{q}} &= 0 \\ g_{\dot{\boldsymbol{q}}} &= \boldsymbol{W}\dot{\boldsymbol{q}} \end{aligned} \tag{3.29}$$

and, thus, equation (3.27) transforms into:

$$\boldsymbol{W}\ddot{\boldsymbol{q}} + \boldsymbol{J}^\intercal \boldsymbol{\lambda} = 0 \tag{3.30}$$

Assuming that matrix $\boldsymbol{W}$ is invertible, and by performing a pre-multiplication of $\boldsymbol{W}^{-1}$ in the previous expression, equation (3.30) develops into:

$$\ddot{\boldsymbol{q}} + \boldsymbol{W}^{-1}\boldsymbol{J}^\intercal \boldsymbol{\lambda} = 0 \tag{3.31}$$

Then, recalling the definition of pseudo-inverse $\boldsymbol{J}_W^\dagger = \boldsymbol{W}^{-1}\boldsymbol{J}^\intercal \left(\boldsymbol{J}\ \boldsymbol{W}^{-1}\boldsymbol{J}^\intercal\right)^{-1}$ and the definition of weighted orthogonal projector $\boldsymbol{P}_W$ onto the null-space of $\boldsymbol{J}(\boldsymbol{q})$:

$$\boldsymbol{P}_W = \left(\boldsymbol{I} - \boldsymbol{J}_W^\dagger \boldsymbol{J}\right) \tag{3.32}$$

by multiplying equation (3.31) by this quantity, it follows that:

$$\boldsymbol{P}_W \ddot{\boldsymbol{q}} + \boldsymbol{P}_W \boldsymbol{W}^{-1}\boldsymbol{J}^\intercal \boldsymbol{\lambda} = 0 \tag{3.33}$$

In turn, after some simple computations, equation (3.33) comes to be:

$$\boldsymbol{P}_W \ddot{\boldsymbol{q}} + \left(\boldsymbol{W}^{-1} \boldsymbol{J}^\intercal - \boldsymbol{J}_W^\dagger \ \boldsymbol{J} \ \boldsymbol{W}^{-1} \boldsymbol{J}^\intercal \right) \boldsymbol{\lambda} = 0 \tag{3.34}$$

which can be further spread into:

$$\boldsymbol{P}_W \ddot{\boldsymbol{q}} + \left(\boldsymbol{W}^{-1} \boldsymbol{J}^\intercal - \boldsymbol{W}^{-1} \boldsymbol{J}^\intercal \left(\boldsymbol{J} \ \boldsymbol{W}^{-1} \boldsymbol{J}^\intercal \right)^{-1} \boldsymbol{J} \ \boldsymbol{W}^{-1} \boldsymbol{J}^\intercal \right) \boldsymbol{\lambda} = 0 \tag{3.35}$$

that, eventually, takes a simpler form:

$$\boldsymbol{P}_W \ddot{\boldsymbol{q}} = 0 \tag{3.36}$$

In going from (3.31) to (3.36) it is assumed that $\boldsymbol{J}$ has full rank and its inverse $\boldsymbol{J}_W^\dagger = \boldsymbol{W}^{-1} \boldsymbol{J}^\intercal (\boldsymbol{J} \ \boldsymbol{W}^{-1} \boldsymbol{J}^\intercal)^{-1}$ exists. At kinematic singularities, this assumption is false. In practice, optimal joint space trajectories may or may not contain singular configurations, so they cannot be completely ignored.

Returning to system (3.23), for what concerns the manipulator forward kinematics, it is possible to work out a second-order form by deriving twice with respect to time:

$$\ddot{\boldsymbol{x}} = \dot{\boldsymbol{J}}(\boldsymbol{q}, \dot{\boldsymbol{q}}) \dot{\boldsymbol{q}} + \boldsymbol{J}(\boldsymbol{q}) \ddot{\boldsymbol{q}} \tag{3.37}$$

In analogy to the first-order inverse kinematics, a possible solution to the kinematic inversion of this form is given by:

$$\ddot{\boldsymbol{q}} = \boldsymbol{J}_W^\dagger \left(\ddot{\boldsymbol{x}} - \dot{\boldsymbol{J}} \dot{\boldsymbol{q}} \right) + \boldsymbol{P}_W \boldsymbol{\alpha} \tag{3.38}$$

in which $\boldsymbol{\alpha}$ is an acceleration vector whose projection onto the null-space of the Jacobian does not provide any contribution to the motion of the end-effector.
Here again, pre-multiplying $\boldsymbol{P}_W$ to both sides of (3.38):

$$\boldsymbol{P}_W \ddot{\boldsymbol{q}} = \boldsymbol{P}_W \boldsymbol{J}_W^\dagger \left(\ddot{\boldsymbol{x}} - \dot{\boldsymbol{J}} \dot{\boldsymbol{q}} \right) + \boldsymbol{P}_W \boldsymbol{P}_W \boldsymbol{\alpha} \tag{3.39}$$

and by exploiting the following orthogonal projector properties:

$$\begin{aligned} \boldsymbol{P}_W \boldsymbol{J}_W^\dagger &= \left(\boldsymbol{I} - \boldsymbol{J}_W^\dagger \boldsymbol{J} \right) \boldsymbol{J}_W^\dagger = 0 \\ \boldsymbol{P}_W \boldsymbol{P}_W &= \left(\boldsymbol{I} - \boldsymbol{J}_W^\dagger \boldsymbol{J} \right) \left(\boldsymbol{I} - \boldsymbol{J}_W^\dagger \boldsymbol{J} \right) = \left(\boldsymbol{I} - \boldsymbol{J}_W^\dagger \boldsymbol{J} \right) = \boldsymbol{P}_W \end{aligned} \tag{3.40}$$

it results that:

$$\boldsymbol{P}_W \ddot{\boldsymbol{q}} = \boldsymbol{P}_W \boldsymbol{\alpha} \tag{3.41}$$

which can be further simplified by taking advantage of equation (3.36):

$$\boldsymbol{P}_W \boldsymbol{\alpha} = 0 \tag{3.42}$$

One considerable remark is that the previous outcomes are specifically retrieved for g in (3.28), i.e., for a quadratic cost function. Then, gathering results (3.38) and (3.42), the system of equations evolves into:

$$\ddot{\boldsymbol{q}} = \boldsymbol{J}_W^{\dagger} \left(\ddot{\boldsymbol{x}} - \dot{\boldsymbol{J}} \dot{\boldsymbol{q}} \right) \tag{3.43}$$

In order to achieve (3.43), both Euler-Lagrange and forward kinematics expressions have been applied. As a result, it is completely equivalent to the starting system of DAEs (3.23). Moreover, (3.43) is a system consisting of n second-order differential equations, which can be converted into $2n$ first-order differential equations.

In addition to this, redefining the joint positions vector $\boldsymbol{\xi} = \boldsymbol{q}$ and joint velocities vector $\boldsymbol{\zeta} = \dot{\boldsymbol{\xi}} = \dot{\boldsymbol{q}}$, (3.43) changes into:

$$\begin{aligned} \dot{\boldsymbol{\xi}} &= \boldsymbol{\zeta} \\ \dot{\boldsymbol{\zeta}} &= \boldsymbol{J}_W^{\dagger} \left(\ddot{\boldsymbol{x}} - \dot{\boldsymbol{J}} \boldsymbol{\zeta} \right) \end{aligned} \tag{3.44}$$

It is worth pointing out that (3.44) is a set of non-linear differential equations. For this reason, a unique solution to (3.44) is not granted, such as in the linear case, and multiple solutions may appear, even for fixed boundary condition values.

3.5.3 Euler-Lagrange Boundary Conditions

The resolution of a system made-up of $2n$ equations requires as many boundary conditions. The optimality of the solution to problem (3.17) strictly depends on the choice of these conditions. As mentioned in Appendix A , calculus of variations imposes "natural" boundary conditions for the computation of the optimal solution. Moreover, in some circumstances, "forced" boundary conditions are utilized, providing sub-optimal solutions to the optimization problem: although they guarantee the fulfillment of the forward kinematics, higher values of performance index are achieved. It is worth noting that the fulfillment of natural boundary conditions is only a necessary condition to the optimality of the solution, since the integration of (3.44) with these conditions may lead to a sub-optimal solution as well.

The following discussion presents some study cases for different Euler-Lagrange boundary conditions, for which mathematical details can be found in Appendix A.

Case 1: Free Endpoints

The first case studied is the one with the minimum number of imposed conditions. In this situation, the only requirement on the joint position values is that, at times t_0 and t_1, they result in the end-effector position specified by the kinematic constraints in (3.17):

$$\boldsymbol{x}(t_i) = \boldsymbol{f}\left(\boldsymbol{q}(t_i)\right), \quad t_i = \{t_0, t_1\} \tag{3.45}$$

which can be put in the implicit form:

$$S_k(\boldsymbol{q}, t_i) = 0, \quad k = 1, \ldots, m, \quad t_i = \{t_0, t_1\} \tag{3.46}$$

This constraint formulation is legitimate considering the vectorial nature of the manipulator kinematic equations. It may also be convenient to look at equations (3.46) from a geometric point of view. These non-linear equations represent m hypersurfaces in the n-dimensional joint space.

Since none of the joint positions are prescribed at these values, then $\delta\boldsymbol{q}(t_i)$ no longer vanishes in equation (A.15). In order to satisfy equation (A.15), the following condition should be met:

$$\left(\frac{\partial g}{\partial \dot{\boldsymbol{q}}(t_i)}\right)^{\intercal} \delta\boldsymbol{q}(t_i) = 0, \quad t_i = \{t_0, t_1\} \tag{3.47}$$

Clearly, the joint position values $\boldsymbol{q}(t_i)$ should satisfy both (3.46) and (3.47) at any given time instant t_i: in (3.46) joints $\boldsymbol{q}(t_i)$ are represented by points onto the hypersurfaces S_k, while (3.47) implies that variations $\delta\boldsymbol{q}(t_i)$ must be tangent to those hypersurfaces. Recalling that the gradient of a surface is a vector orthogonal to it in each point belonging to that surface, it follows that $\delta\boldsymbol{q}(t_i)$ has to be orthogonal to all the gradients of each hypersurface S_k, and, moreover, it will be orthogonal to a linear combination of these gradients:

$$\frac{\partial g}{\partial \dot{\boldsymbol{q}}(t_i)} = \left(\frac{\partial S_k(\boldsymbol{q}, t_i)}{\partial \boldsymbol{q}(t_i)}\right)^{\intercal} \boldsymbol{p} \tag{3.48}$$

where $\boldsymbol{p} \in \mathbb{R}^m$ is a constant vector and the term in brackets is the Jacobian matrix of the manipulator at boundary time t_i. Equation (3.48) can be, thus, simplified in:

$$\frac{\partial g}{\partial \dot{\boldsymbol{q}}(t_i)} = \boldsymbol{J}^{\intercal}(t_i)\boldsymbol{p} \tag{3.49}$$

Equation (3.49) can be further developed by exploiting (3.28):

$$\boldsymbol{W}\dot{\boldsymbol{q}}(t_i) = \boldsymbol{J}^{\intercal}(t_i)\boldsymbol{p} \tag{3.50}$$

otherwise written as:

$$\dot{\boldsymbol{q}}(t_i) = \boldsymbol{W}^{-1}\boldsymbol{J}^{\intercal}(t_i)\boldsymbol{p} \tag{3.51}$$

In addition, by use of the first-order differential kinematics (3.4) of the manipulator:

$$\dot{\boldsymbol{x}}(t_i) = \boldsymbol{J}^{\intercal}(t_i)\dot{\boldsymbol{q}}(t_i) \tag{3.52}$$

and substituting equation (3.51) into (3.52):

$$\dot{\boldsymbol{x}}(t_i) = \boldsymbol{J}(t_i)\boldsymbol{W}^{-1}\boldsymbol{J}^{\intercal}(t_i)\boldsymbol{p} \tag{3.53}$$

an expression of $\boldsymbol{p}$ is, so, obtained:

$$\boldsymbol{p} = \left(\boldsymbol{J}(t_i)\boldsymbol{W}^{-1}\boldsymbol{J}^{\intercal}(t_i)\right)^{-1} \dot{\boldsymbol{x}}(t_i) \tag{3.54}$$

This newly found expression can be replaced into (3.51) to generate:

$$\dot{\boldsymbol{q}}(t_i) = \boldsymbol{W}^{-1}\boldsymbol{J}^{\intercal}(t_i)\left(\boldsymbol{J}(t_i)\boldsymbol{W}^{-1}\boldsymbol{J}^{\intercal}(t_i)\right)^{-1}\dot{\boldsymbol{x}}(t_i) \tag{3.55}$$

which is, finally, reduced to:

$$\dot{\boldsymbol{q}}(t_i) = \boldsymbol{J}_W^{\dagger}\dot{\boldsymbol{x}}(t_i), \quad t_i = \{t_0, t_1\} \tag{3.56}$$

Equation (3.56) is immediately recognized to be the least square solution of joint velocities for the specified boundary velocities $\dot{\boldsymbol{q}}(t_0)$ and $\dot{\boldsymbol{q}}(t_1)$. This equation is referred to as the *natural boundary condition* (or, free boundary condition) and should be satisfied at both t_0 and t_1.

The set of differential equations (3.44) and the natural boundary condition (3.56) together define a system of second-order differential equations with split boundary conditions (i.e., at t_0 and t_1) on velocities. The solution to this *boundary value problem* (BVP) guarantees a minimum value for the objective function (3.18) for a given trajectory.

Case 2: Imposed Initial Joint Position Values

When the joint positions are specified at the beginning of the trajectory, then $\delta\boldsymbol{q}(t_0)$ is zero and the natural boundary condition exists only at the final time t_1. Therefore, the boundary conditions corresponding to the set of differential equations (3.44) are:

$$\begin{aligned} \boldsymbol{q}(t_0) &= \boldsymbol{q}_s \\ \dot{\boldsymbol{q}}(t_1) &= \boldsymbol{J}_W^{\dagger}\dot{\boldsymbol{x}}(t_1) \end{aligned} \tag{3.57}$$

It is important to observe that binding the manipulator initial configuration permits to find only a sub-optimal solution: the research of the minimum value of the objective function is not done anymore into the space of all possible joint space trajectories, but by searching into a subspace of it, i.e., into the space of all the possible joint space trajectories that start with a given manipulator initial configuration. As a result, the value of the objective function is expected to be larger than the one in *Case 1*.

Case 3: Imposed Initial Joint Position and Velocity Values

A particular case presents when both joint positions and joint velocities at time t_0 are given. Here, none of the two natural boundary conditions is applied and it becomes an *initial value problem* (IVP) with boundary conditions:

$$\begin{aligned} \boldsymbol{q}(t_0) &= \boldsymbol{q}_s \\ \dot{\boldsymbol{q}}(t_0) &= \boldsymbol{J}_W^{\dagger}\dot{\boldsymbol{x}}(t_0) \end{aligned} \tag{3.58}$$

Once again, a sub-optimal solution is reached. However, since the natural boundary condition at t_1 is ignored, the resulting path is a "weak" minimum. The "strong" minimum solution, which can be seen in *Case 2*, is achieved by a sudden jump from the imposed joint velocities (3.58) to those dictated by the natural boundary conditions (3.56). Such a discontinuity, $\ddot{q} = \infty$, is physically impossible for the manipulator dynamics, so the best strategy is to choose initial joint velocities as close as possible to those determined by the natural boundary conditions.

Case 4: Imposed Joint Position Values at Both Boundaries

When both initial and final joint positions are assigned, no natural boundary condition is defined and the forced boundary conditions are simply split at the initial time and the final time:

$$\begin{aligned} \boldsymbol{q}(t_0) &= \boldsymbol{q}_s \\ \boldsymbol{q}(t_1) &= \boldsymbol{q}_f \end{aligned} \tag{3.59}$$

A conservative solution in the joint space can be reached by specifying the same point for the beginning and the end of the trajectory. This circumstance represents a *periodic boundary value problem* and happens when the task space trajectory is periodic, i.e., $\boldsymbol{x}(t_0) = \boldsymbol{x}(t_1)$. In this particular situation, the problem is subject to the following constraints:

$$\begin{aligned} \boldsymbol{q}(t_0) &= \boldsymbol{q}(t_1) \\ \dot{\boldsymbol{q}}(t_0) &= \dot{\boldsymbol{q}}(t_1) \end{aligned} \tag{3.60}$$

All the previously considered boundary value problems, that is, every study case but number 3, are also referred to as *two-point boundary value problems* (TPBVP). The word *two-point* refers to the fact that the boundary conditions are evaluated at the two interval endpoints t_0 and t_1, unlike for initial value problems where the initial conditions are all evaluated at a single point. Occasionally, problems arise where constraints are also evaluated at other points in $[t_0, t_1]$. In these cases, a *multipoint BVP* arise. A multipoint problem may be converted to a two-point problem by defining separate sets of variables for each subinterval between the points and adding boundary conditions which ensure continuity of the variables across the whole interval [24].

3.5.4 Reduced-Order Form

As previously pointed out, the resolution of the inverse kinematic problem in a globally-optimal form means to solve a $2n$ first-order differential equations system. In this sense, a simpler set of equations can be introduced to decrease the complexity of the problem. Before starting to reduce the degree of the system, it is appropriate to present the concept of *redundancy parameter*.

This quantity consists of r elements, where r is the degree of redundancy of the robot (3.1), and is such that, for each point of the end-effector path, the set of all possible inverse kinematics solutions becomes finite [25].

The following considerations concern system (3.38), which is the direct derivation of a quadratic performance index, but, it can be applied to a wide range of systems, being (3.38) the solution to most of the performance indices found in the literature.

Given vector $\boldsymbol{\gamma}$ of null-space velocities, $\boldsymbol{N}$ a null-space basis of the manipulator Jacobian matrix and $\boldsymbol{N}_W = \boldsymbol{W}\boldsymbol{N}$ a weighted null-space basis, the following relations hold:

$$\begin{aligned} \boldsymbol{\gamma} &= \boldsymbol{N}_W^\intercal \dot{\boldsymbol{q}} \\ \dot{\boldsymbol{\gamma}} &= \dot{\boldsymbol{N}}_W^\intercal \dot{\boldsymbol{q}} + \boldsymbol{N}_W^\intercal \ddot{\boldsymbol{q}} \end{aligned} \tag{3.61}$$

Recalling the generic first-order (3.15) and second-order (3.38) solutions:

$$\begin{aligned} \dot{\boldsymbol{q}} &= \boldsymbol{J}_W^\dagger \dot{\boldsymbol{x}} + \boldsymbol{P}_W \boldsymbol{\nu} \\ \ddot{\boldsymbol{q}} &= \boldsymbol{J}_W^\dagger \left(\ddot{\boldsymbol{x}} - \dot{\boldsymbol{J}} \dot{\boldsymbol{q}} \right) + \boldsymbol{P}_W \boldsymbol{\alpha} \end{aligned} \tag{3.62}$$

if multiplied by $\boldsymbol{N}_W^\intercal$, an equivalent formulation is found:

$$\begin{aligned} \boldsymbol{N}_W^\intercal \dot{\boldsymbol{q}} &= \boldsymbol{N}_W^\intercal (\boldsymbol{J}_W^\dagger \dot{\boldsymbol{x}} + \boldsymbol{P}_W \boldsymbol{\nu}) \\ \boldsymbol{N}_W^\intercal \ddot{\boldsymbol{q}} &= \boldsymbol{N}_W^\intercal (\boldsymbol{J}_W^\dagger (\ddot{\boldsymbol{x}} - \dot{\boldsymbol{J}} \dot{\boldsymbol{q}}) + \boldsymbol{P}_W \boldsymbol{\alpha}) \end{aligned} \tag{3.63}$$

Taking into account both $\boldsymbol{N}_W^\intercal \boldsymbol{J}_W^\dagger = 0$ and $\boldsymbol{N}_W^\intercal \boldsymbol{P}_W = \boldsymbol{N}_W^\intercal$, the previous equations become:

$$\begin{aligned} \boldsymbol{N}_W^\intercal \dot{\boldsymbol{q}} &= \boldsymbol{N}_W^\intercal \boldsymbol{\nu} \\ \boldsymbol{N}_W^\intercal \ddot{\boldsymbol{q}} &= \boldsymbol{N}_W^\intercal \boldsymbol{\alpha} \end{aligned} \tag{3.64}$$

Then, folding (3.64) into (3.61), a new expression of $\boldsymbol{\gamma}$ and $\dot{\boldsymbol{\gamma}}$ is obtained:

$$\begin{aligned} \boldsymbol{\gamma} &= \boldsymbol{N}_W^\intercal \boldsymbol{\nu} \\ \dot{\boldsymbol{\gamma}} &= \dot{\boldsymbol{N}}_W^\intercal \dot{\boldsymbol{q}} + \boldsymbol{N}_W^\intercal \boldsymbol{\alpha} \end{aligned} \tag{3.65}$$

The inverse of the first equation in (3.65) is:

$$\boldsymbol{\nu} = {\boldsymbol{N}_W^\intercal}^{-1} \boldsymbol{\gamma} \tag{3.66}$$

which substituted into (3.15), while bearing in mind the definition of weighted orthogonal projector $\boldsymbol{P}_W = \boldsymbol{N}(\boldsymbol{N}_W^\intercal N)^{-1} \boldsymbol{N}_W^\intercal$, gives birth to:

$$\dot{\boldsymbol{q}} = \boldsymbol{J}_W^\dagger \dot{\boldsymbol{x}} + \boldsymbol{N}(\boldsymbol{N}_W^\intercal \boldsymbol{N})^{-1} \boldsymbol{\gamma} \tag{3.67}$$

Eventually, the system composed by (3.67) and the second equation in (3.65) represent a set of $n + r = 2n - m$ first-order ordinary differential equations in $\boldsymbol{q}$ and $\boldsymbol{\gamma}$, and constitutes an equivalent reduced-order form of (3.38).

3.6 Redundancy Resolution with Dynamic Programming

The application of calculus of variations to redundant manipulators, either by means of the Euler-Lagrange equation or Pontryagin's maximum principle, provides an analytic formulation of the inverse kinematic problem in the form of a TPBVP. Two-point boundary value problems are often hard to solve, also with the usage of modern tools. In addition, they are weak at finding Pareto-optimal sets, as the optimization of multiple performance indices at the same time can only be done through the application of weights [26].

As a result, alternatives to calculus of variations have been explored to seek globally-optimal solutions to the inverse kinematic problem. Recently, algorithms based on dynamic programming (DP) are preferred among other options. It has been demonstrated that DP-inspired techniques can easily handle multiple objective functions, ensuring the Pareto-optimality of the solution: they show better computational performance due to a reduced search space size given by the additional constraints imposed through each objective function. Accordingly, although DP is more demanding in terms of memory usage and computational time, when compared to numerical integrations, the aforesaid properties make it much more flexible in meeting the challenges of real applications.

3.6.1 Dynamic Programming

Dynamic programming is both a mathematical optimization method and a computer programming method. In these contexts, it refers to simplifying a complex problem by breaking it down into simpler sub-problems. If sub-problems can be nested recursively within larger problems, then there is a relation between the value of the larger problem and the values of the sub-problems, termed the *Bellman equation*. In computer science, if a problem can be optimally solved by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure. There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub-problems.

Optimal substructure means that the solution to a given optimization problem can be calculated by the combination of optimal solutions to its sub-problems. Such optimal substructures are usually described by means of recursion.

Overlapping sub-problems means that the space of sub-problems must be small, that is, any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems. Dynamic programming takes account of this fact and solves each sub-problem only once.

3.6.2 Analysis with Dynamic Programming

Although a continuous-time formulation of the dynamic programming problem can be given, it is necessary to have a discrete-time form so that an algorithm can be easily derived from it. The discussed DP-inspired algorithm is based on Bellman's principle of optimality and its corresponding equation. As already mentioned, the problem has to be discretized so that a number of sub-problems can be extracted from it and solved by applying the algorithm.

Assuming that a task is assigned to a robotic system, and performed within a time interval $[t_0, t_1]$, this can be divided into a number of $N_i = (t_1 - t_0)/\tau$ samples, where τ, the *sampling interval*, is such that the following relation holds $t = i\tau$, with $t \in [t_0, t_1]$ the *discrete time samples* and $i = 0,1,\ldots,N_i$ the *waypoint index*. The following discrete kinematic model of the robot is supposed, with corresponding initial conditions:

$$\boldsymbol{q}(i+1) = \boldsymbol{f}\left(\boldsymbol{q}(i), \boldsymbol{u}(i)\right), \quad \boldsymbol{q}(0) = \boldsymbol{q}_s \tag{3.68}$$

As in previous paragraphs, $\boldsymbol{q}$ is the *state vector* of the system and $\boldsymbol{u}$ is the *input vector* (or *control vector*). The goal is to find the optimal time sequence of $\boldsymbol{u}(i)$ that optimizes a given cost function. The cost function is, generally, dependent on $\boldsymbol{q}(i)$, $\boldsymbol{u}(i)$ and their derivatives.
An example of objective function [27] is the following one:

$$I(0) = \Psi(\boldsymbol{q}(N_i)) + \sum_{j=0}^{N_i-1} l\left(\boldsymbol{q}(j), \dot{\boldsymbol{q}}(j), \boldsymbol{u}(j), \dot{\boldsymbol{u}}(j)\right) \tag{3.69}$$

where $\Psi(\boldsymbol{q}(N_i))$ is the cost at final configuration, and $l\left(\boldsymbol{q}(j), \dot{\boldsymbol{q}}(j), \boldsymbol{u}(j), \dot{\boldsymbol{u}}(j)\right)$ is the local cost. This formulation can be slightly modified by using the Euler approximation for the derivatives. Thus, given:

$$\begin{aligned} \dot{\boldsymbol{q}}(j) &\approx \frac{\boldsymbol{q}(j+1) - \boldsymbol{q}(j)}{\tau} \\ \dot{\boldsymbol{u}}(j) &\approx \frac{\boldsymbol{u}(j+1) - \boldsymbol{u}(j)}{\tau} \end{aligned} \tag{3.70}$$

the new form of the objective function is:

$$I(0) = \Psi(\boldsymbol{q}(N_i)) + \sum_{j=0}^{N_i-1} l\left(\boldsymbol{q}(j), \boldsymbol{q}(j+1), \boldsymbol{u}(j), \boldsymbol{u}(j+1)\right) \tag{3.71}$$

which, at a generic stage i, becomes:

$$I(i) = \Psi(\boldsymbol{q}(N_i)) + \sum_{j=i}^{N_i-1} l\left(\boldsymbol{q}(j), \boldsymbol{q}(j+1), \boldsymbol{u}(j), \boldsymbol{u}(j+1)\right) \tag{3.72}$$

As already stated, in order to proceed with a dynamic programming approach, the problem must be broken into many sub-problems and, then, a solution can be found by means of recursion. Consequently, expression (3.72) is put in a recursive form:

$$\begin{aligned} I(N_i) &= \Psi(\boldsymbol{q}(N_i)) \\ I(i) &= I(i+1) + l\left(\boldsymbol{q}(i), \boldsymbol{q}(i+1), \boldsymbol{u}(i), \boldsymbol{u}(i+1)\right) \end{aligned} \tag{3.73}$$

The last computed cost function will be $I(0)$, which is the original objective function to be optimized. Actually, choosing the cost minimization as optimization criterion, Bellman's principle can be exploited:

$$\begin{aligned} I(N_i) &= \Psi(\boldsymbol{q}(N_i)) \\ I_{opt}(i) &= \min_{\boldsymbol{u}} \left[I(i+1) + l\left(\boldsymbol{q}(i), \boldsymbol{q}(i+1), \boldsymbol{u}(i), \boldsymbol{u}(i+1)\right)\right] \end{aligned} \tag{3.74}$$

In (3.74), $I_{opt}(i)$ is the *optimal return function* and represents the minimum value of the objective function if the process started at stage i. Hence, $I(0)$ coincides with the optimal return function computed from starting time $t = 0$, and, so, it represents the optimized function throughout the entire trajectory.

3.6.3 Redundancy Parametrization

One big difference between the Euler-Lagrange approach and dynamic programming approach is the input vector $\boldsymbol{u}$. This quantity provides a way to control the kinematic system, and this is why it has to be carefully chosen to properly optimize the cost function. Most times, $\boldsymbol{u}$ is restrained to a bounded time-variant domain $\mathcal{A}_i$, and its time derivative $\dot{\boldsymbol{u}}$ as well. To be more precise, $\dot{\boldsymbol{u}}$ is limited to a domain $\mathcal{B}_i(\boldsymbol{u}(i))$ which is, basically, both time-variant and input-variant. Therefore, at each i, the set of admissible values of $\boldsymbol{u}(i)$, from which it is possible to reach $\boldsymbol{u}(i+1)$, is given by the set $\mathcal{C}_i$, intersection between $\mathcal{A}_i$ and the set of values of $\boldsymbol{u}$ that respect the constraint on the derivative. In summary [27]:

$$\begin{aligned} \boldsymbol{u}(i) &\in \mathcal{A}_i \\ \dot{\boldsymbol{u}}(i) &\in \mathcal{B}_i(\boldsymbol{u}(i)) \\ \mathcal{C}_i = \mathcal{A}_i \cap \left\{ \boldsymbol{u}(i) \mid \frac{\boldsymbol{u}(i+1) - \boldsymbol{u}(i)}{\tau} \in \mathcal{B}_i(\boldsymbol{u}(i)) \right\}, &\quad \boldsymbol{u}(i+1) \in \mathcal{A}_{i+1} \end{aligned} \tag{3.75}$$

An important matter, when dealing with the determination of an optimal joint space trajectory, is to decide which quantity the input vector $\boldsymbol{u}$ corresponds to. A common choice is to adopt as input vector a subset of the joint position variables, and, from these, the remaining joint positions are computed at each i. An alternative to *joint selection* is to parametrize the redundancy with respect to some *joint combination* [26]. Frequently, joint selection is preferred as primary option, so both terms $\boldsymbol{u}$ and $\boldsymbol{q_j}$ (the vector of joints chosen as redundancy parameters) are often used interchangeably.

3.6.4 Minimum Number of Equations

Analyses with dynamic programming typically require an enormous amount of calculations. The resolution of a large number of equations involves long computational times, so the amount of equations to deal with has to be kept as low as possible. Thanks to redundancy parametrization, a further minimization in the number of manipulator kinematic equations [25] is attained.
The optimization problem (3.21):

$$\min_{\boldsymbol{q}(t)} G^* = \int_{t_0}^{t_1} \mathcal{L}(t, \boldsymbol{\lambda}, \boldsymbol{q}, \dot{\boldsymbol{q}}) dt$$

with performance index (3.28):

$$g(t, \boldsymbol{q}, \dot{\boldsymbol{q}}) = \frac{1}{2} \dot{\boldsymbol{q}}^\intercal \boldsymbol{W} \dot{\boldsymbol{q}}$$

once applied the Euler-Lagrange equations, yields to solution (3.43):

$$\ddot{\boldsymbol{q}} = \boldsymbol{J}_W^\dagger \left(\ddot{\boldsymbol{x}} - \dot{\boldsymbol{J}} \dot{\boldsymbol{q}} \right)$$

For the sake of simplicity, the adopted performance index is the square norm of the joint velocities vector (3.28). However, the concept can be generalized to any performance index that can be managed with the Euler-Lagrange method and leads to n second-order differential equations.
To the purpose of simplifying the following notation, two assumptions are made:

$$\begin{gathered} \boldsymbol{W} = \boldsymbol{I} \\ r = n - m = 1 \end{gathered} \tag{3.76}$$

The first of (3.76) gives equal weight to each component of the square norm, and permits to put (3.43) in a simpler form, which, without loss of generality, can be expressed in function of $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$:

$$\ddot{\boldsymbol{q}} = \boldsymbol{J}^\dagger(\boldsymbol{q}) \left(\ddot{\boldsymbol{x}} - \dot{\boldsymbol{J}}(\boldsymbol{q}, \dot{\boldsymbol{q}}) \dot{\boldsymbol{q}} \right) \tag{3.77}$$

The second equation in (3.76) explicitly claims one degree of redundancy, which means that one of the manipulator's joints, q_j, is chosen as redundancy parameter and it is parametrized. Therefore, the j-th equation picked from the system (3.77) is written as:

$$\ddot{q}_j = \boldsymbol{\eta}_{j,*}^\dagger(\boldsymbol{q}) \left(\ddot{\boldsymbol{x}} - \dot{\boldsymbol{J}}(\boldsymbol{q}, \dot{\boldsymbol{q}}) \dot{\boldsymbol{q}} \right) \tag{3.78}$$

where $\boldsymbol{\eta}_{j,*}^\dagger(\boldsymbol{q})$ refers to the j-th row of $\boldsymbol{J}^\dagger(\boldsymbol{q})$. As the terms on the right-hand side of the previous equation depend on $\boldsymbol{q}$, it is preferable to separate q_j from the other joints.

To this end, the position vector $\boldsymbol{q}_s \in \mathbb{R}^m$ of the kinematic substructure, i.e., the kinematic chain obtained by removing q_j, is defined:

$$\boldsymbol{q}_s = [q_1 \dots q_{j-1}\ q_{j+1} \dots q_n]^\intercal \tag{3.79}$$

and (3.78) translates into:

$$\ddot{q}_j = \boldsymbol{\eta}^\dagger_{j,*}(q_j, \boldsymbol{q}_s) \left(\ddot{\boldsymbol{x}} - \dot{\boldsymbol{J}}(q_j, \boldsymbol{q}_s, \dot{q}_j, \dot{\boldsymbol{q}}_s)\dot{\boldsymbol{q}}\right) \tag{3.80}$$

At this point, it is convenient to recall the manipulator forward kinematic function (3.2) highlighting q_j and $\boldsymbol{q}_s$:

$$\boldsymbol{x} = \boldsymbol{f}(q_j, \boldsymbol{q}_s) \tag{3.81}$$

Assuming that q_j is driven by the differential equation (3.80), the inverse of (3.81) in $\boldsymbol{q}_s$ returns a finite set of solutions:

$$\boldsymbol{q}_s = \boldsymbol{f}^{-1}(\boldsymbol{x}, q_j) \tag{3.82}$$

Furthermore, choosing joint selection as redundancy parametrization criterion, and parametrizing the input vector $\boldsymbol{u} = q_j$ with discrete values, the manipulator forward kinematics function simplifies into $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{q}_s)$, which suggests that the Jacobian matrix of $\boldsymbol{f}$ is square.

Now, defining $\boldsymbol{\eta}_{*,j}$ as the j-th column of $\boldsymbol{J}$, and $\boldsymbol{J}_s$ as the square matrix obtained from $\boldsymbol{J}$ by removing the j-th column, also called *reduced Jacobian*, the derivative of forward kinematic function $\dot{\boldsymbol{x}} = \boldsymbol{J}(q_j, \boldsymbol{q}_s)\dot{\boldsymbol{q}}$ turns into:

$$\dot{\boldsymbol{x}} = \boldsymbol{\eta}^\dagger_{*,j}(q_j, \boldsymbol{q}_s)\dot{q}_j + \boldsymbol{J}_s(q_j, \boldsymbol{q}_s)\dot{\boldsymbol{q}}_s \tag{3.83}$$

Taking for granted that the kinematic substructure is far from mechanical singularity, again, by inverting the kinematic relation, it is possible to solve for $\boldsymbol{q}_s$:

$$\dot{\boldsymbol{q}}_s = \boldsymbol{J}_s^{-1}(q_j, \boldsymbol{q}_s) \left(\dot{\boldsymbol{x}} - \boldsymbol{\eta}^\dagger_{*,j}(q_j, \boldsymbol{q}_s)\dot{q}_j\right) \tag{3.84}$$

At last, collecting the results for q_j and $\boldsymbol{q}_s$, the following set of equations is achieved:

$$\begin{aligned} \ddot{q}_j &= \boldsymbol{\eta}^\dagger_{j,*}(q_j, \boldsymbol{q}_s) \left(\ddot{\boldsymbol{x}} - \dot{\boldsymbol{J}}(q_j, \boldsymbol{q}_s, \dot{q}_j, \dot{\boldsymbol{q}}_s)\dot{\boldsymbol{q}}\right) \\ \dot{q}_j &= \frac{dq_j}{dt} \\ \dot{\boldsymbol{q}}_s &= \boldsymbol{J}_s^{-1}(q_j, \boldsymbol{q}_s) \left(\dot{\boldsymbol{q}} - \boldsymbol{\eta}^\dagger_{*,j}(q_j, \boldsymbol{q}_s)\dot{q}_j\right) \\ \boldsymbol{q}_s &= \boldsymbol{f}^{-1}(\boldsymbol{x}, q_j) \end{aligned} \tag{3.85}$$

This system is equivalent to (3.78) but composed of only $2r$ differential equations in q_j, being $\boldsymbol{q}_s$ given by the last two of (3.85), once the redundancy parameter is set. As a result, it represents a reduced-order form of the initial group of solutions.

These results on the minimum number of equations are useful only when $\boldsymbol{J}_s$ is full-rank, so it can be inverted. In order to exploit equation (3.84) so to make sure that (3.84) returns a finite number of solutions, singularities for the reduced Jacobian must be detected. In fact, a singular Jacobian is a necessary but not sufficient condition to get an infinite set of solutions, that is, if a singularity is determined for the reduced Jacobian, it may lead to a group of finite solutions. To this purpose, an effective procedure on how to correctly select the redundant joints to be parametrized is illustrated in [25].

3.7 A DP-inspired algorithm

Several DP-inspired algorithms have been suggested in the literature, such as [28, 29, 30] and [27]. The latter introduces a forward algorithm, which permits to broadly explore the joint space of a robot considering the possibility of manipulator reconfiguration (e.g., from elbow-up to elbow-down and vice-versa). This is a crucial feature for achieving the globally-optimal solution. As stated in [25, 26, 27], manipulator reconfiguration happens in the vicinity of a kinematic singularity of the robot. In particular, given a redundant robot and a task space trajectory, the assigned task may be accomplished in multiple ways, that is, there are several feasible joint space trajectories, and some of them may contain kinematic singularities. The possibility to exploit singular configurations of a joint space trajectory allows to obtain the globally-optimal solution, with the lowest value of cost function, not attainable otherwise.

The above-mentioned algorithm investigates the topological features of redundant manipulators kinematics, which for some simple cases (e.g., a 4R manipulator with one redundancy parameter) can give graphical representation of the globally-optimal solution in terms of joint space trajectory.

3.7.1 Grid Representation

Besides a graphical feedback of the inverse kinematics solution, for the building of the algorithm, it is convenient to generate a data structure which includes information relating waypoints and redundancy parameters. Such a structure can be effectively identified as a grid having on one axis the discrete time variable t_i, with $i = 1, \dots, N_i$ the *stage index*, or *waypoint index*, and on the other axes the discretized redundancy parameters labelled as $\boldsymbol{u_j}$, where $\boldsymbol{j} \in \mathbb{N}^r$ is the vector of *redundancy parameters indices*. In addition, parametrizing the redundancy by means of joint selection, it holds that $\boldsymbol{u_j} = \boldsymbol{q_j}$.

For the sake of simplicity, as already done above, it is supposed to work with only one redundancy parameter. Thus, keeping $r = 1$, it brings to $\boldsymbol{j} = j \in \mathbb{N}$ with $j = 1, \dots, N_u$, being N_u the number of *redundancy parameter samples*, from which follows that $\boldsymbol{u_j} = u_j$; consequently, a two-dimensional grid is produced.

Each cell (i, j) of the grid is designed as an object containing various attributes. One of these is the joint position vector computed by solving the manipulator inverse kinematics, taking advantage of (3.85). As a result, each cell takes in a vector:

$$\boldsymbol{q}(i,j) = \boldsymbol{f}^{-1}(\boldsymbol{x}(i), u_j) = [q_1(i,j), q_2(i,j), \ldots, q_n(i,j)]^\intercal \tag{3.86}$$

The reason behind this approach is found in the avoidance of a well-known issue in dynamic optimization, whose name, coined by Richard E. Bellman, is *curse of dimensionality*. The curse of dimensionality refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces that do not occur in low-dimensional settings, such as the three-dimensional physical space of everyday experience. As discussed in [31], it is possible to refer to three curses of dimensionality: the large dimensionality of the state space, of the action space, and the difficulty or impossibility of computing expectations. The former is the most relevant in the present case of study, and to deal with it, as also suggested in other works, a discretization of the state space is realized instead of the input space. Inputs are then selected from a continuous set, allowing the system to evolve from one discrete state to another. This way, a solution to the problem can be found regardless of the Bellman optimality principle, necessary in case of continuous state space set, and the dynamic programming algorithm turns into an optimal path search algorithm [27].

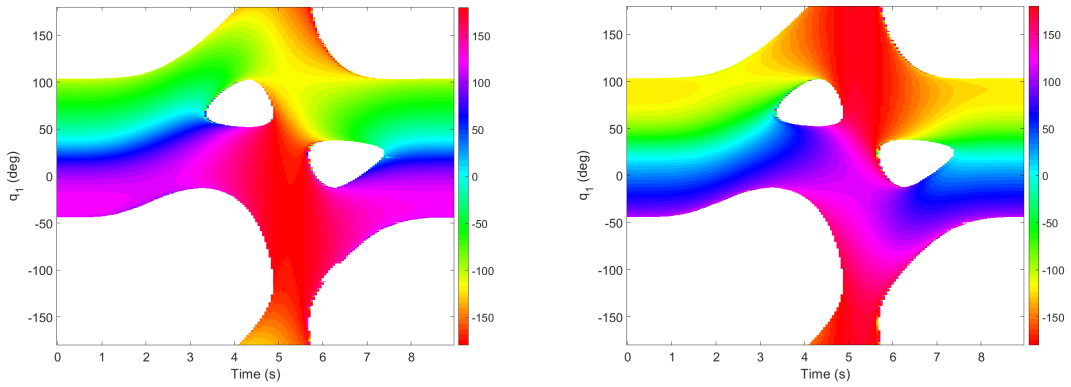
This said, given a robotic manipulator and a specified task space trajectory, equation (3.86) returns a finite set of solutions. These solutions differ one from another and, in order to study their arrangement in the joint space, multiple grids need to be analyzed at the same time. This is the proper way to explore the whole joint space and attain the globally-optimal solution.

From [32], a manipulator is said to have a type 1 geometry if it must pass through a singularity when changing posture. In the remainder of the discussion, given a type 1 manipulator, the term *extended aspect* will be used in reference to subspaces of the joint space into which it is possible to locate one and only one solution to the inverse kinematics problem, when $\boldsymbol{u}$ is given. Under this hypothesis, each grid will represent a different extended aspect of the robotic arm for the assigned task space path. As defined in [33], for a type 1 manipulator, once a redundant parameter is defined, a grid is homogeneous if all the inverse kinematics solutions contained in it belong to the same extended aspect. Although more performant algorithms can be designed if the homogeneity property is satisfied, it is not a necessary condition for an algorithm to find the globally-optimal solution.

Once the redundancy parameters are set, the number of solutions generated from the inverse kinematics problem only depends on the mechanical characteristics of the manipulator. This implies that, for a given manipulator, the maximum *number of grids* N_g is constant and equal to the number of maximum inverse kinematic solutions [34, 4].

Thus, N_g evaluates to 2, 2, 4 and 16 for planar, spherical, regional and spatial manipulators, respectively. It is worth to note that for some specific kinematic structures, as well as for some particular task space trajectories, the actual number of grids produced is less than the maximum theoretical value. Indeed, for most six-axis industrial manipulators, the number of distinct solutions is equal to 8, while for orthogonal manipulators it is 16.

In Figure 3.2, an example of grids generated from a 4-DOF planar manipulator with one redundancy parameter is presented. The considered manipulator [9] is employed to draw a circular task space trajectory in the 2-dimensional task space with constrained orientation, which means a 3-DOF task. The robotic arm is clearly redundant for the assigned task, and q_1 is chosen as redundancy parameter, whose discrete values are located on the ordinate axis. On the abscissa axis, instead, lie the time parametrized waypoints of the trajectory. The colormaps in figures show the values assumed by joint q_2 for each *waypoint*-q_1 pair, while white areas correspond to nodes in the grids for which no solution to the inverse kineamatics is found.



(a) Elbow-left grid

(b) Elbow-right grid

Figure 3.2: Colormaps of homogeneous grids indicating the value of joint q_2.

Besides, further researches on topological aspects of globally-optimal inverse kinematics solutions for redundant manipulators can be found in [26].

3.7.2 Algorithm Formulation

The algorithm presented in [27] is demonstrated to be able to find the globally-optimal solution by transiting from one extended aspect to another. It makes use of a forward implementation of (3.74), as explained in [27], where the calculation advances from $t = t_0$ towards greater values of time steps, ending at $t = t_1$:

$$\begin{aligned} I(0) &= \Psi(\boldsymbol{q}(0)) \\ I_{opt}(i) &= \min_{\boldsymbol{u}} \left[I(i-1) + l\left(\boldsymbol{q}(i), \boldsymbol{q}(i-1), \boldsymbol{u}(i), \boldsymbol{u}(i-1)\right) \right] \end{aligned} \tag{3.87}$$

The discussed formulation applies equation (3.87), using multiple grids and ignoring any assumption on their homogeneity. The algorithm exploits the information stored in each node of the grid to perform comparisons between adjacent clusters. A *cluster* is defined in [29] as the set of inverse kinematics solutions, obtained from (3.86) for fixed values of grid index k and waypoint index i, and variable redundancy parameter index j. Thus, referring to Figure 3.2, a cluster is the vector containing all the inverse kinematics solutions at a given waypoint. Since the generic cluster at current waypoint $\{\boldsymbol{q}(i, j, k),\ j = 1, \ldots, N_u\}$ has $2N_g$ adjacent clusters, namely $\{\boldsymbol{q}(i-1, j, k),\ j = 1, \ldots, N_u,\ k = 1, \ldots, N_g\}$ and $\{\boldsymbol{q}(i+1, j, k),\ j = 1, \ldots, N_u,\ k = 1, \ldots, N_g\}$, the number of comparisons to make at each iteration of the algorithm is quite considerable.

For a given waypoint index i, each cell, or *node*, of cluster at current waypoint $\{\boldsymbol{q}(i, j, k),\ j = 1, \ldots, N_u\}$ is compared to all the nodes (of cardinality N_u) of clusters at next waypoint $\{\boldsymbol{q}(i+1, j, k),\ j = 1, \ldots, N_u,\ k = 1, \ldots, N_g\}$ (of cardinality N_g). Comparisons are made up as far as the solution in the explored node is feasible, otherwise the grid is abandoned. When a grid different from the current one is visited at a node where robot reconfiguration is not possible, the grid is abandoned.

If homogeneous grids can be obtained instead, an optimized algorithm that exploits kinematic constraints and cuts down the number of comparisons performed between adjacent clusters is provided in [33]. To this end, comparisons are made up until the constraints are satisfied and the grid is abandoned when they are not. The group of nodes of the cluster at next waypoint index $i + 1$, visited from every node of the cluster at current waypoint index i, is named *lookup window* [33], and has cardinality N_w due to the reduced comparison number. The present implementation, due to lower values of N_w with respect to N_u, allows reducing the computational complexity [33] of the algorithm, whose original value [29] is $O(N_i N_u^2)$, which represents an approximation of $O(N_i N_u^2 N_g^2)$ for N_g constant. For typical applications, N_w is one to three orders of magnitude lower than N_u, yielding a considerable reduction in the execution time.

Chapter 4

Analysis of Planning Technologies

4.1 Optimal Trajectory Planning

In Chapter 2, it has been showed that the presence of planners is essential for ground stations to control the robot. Figure 2.4 reveals that ground control operations for space robotics activities mainly consist of path planning and trajectory planning, followed by the inverse kinematics problem resolution. This is true in general, but alternative solutions, such as inverse kinematics of task space path followed by joint space trajectory planning, are reasonable as well. When dealing with trajectory planning of robotic systems, employed in the field of space exploration, redundancy resolution may play a significant role: robots used for robotic exploration are autonomous systems that need a high degree of dexterity, necessary to fulfill the assigned task with good efficiency in short time; naturally, the higher the dexterity, the higher the degree of redundancy. In order to exploit the higher degree of motion of the structure, optimal trajectory planners are developed to effectively manage the extra degrees of freedom of the system and receive benefits in terms of performance.

Frequently, the term trajectory planner is associated with both the actual trajectory planner and the subsequent kinematics inversion. This process yields the joint reference values $\boldsymbol{q}_r(t)$ from an assigned geometric path $\boldsymbol{x}_r(\lambda)$. In some planning problems, where, for instance, it might be advantageous to find the shortest trajectory between two configurations, the attainment of *any* valid trajectory between a start and a target configuration may not be of practical interest [35]. In these situations, the main objective is the research of optimal trajectories: trajectories which satisfy some constraints (e.g., connects start and target configurations without collisions) and also optimizes some performance index. Trajectory planners which attempt to optimize performance indices are known as *optimal planners*.

A possible control scheme that integrates an optimal trajectory planner is depicted in Figure 4.1.

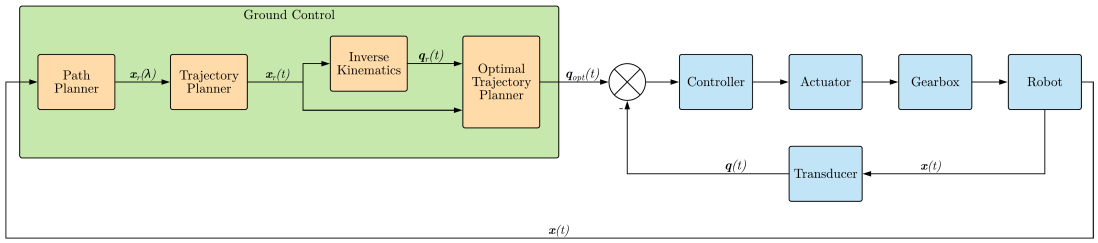


Figure 4.1: Scheme of a closed-loop control system with optimal trajectory planner.

Given a task space trajectory $\boldsymbol{x}_r(t)$, with a prescribed time law and kinematic constraints, and the set of all possible solutions $\boldsymbol{q}_r(t)$ to the kinematic inversion at each waypoint of $\boldsymbol{x}_r(t)$, the scope of an optimal planner is to compute the globally-optimal configuration space trajectory $\boldsymbol{q}_{opt}(t)$ that will be sent to the robotic system. Thus, in the remainder of this dissertation, when talking about *optimal trajectory planners*, it will be referred to the software suite that computes the globally-optimal joint reference values $\boldsymbol{q}_{opt}(t)$ from an assigned path and related kinematic constraints, that is, a task space trajectory with predefined time law $\boldsymbol{x}_r(t)$. This is different from just computing the inverse kinematics of a redundant system, since an optimal trajectory planner is capable of finding the best (optimized) configuration space trajectory taking into account the whole space of inverse kinematics solutions, choosing among them the ones that, along the whole task space trajectory, minimize or maximize some cost function.

In building an optimal trajectory planner, theoretical aspects, such as kinematics inversion and redundancy resolution (discussed in Chapter 3), are important, as well as practical elements, such as the technologies that currently tackle this problem and how to implement them.

4.2 ROS

The first topic to address, when dealing with a software implementation of a trajectory planner, is the selection of the framework onto which the software is built. A common choice among the members of the scientific community is *ROS*.

The *Robot Operating System* (ROS) is an open-source, meta-operating system for robot software development [36]. It includes tools, libraries, and conventions that aim at simplifying the task of creating complex and robust robot behavior [37]. Although ROS is not properly an operating system, it provides services designed for computer clusters, such as low-level device control and package management.

One of the primary advantages of using ROS is its open source characteristic. Open source development usually offers a more flexible technology and quicker innovation with respect to proprietary software. It is, also, more reliable since it typically has thousands of independent programmers testing the software.

The Robot Operating System is not a real-time framework, though it is possible to integrate ROS with real-time code. To this purpose, a suggestion of real-time system implementation has been arranged for ROS 2, the new version of ROS, and other projects, such as ESROCOS [38], have been designed to meet the needs of real-time applications. Nevertheless, this is not a major issue for *offline trajectory planning*, where the whole trajectory is scheduled before the begin of any action.

The principal ROS client libraries (C++, Python, and Lisp) are efficiently fit for Unix-like systems, primarily because of their dependence on large collections of open-source software. For these client libraries, Ubuntu is listed as the preferred Linux distribution, as it was extensively used for its development since the beginning of the ROS project. The present dissertation makes use of the C++ client library, *roscpp*, being the most widely used ROS client library and designed to be the high-performance library for ROS, enabling C++ programmers to quickly interface with ROS concepts, especially those at the Computation Graph level.

The *Computation Graph* [39] is a network of ROS processes, which provide data to the Graph in different ways. The basic Computation Graph concepts of ROS are nodes, Master, Parameter Server, messages, services, topics, and bags.

Node

A ROS node is a process that performs computation [40]. A robot control system usually comprises many nodes. Nodes are combined together into a graph and communicate with one another using topics, services, and the Parameter Server.

Master

The role of the Master is to enable individual ROS nodes to locate one another [41]. It tracks publishers and subscribers to topics as well as services. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services. The Master also provides the Parameter Server.

Parameter Server

A parameter server is a shared dictionary that allows data to be stored in a central location [42]. Nodes use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static data structures such as setup parameters. Since it is part of the Master, the parameter server is globally viewable; this way tools can easily inspect information about the system and modify them if necessary.

Message

Nodes communicate with each other by publishing messages to topics [43]. A message is a simple data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as well as arrays of primitive types. Nodes can also exchange a request and response message as part of a ROS service call.

Service

Request and reply are done via a service, which is defined by a pair of messages: one for the request and one for the reply [44]. A ROS node offers a service under a string name, while a client calls the service by sending the request message and awaiting the reply. This concept will be recalled later on for the development of a generalized redundancy resolution service.

Topic

Topics are named buses over which nodes exchange messages [45]. Topics have a publish/subscribe semantics. This way, nodes that are interested in data can subscribe to the relevant topic, while nodes that generate data can publish to the relevant topic. However, topics are intended for unidirectional streaming communication: nodes that need to perform procedure calls, i.e., to receive a response to a request, should use services instead.

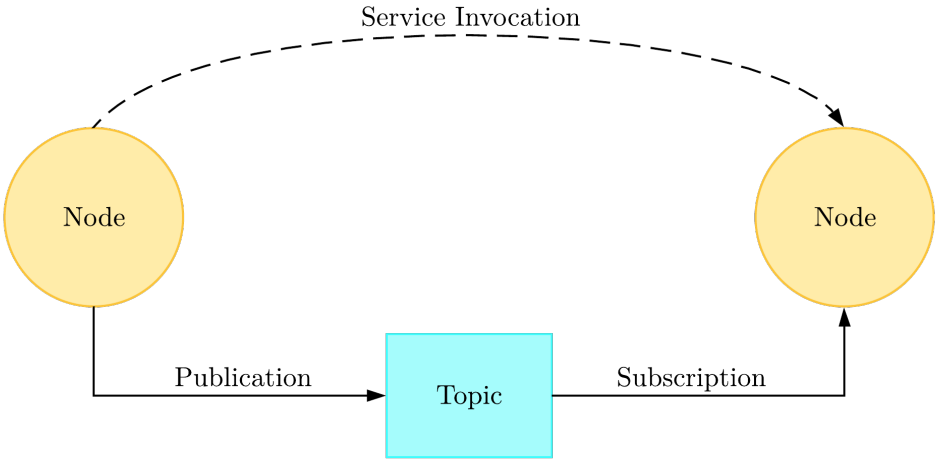


Figure 4.2: Two nodes communicating over a topic during a service invocation.

4.3 Moveit!

Inside ROS, *MoveIt!* is the package that incorporates the latest advances in motion planning, manipulation, and kinematics [46]. Motion planners take care of both the trajectory planning and the execution of the robot motion. In MoveIt!, motion planners are loaded using a plugin infrastructure, allowing the package to communicate at runtime with different planners from multiple libraries.

Motion planning becomes an easy task when using the *MoveGroupInterface* class. MoveGroupInterface is the simplest user interface in MoveIt!, communicating over the more general *MoveGroup* node through ROS concepts, such as services and messages. It offers user-friendly functionalities for most operations that a user may want to carry out, specifically setting joint or pose goals, creating motion plans, moving the robot, and adding objects into the environment. Indeed, the planning of a trajectory can be accomplished by just setting the current robot configuration and the specified target.

The default motion planners in MoveIt! are configured to use *OMPL*, which is an open-source motion planning library that implements randomized planners [47]. Depending on the planner used, MoveIt! can choose between joint space and Cartesian space for the representation of the problem. Natively, planning requests with orientation path constraints are sampled in Cartesian space.

OMPL also provides several sample-based optimal planning algorithms. Some of them use a general framework to express the "cost" of robot configurations and paths, allowing to, e.g., maximize the minimum clearance along a path, minimize the mechanical work, or some arbitrary user-defined optimization criterion. However, the convergence to optimality is not guaranteed when optimizing over some index or metric which is not the path length [48].

4.4 Optimal Planners and Redundancy Resolution

In robotic missions for space exploration, robots are assigned a number of objectives, achieved by completing one or more tasks. During the execution of these tasks, the robot spends a lot of resources, which can be difficult to fetch on a remote and unfamiliar territory. In these situations, where resource depletion is a crucial matter, the optimal planning of the task, and, thus, of the robot motion is essential. Optimal planners are especially suited for space exploration missions since they allow the definition of a quality metric, or a performance index, to be optimized during the execution of the task. For instance, one of the key resources that an autonomous vehicle should preserve is the power consumption, hard to produce on planets distant from an energy source.

One way to exploit the main feature of optimal planners is with the usage of redundant robots, whose extra degrees of freedom help in achieving the minimization or maximization of a cost function. The higher the degree of redundancy, the better the optimization of the quality metric will be. Conversely, increasing the number of redundancy parameters, the complexity of finding a solution increments as well.

Among the available methods for redundancy resolution, approaches based on dynamic programming seems to fit well the requirements of the problem. An example of redundancy resolution by means of dynamic programming techniques was presented in Chapter 3, which refers to researches done on the globally-optimal resolution of redundancy exposed in [27]. This method relies on a grid search algorithm that solves the inverse kinematics problem for each discretized value of the redundancy parameters. Following this approach, the realization of a motion planner, and, in particular, of a trajectory planner, requires the help of kinematics tools to work out the robot inverse kinematics problem.

To this purpose, MoveIt! not only includes a series of inverse kinematics solvers by default, but it also allows users to write their own algorithms for the inverse kinematics resolution.

4.5 Inverse Kinematics Solvers

MoveIt! offers a plugin infrastructure to manage both motion planners and inverse kinematics solvers. The default inverse kinematics plugin for MoveIt! is configured for using the *KDL* kinematics plugin. The KDL plugin wraps around the numerical Jacobian-based inverse kinematics solver provided by the Orocos KDL package [49]. Although it works fine with robots having a number of DOM greater than 6, it does not fit well for robots with DOM less than 6. MoveIt! grants alternatives to its default solver as well, such as the *LMA* (Levenberg-Marquardt) kinematics plugin, that also wraps around a numerical inverse kinematics solver provided by the Orocos KDL package, and the *TRAC-IK*, developed by TRACLabs [50], that combines two IK implementations to obtain more reliable solutions than common available open source IK solvers.

However, as seen in Chapter 3, numeric solvers are too slow when dealing with extremely large amounts of computations, needed for the generation of the grids, and, if available, analytical methods must be used.

Analytical inverse kinematics solvers can be significantly faster than numerical solvers and grant more than one solution for a given end-effector pose. To this end, MoveIt! provides a tool to create plugins from C++ code generated with the robot kinematics compiler *IKFast*. The IKFast open-source program can solve for the complete set of analytical solutions of most common robot manipulators and generate C++ code for them. The generated solvers can be rapidly wrapped into plugins, retrieving all the inverse kinematics solutions in microseconds, on recent computers.

The key features that make IKFast one of the most valued robot kinematics compilers are [51]:

- the handling of robots with any number of DOM (with some limitations)
- the handling of robots with arbitrary joint complexity (e.g., non-intersecting axes)
- the computation of all possible discrete inverse kinematics solutions
- the detection of degenerate cases, e.g., where two or more axes align and generate infinite solutions

Despite that, some restrictions are present. First, IKFast only works with kinematic chains. Second, although IKFast is able to manage by itself the choice of redundancy parameters, it also permits to manually select these "free" joints. However, while it is possible to pick any arbitrary joint, from the base to the tip of a kinematic chain, choosing as redundancy parameters joints far from the tip (or end-effector) highly decreases the chances to obtain a solution to the inverse kinematics problem. So, as the author of IKFast suggests, a general rule of thumb for the choice of redundant parameters, when working with IKFast, is that the closer it is to the end effector, the better. Also, an analytic inverse kinematics solution cannot always be found by means of IKFast, especially if the robot number of DOM is greater than 7, e.g., for most mobile manipulators. In that case, MoveIt! does not support the generation of a plugin and the inverse kinematics solutions cannot be obtained.

Therefore, when dealing with special typologies of robots, the major problem is the acquisition of a working inverse kinematics solver. Generally, it is not hard to find inverse kinematics tools for common robotic structures, such as manipulators or mobile platforms. A big shortage in the implementation of control algorithms for compound exploration robots is the lack of analytic inverse kinematics solvers.

The standard strategy is to divide the motion planning of the robot into multiple steps. Considering, for instance, the case of a mobile manipulator, the first step would be the computation of the inverse kinematics solution for the moving platform. Then the motion of the vehicle would follow. When the moving platform stops, the manipulator inverse kinematics is solved and, with it, the motion of the arm is performed. It is clear that this way a lot of time is spent in sending signals for communications; also, the task may not be carried out in an optimal way since, to this end, the structure has to be considered as a whole to globally optimize some quality metric.

To this purpose, the solution proposed in this dissertation consists in building a plugin that performs the kinematic inversion of redundant systems composed of a manipulator, also called *arm*, mounted on a mobile platform, also termed *base*.

Once the inverse kinematics solutions are obtained for the specified robotic structure, a dynamic programming algorithm determines, through the optimization of a desired cost functional, the most efficient way the trajectory has to be traveled, i.e., the set of joint position values that minimizes or maximizes the overall performance index along the whole trajectory.

Chapter 5

Design of an Optimal Planner

5.1 Overview

Optimal planners are able to find the best joint space trajectory a robot has to follow, given a task space path and the corresponding time law. Essentially, they solve optimization problems in which one or more quality metrics, or performance indices, are minimized or maximized. One practical difficulty, in this sense, is the development of techniques and tools for the determination of a solution to the optimization problem. Typically, sample-based planners are employed, but the attained solutions may not be globally optimal. This means that the obtained optimized solutions for each trajectory sample do not always provide the global minimum or maximum value of performance index for the whole trajectory.

As remarked in Chapter 4, the building of an optimal planner must take into account redundancy resolution, and, as seen in Chapter 3, a dynamic programming approach can be employed to solve the redundancy in a globally-optimal way. To this purpose, an implementation of a DP-inspired algorithm has been provided by ALTEC, using the formulation in [33]. The algorithm has been demonstrated to work fine with redundant robots, such as 4R and 7R manipulators, that only have one redundancy parameter. The major challenge is to generalize its formulation so to support robots with an arbitrary number of degree of redundancy, r.

The aforementioned algorithm searches for the optimal trajectory in the robot joint space, exploring all the extended aspects at once, and, if necessary, performing robot reconfiguration, moving from one extended aspect to another. Therefore, the generalization of the algorithm involves the acquisition, or the creation, of robots having a degree of redundancy greater than 1, and tools for inverse kinematics resolution.

5.2 Mobile Manipulators

Redundancy is a concept that can only be defined in association with an assigned task. Accordingly, a conservative decision would be the study of robots applied in tasks having the highest DOF, that is, considering tasks characterized by 6 degrees of freedom in the 3D Cartesian space. The first issue, then, is to procure a robot with a high number of DOM, and, in particular, greater than 7.

Concerning space exploration, the generic task imposes a robotic structure able to move on extreme terrains and capable of dexterous manipulations. For example, in the perspective of planet colonization, a demanding operation could be the remote construction of buildings or infrastructures. In this view, mobile manipulators can be considered a suitable option, being equipped with a moving platform, for the motion over uneven terrain, upon which a robotic arm is mounted for manipulative purposes.

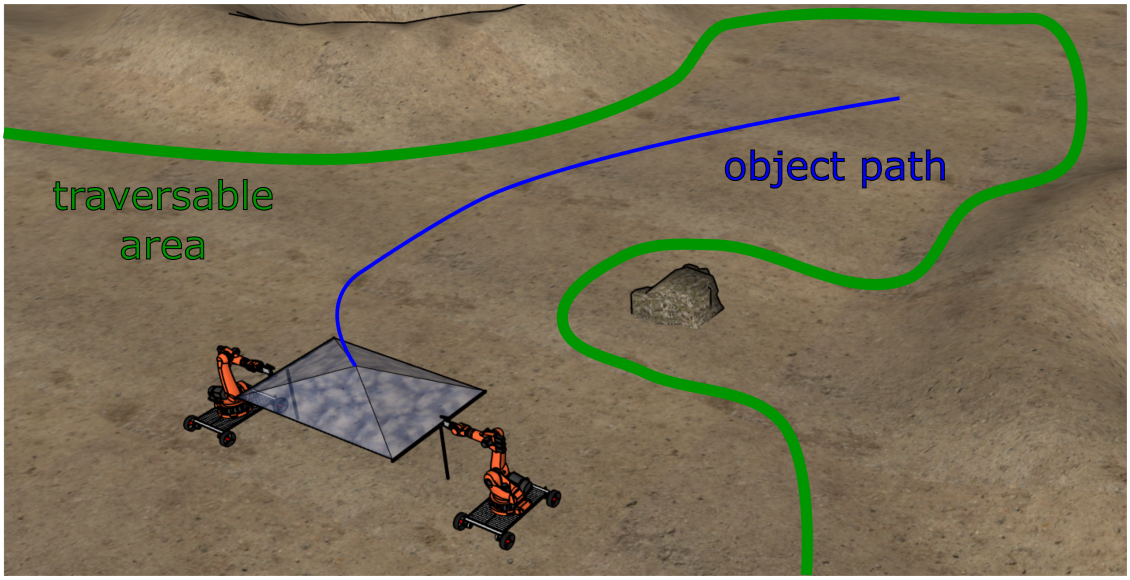


Figure 5.1: Representation of a building task for cooperative robots.

Choosing mobile manipulators as a reference structure, the first step was to find a virtual model to start from for the implementation of the trajectory planner. In doing this, some mobile platforms have been taken into account, such as ClearPath's Husky and Jackal. The latter has been employed in the beginning, but due to issues related to the lack of inverse kinematics solvers and the difficulty in generating one, it has eventually been discarded. Not so many difficulties with the manipulator selection have been encountered instead.

Several robotic arms have been examined, such as Franka Emika's Panda, Kinova's Jaco and Mico, and Universal Robots UR5, having regard for the availability of inverse kinematics solvers. Among them, two have been used to test the planner, namely the 7-DOM Franka Emika's Panda and the 6-DOM Kinova's Mico.

At this point, considering the implications of using a wheeled moving platform on the generation of an inverse kinematics solver, it was decided to create a moving base with suitable, but simpler characteristics. In fact, given the necessity to obtain an analytical solver, mainly due to the large number of computations that will be executed inside the algorithm, the most convenient way of formulating one inside ROS was by means of MoveIt! IKFast inverse kinematics plugin generator. The plugin generator is based on the IKFast robot kinematics compiler, which, unfortunately, only works with kinematic chains. Although the combination of a wheeled platform and a robotic arm could be seen as a kinematic chain, e.g., by using one wheel as active joint and mimicking the others, for a simpler management of the motion of the moving base, a prismatic platform has been designed on purpose.

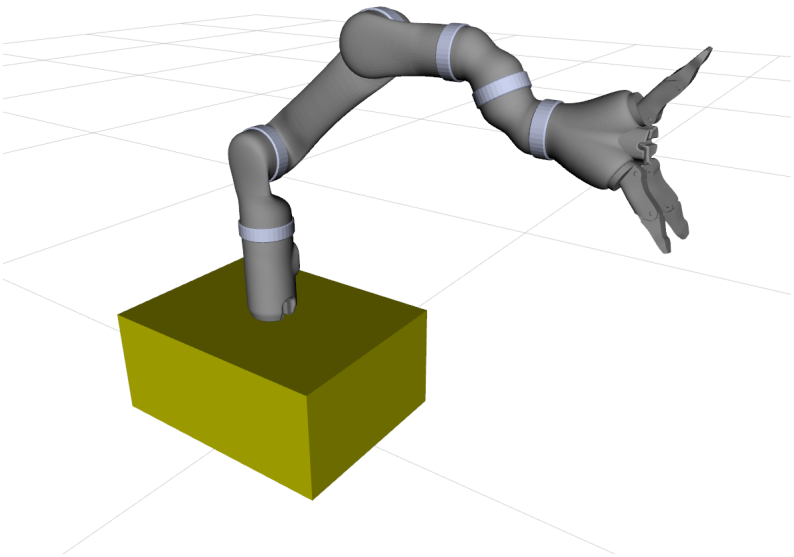


Figure 5.2: Model of mobile manipulator with Kinova Mico as robotic arm.

The underlying idea is that the base, modeled in a first instance as a parallelepiped, is composed of three prismatic joints (X, Y, Z) and three revolute joints (Roll, Pitch, Yaw), thus, totaling a 6-DOM structure. However, not all of them are active joints, i.e., controllable joints: three (active) are chosen as command inputs and three (passive) are given by the terrain shape transformation. This way, active joints are chosen as redundant joints that will be parametrized during the redundancy resolution process. The most appropriate choice is to select as redundant joints a triad of two prismatic joints and a revolute joint: (X, Y, Yaw).

5.3 Base-Arm Kinematics Plugin

The next step in building a planner is to provide an interface for the kinematics management, so to obtain the inverse kinematics solutions for mobile manipulators, or *Base-Arm* robotic systems.

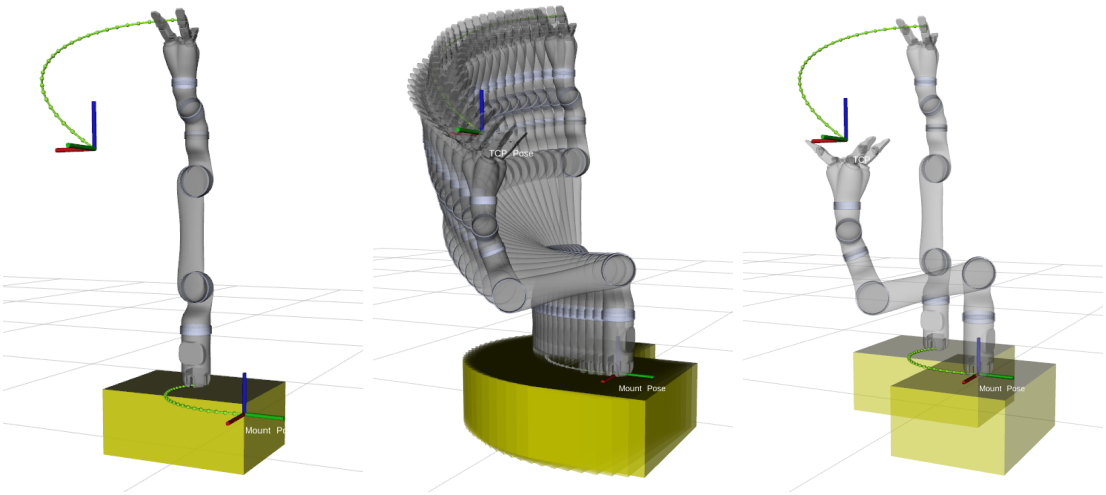
As seen in Chapter 4, while analyzing the availability of inverse kinematics solvers, MoveIt! offers a plugin infrastructure to handle motion planners and kinematics tools. In principle, it is possible to build a kinematics solver by implementing the *KinematicsBase* class methods, which is a MoveIt! interface that enables users to either write their own forward and inverse kinematics solvers or wrap around external kinematics solver libraries. On this thread, the *base-arm kinematics plugin* has been designed, allowing kinematics computations for robotic systems composed of an arm mounted on a mobile base.

The base-arm plugin is part of the *base-arm* package suite, together with the *base-arm description* package, for the handling of robotic structures comprising a base and an arm. The two packages work jointly providing both the robot models, through description files such as *URDF* and *XACRO*, and the management of their kinematics. The main class of the base-arm plugin implements several methods derived from *KinematicsBase*, among which the most relevant one deals with kinematic inversion.

Due to the lack of a kinematics solver that solves the redundant inverse kinematics problem of systems with $r > 1$, the proposed technique tackles this issue by breaking the problem into easier sub-problems. Basically, the inverse kinematics resolution of the kinematic chain, composed by the whole mobile manipulator, is separated into two resolution processes, involving the single sub-chains, i.e., base and arm. In MoveIt!, *planning groups* are used for semantically describe different parts of the robot. In this case, the three principal planning groups are the *arm planning group*, the *base planning group* and the *chain planning group*; the latter includes the previous two.

A kinematics plugin can be associated to each planning group, that performs computations for the forward and inverse kinematics resolution of the group kinematic sub-chain. Each plugin, assigned this way, is provided of a kinematic solver for the resolution of at least the inverse kinematics problem. Following this logic, the chain planning group is linked to the base-arm kinematics plugin and, thus, to the corresponding kinematics solver. Actually, the base-arm inverse kinematics solver wraps around both the base group solver and the arm group solver: it first sets the base redundant joints, which are also the active joints, then computes the transformation of the base mount point (on which the arm mount link is rooted) in the joint space, and eventually feeds the new arm root position to the arm kinematics solver, that yields the solution for the manipulator.

The following figures show the motion of a mobile manipulator formed by the parallelepiped base and the Kinova Mico 6R (M1N6S300). The base-arm plugin has been used to pick one among a set of different solutions obtained from the manipulator analytical kinematics solver. Indeed, while the base planning group does not make use of any inverse kinematics solver, since all the active base joints are redundancy parameters and must be set, the arm inverse kinematics solver returns a set of at most 8 (theoretically 16, as seen in Chapter 3) analytic solutions. In this simulation, where the goal was to check the correct behavior of the plugin, only the initial and final end-effector poses were provided, while OMPL was used to choose an arbitrary planner to solve the trajectory in the robot joint space.



(a) Starting configuration (b) Motion trail (c) Start/End configuration

Figure 5.3: Base-arm kinematics plugin applied to a mobile manipulator.

One of the peculiarities of the base-arm model is its flexible kinematic description. The choice of having a 6-DOM base built with nonholonomic constraints (i.e., not all of its joints are controllable) allows controlling its motion as if a terrain with variable geometry was present. While three base joints are set as active, the remaining three joints are used to simulate the motion of the base over a terrain: a transformation computes the values of non-controllable base joints taking in the active joints through a set of primitive geometries.

Furthermore, the plugin permits to manage the inverse kinematics for base models with a lower number of joints. In fact, the designed base model also makes possible to operate with only three joints, namely (X, Y, Yaw), instead of 6 (X, Y, Z, Roll, Pitch, Yaw), so that systems typical of many industrial applications, such as manipulators mounted on a 1-DOF or 2-DOF slider, can be taken into account. This way, since the number of controllable joints decreases, the handling of redundancy parameters during the kinematic resolution process is made easier.

5.4 Redundancy Resolution Service

Granted that, for each waypoint of the given trajectory and for each combination of redundancy parameter values, the sets of inverse kinematics solutions are computed, the last step in building the optimal planner is to find, among the freshly computed joint space solutions, those that optimize the desired performance index. The idea is to build a redundancy resolution ROS service based on a generalization of the DP-inspired algorithm [33]. While the original algorithm only supports one redundancy parameter, the proposed implementation extends to $r > 1$ the number of degrees of redundancy.

5.4.1 Grid Generation and Generalized DP Algorithm

Following the procedure in Chapter 3, the computation of the grids (such as in Figure 3.2) is fundamental for the implementation of the algorithm. By definition, the grids exist in a space of dimension $\mathbb{R}^{r+1}$, where r is the degree of redundancy of the robot (3.1). Each grid is composed of $r + 1$ axes: one axis is assigned to the trajectory waypoints, while the other r axes are assigned to each discretized redundancy parameter. So, the first enhancement to be done is to generalize the data structure of the grids.

The library employed for this improvement is *Boost.MultiArray* [52], which allows the building of N-dimensional arrays and provides an interface to operate with these data structures. Among the evaluated options two are worth to mention:

- The C++ template library for linear algebra, *Eigen*, supporting matrices, vectors and numerical solvers.

  + It is fast, reliable, and versatile, supporting matrices of all sizes, all standard numeric types, and matrix decomposition.

  - There is no implementation for matrices with more than two dimensions.

- The Standard Template Library container *std::vector* for a linear representation of multi-dimensional matrices.

  + Vectors are able to represent dynamic arrays with the ability to resize themselves automatically when an element is inserted or deleted.

  + One vector is employed to contain all grid elements, providing optimized methods to manage the std::vector structure.

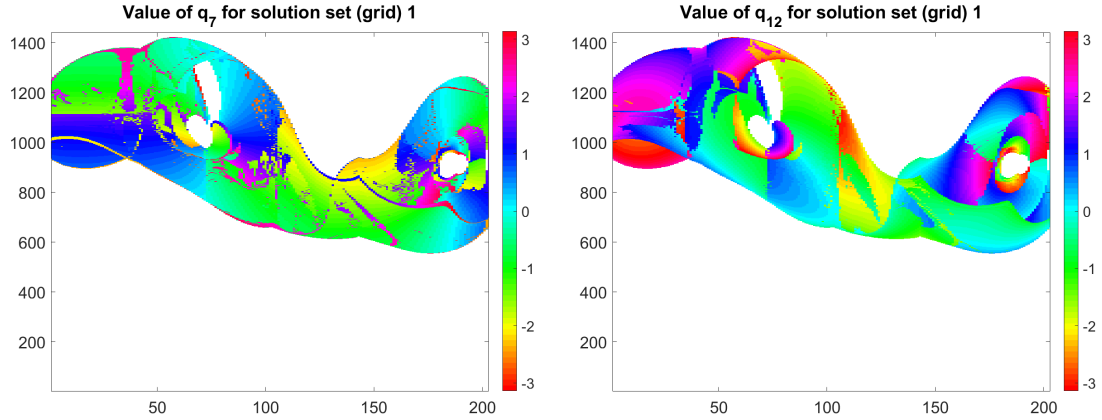
  - An indexing method has to be provided to correctly map each element of the array to the corresponding element of the multi-dimensional matrix.

  - Although this approach is feasible, a test suite may be required to verify the behavior of operations executed on the data structure.

A significant element to take into account, during the generalization of the service, is the number of redundant degrees of freedom r. When using the data structure provided with *Boost.MultiArray*, r must be chosen at compile time, so a decision on the upper limit of r has to be taken. Considering the limitations of nowadays processing power, it has been decided that the maximum number of redundancy parameters of the robot is $r_{max} = 4$, otherwise the computational complexity would be unreasonable.

Although the generalization of the grids gives the possibility to include more than one redundancy parameter, the development of $(r+1)$-dimensional structures does not consent any graphical representation for $r > 2$. A picture of a 3-dimensional grid would not be advantageous as well. In fact, subspaces of inverse kinematics solutions often show a sparse distribution inside the examined joint space sector, and the representation of a 3-dimensional entity would not permit to visualize the interior of it, so it would not be of practical interest. However, one technique that could be useful in this case is to divide the object into several slices, for instance, one at each waypoint, and analyze each slice individually.

Despite that, an example of non-homogeneous grids generated for a mobile manipulator with $r = 1$, using the base-arm kinematics plugin, is illustrated in Figure 5.4. These grids are obtained employing the Kinova Mico 6R as the robotic arm and the 6-DOF parallelepiped as the moving platform. In order to represent a two-dimensional grid, only one redundancy parameter must be set: choosing X as prismatic redundant joint, the remaining two joints (Y, Yaw) are fixed at their initial values.



(a) Arm Joint 1 (b) Arm Joint 6

Figure 5.4: Non-homogeneous grids for different arm joint values.

Once the grids are generated, they are sent to the DP-inspired algorithm, which computes the value of the cost functional at each node of every grid. The algorithm is, basically, an optimal path search algorithm: for each node, at a given waypoint, it explores the adjacent cells at next waypoint and checks if they are feasible; for each feasible node, the cost value is computed, verifying if it satisfies the optimization criterion. This recursive algorithm stops when the last waypoint is reached, or if no feasible node is found at next waypoint. In order to achieve a feasible trajectory, each waypoint must include at least one feasible node. Moreover, the optimal trajectory is found if and only if the final value of the objective function $I(0)$ (3.87) is the optimum (minimum or maximum).

An applicative example of the operating principle of the generalized algorithm is provided below.

5.4.2 Flowchart

For the sake of clarity, and in order to provide a clear insight into the core logic of the algorithm, the flowchart diagram of the dynamic programming solver is illustrated here. It depicts the decision process at the base of the solver, capable to work out the redundancy problem for systems composed of n redundancy parameters. Figure 5.5 and Figure 5.6 represent the flowchart of the generalized DP-inspired algorithm.
To properly comprehend the adopted symbolism, some definitions are given:

- n is the number of redundancy parameters the robot is provided
- j denotes the $j-th$ element of the grid index vector
- S_j is the boolean corresponding to a switch in the exploration direction of the $j-th$ grid index; it evaluates to true if a switch is requested
- Δ_j is the boolean corresponding to whether increase or decrease index j; it evaluates to true if grid index j is increasing

In addition, initial values are defined this way:

- $\boldsymbol{S} = [S_1, \ldots, S_n] \leftarrow [False, \ldots, False]$
- $\boldsymbol{\Delta} = [\Delta_1, \ldots, \Delta_n] \leftarrow [True, \ldots, True]$
- $\boldsymbol{idx} \leftarrow \boldsymbol{idx}_0 = [idx_{0,1}, \ldots, idx_{0,n}]$

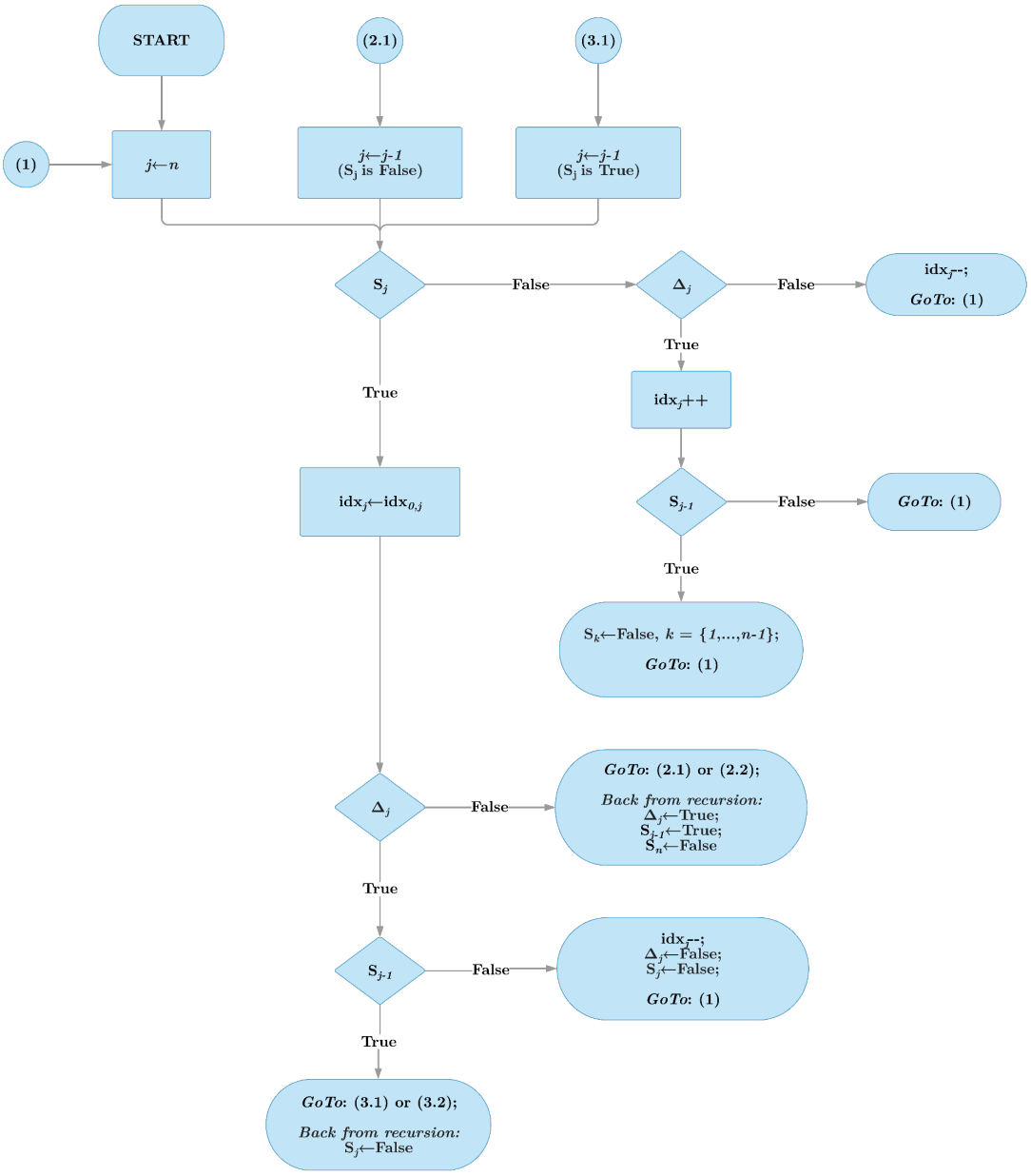


Figure 5.5: Node exploration at generic index j.

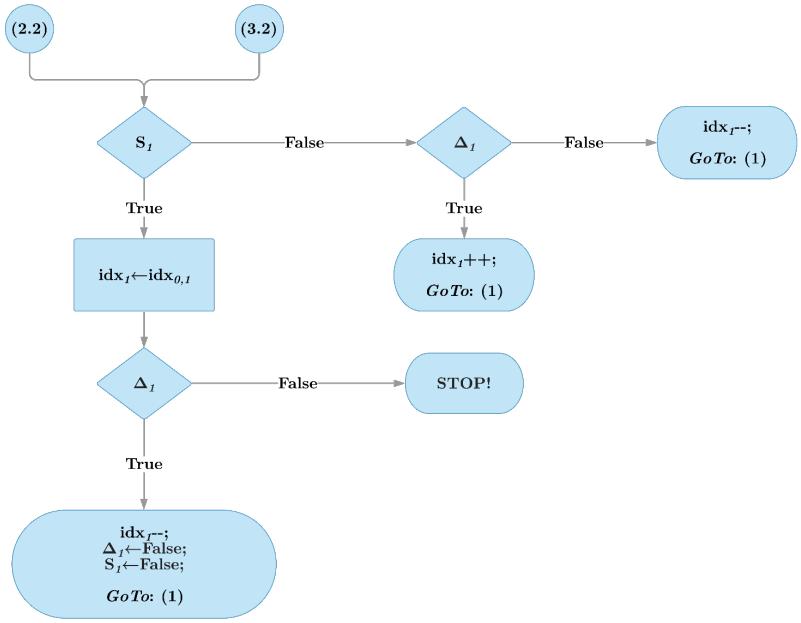


Figure 5.6: Node exploration at index 1.

5.4.3 Example with Two Redundancy Parameters

This section provides an example that clarifies the operating principle of the generalized algorithm. In particular, this example shows how the algorithm works when two redundancy parameters are employed.

The two redundancy parameters, respectively q_1 and q_2, are sampled with discrete values at each waypoint. Redundancy parameter values are accessed through indices, in this case idx_1 and idx_2, stored into a vector $\boldsymbol{idx}$.

Figure 5.7 represents a slice of the state space grid at a fixed waypoint. The ellipse contains the feasible nodes, at the given waypoint, and each node corresponds to a data structure that includes the inverse kinematics solution for each pair of q_1 and q_2 values. Starting from an initial point on the grid, i.e., for a given pair ($idx_{0,1}$, $idx_{0,2}$) that identifies a node on the grid, the algorithm explores adjacent nodes in every dimension. The most "external" dimension is investigated first, which is the dimension corresponding to the last redundancy parameter, e.g., q_2, whose discrete values lie on the horizontal axis, while the discrete values of q_1 are located on the vertical axis.

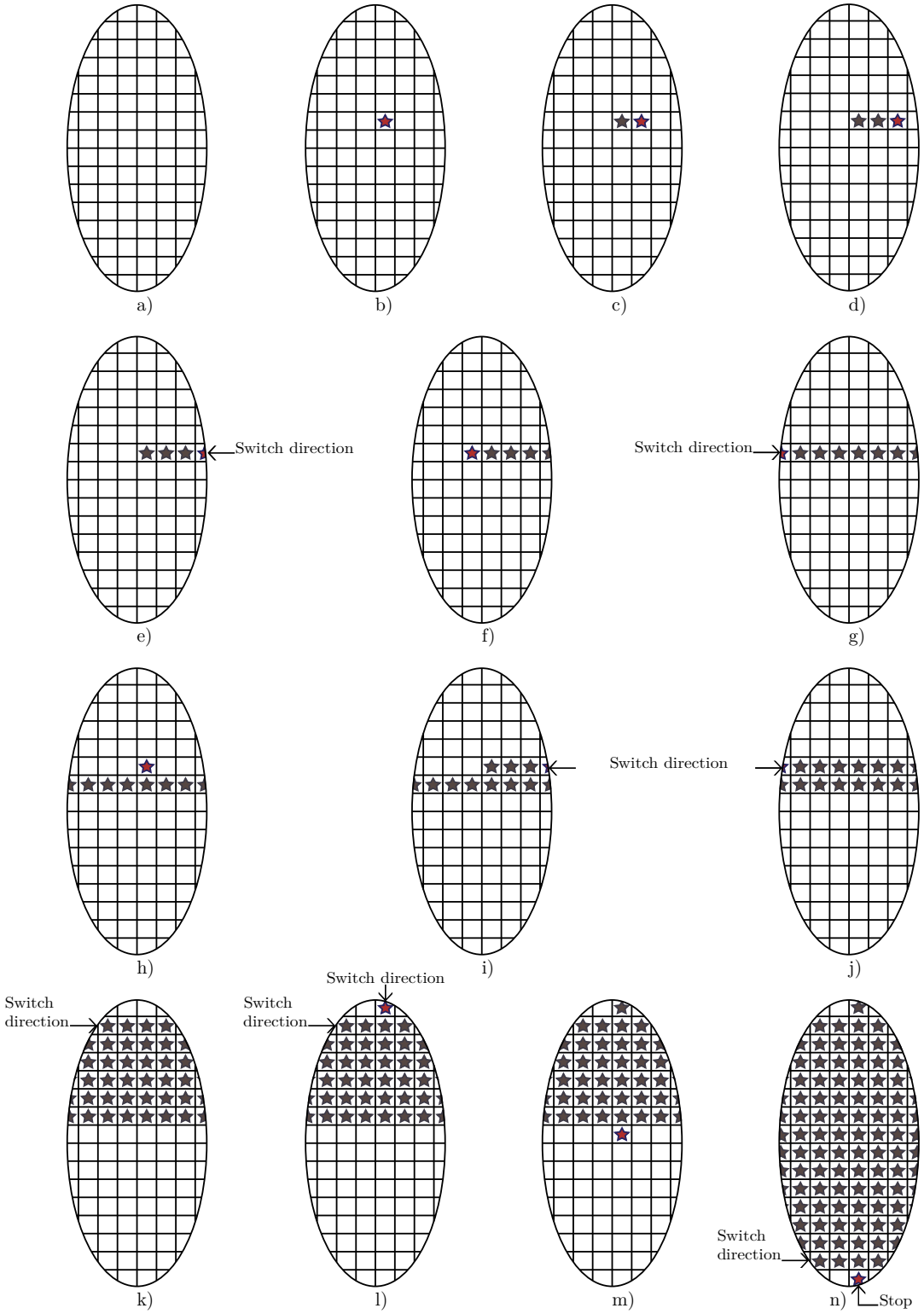


Figure 5.7: Node exploration inside grid at fixed waypoint.

In Figure 5.7, a) depicts an empty grid where no node has been explored yet.

Grid b) shows an initial node represented by a star with empty surroundings. The node is accessed through the vector $\boldsymbol{idx} = [idx_1, idx_2]$ and the cost function is computed. Then, the algorithm explores the successive node.

In grids c) and d), the algorithm advances through the dimension of q_2, increasing at each step the corresponding index idx_2 of one unit ($\Delta_2 = True$).

In grid e), the explored node is not a feasible node, either because the inverse kinematics solution stored inside it is not valid (it does not satisfy the kinematic constraints or no solution has been found) or because the maximum range limit of idx_2 has been reached. In this case, a switch in the search direction is requested ($S_2 = True$).

In grid f), the algorithm goes back to the initial node and starts stepping backwards ($\Delta_2 = False$), exploring all the nodes preceding the initial one until another switch is requested ($S_2 = True$), as showed in grid g).

At this point, in grid h), the algorithm begins the research in other dimensions, and jumps to the successive raw in the dimension of q_1, i.e., vertically ($\Delta_1 = True$). Thus, index idx_1 increases of one unit, and the algorithm starts examining the upper raw in the horizontal dimension.

Once no further exploration is possible in the "lower" dimension, as in grid l), two consecutive switches are requested ($S_1 = True, S_2 = True$), and the algorithms runs backwards in the dimension of q_1.

In grid m), vector $\boldsymbol{idx}$ is reset and index idx_1 decreases, so the exploration of the bottom raws starts.

The last grid in Figure 5.7 shows what happens when the last node is explored: since two consecutive switches are requested and no other dimension can be inspected, the algorithm comes to an end.

Chapter 6

Implementation of the Modules

6.1 Software Architecture

In Chapter 4 the major issues in the design of an optimal planner have been reported, while in Chapter 5 some possible solutions to these problems have been presented. Here, the proposed solutions are explored in a more detailed way, examined from the architectural point of view, investigating the implementation of the principal functions of the service.

6.1.1 Base-Arm Kinematics Plugin

The base-arm kinematics plugin is designed as a ROS package under the name of *base_arm_kinematics_plugin*. This package is placed in a parent directory, the *base_arm_kinematics*, which gathers kinematics tools. The base-arm kinematics plugin package is part of a bigger structure, the package suite *base_arm*, that also includes a robot description package *base_arm_description*, where robot models, initial configurations, and kinematic constraints are put together. Their relation with the *KinematicsBase* class is illustrated by the UML chart in Figure 6.1.

The package of the plugin consists of a main class, the *BaseArmKinematicsPlugin*, that manages the various methods utilized for kinematics computations. Among the different implemented methods, two of them require special attention: the one that handles the inverse kinematics solver, *getPositionIK*, and the one that runs the forward kinematics solver, *getPositionFK*.

The method *getPositionIK* takes in an initial guess solution for the inverse kinematics (including redundant parameters, if any), which can simply be the set of joints of the current robot configuration, and the end-effector pose specified in the task space.

The function returns either the set of joints corresponding to the solution with minimum distance from the initial guess or all the sets of inverse kinematics solutions found. In theory, for a given pose, the maximum number of inverse kinematics solutions is 16. This is valid as long as either 6-DOF non-redundant robots are considered or, as in this case, $(6 + r)$-DOF redundant robots with fixed redundant parameters are taken into account. Moreover, if non-orthogonal manipulators are employed, the maximum number of kinematic solutions is 8.

The method *getPositionFK* accepts as input parameters the robot configuration for which forward kinematics is computed (i.e., joint position values) and the set of links of the corresponding kinematic chain. The function computes the task space pose of each link of the robot.

One more method, worth to mention, is *applyTerrainTransformation_*: it receives the set of base joint values and transforms it accordingly to the terrain shape, returning the set with modified passive base joints.

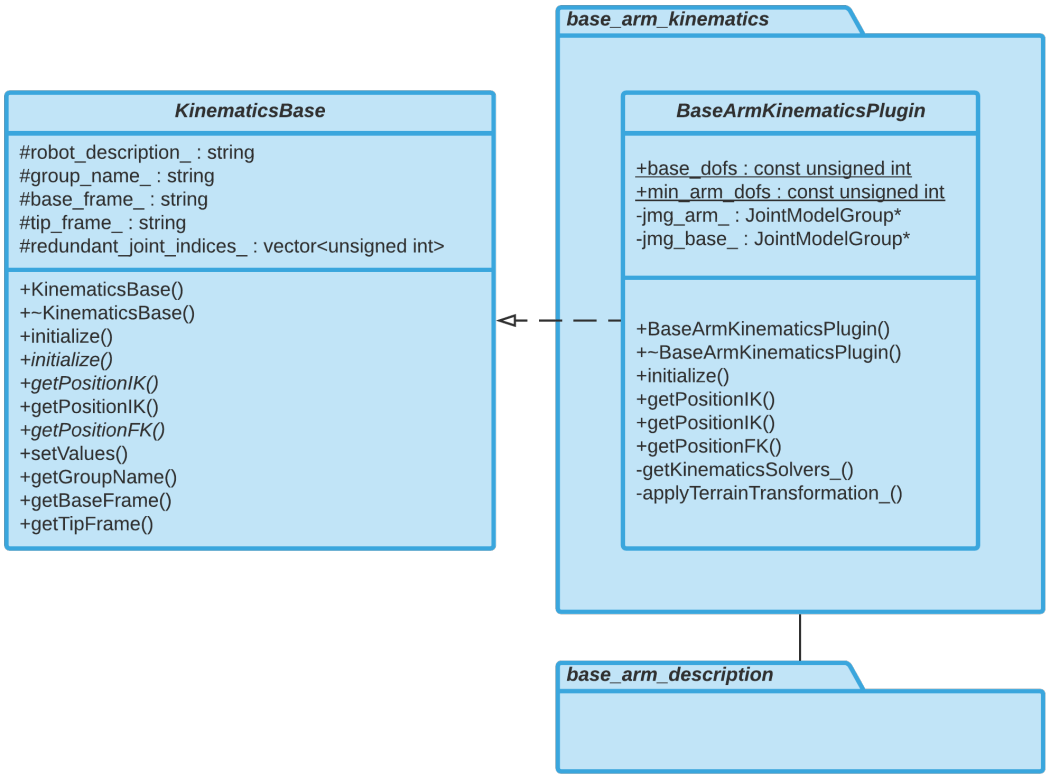


Figure 6.1: UML class diagram depicting the design structure of the plugin.

6.1.2 Redundancy Resolution Service

The redundancy resolution service is designed as a ROS service, comprehensive of two packages: *moveit_dp_redundancy_resolution*, that collects all the libraries necessary to implement the redundancy resolution through dynamic programming optimization, and *moveit_dp_redundancy_resolution_msgs*, that includes messages and definitions necessary for the service to work in the right way. Unlike the plugin's layout, the arrangement of this service is organized in several classes:

- *WorkspaceTrajectory* generates a task space trajectory from the computed joint space trajectory
- *StateSpaceMultiGrid* creates state space multi-dimensional grids, solves the inverse kinematic problem, stores the solutions in node objects and provides import/export functionalities for the grids
- *ObjectiveFunction* manages the desired performance index and the applied optimization criterion
- *DynamicProgrammingSolver* executes the dynamic programming algorithm and returns the globally-optimal solution in the form of a joint space trajectory
- *MoveGroupDPRedundancyResolutionService* comprises the main service of the plugin, supervising and controlling the other classes

The primary aspects of the generalization reside in having a new grid data structure, that supports more than two dimensions, and the extension of the search algorithm, that handles more than one redundancy parameter.

Concerning grid generation, most of the effort has been put on the provision of methods to assist operations with the new data structure, *boost::multi_array*. This data structure acts as a container for the element of the grids. Each element is an instance of the *StateSpaceNode* class, which represents a unique grid node. Node objects are given a generic implementation so that future developments can grow in many directions.

Regarding the dynamic programming solver, a large part of the work focused on making the multi-dimensional grids, objects of class *StateSpaceMultiGrid*, work with already existing solver methods (generalized, if necessary). In some cases, iterative structures were specifically constructed for node indexing, compelled by some operations on grid nodes.

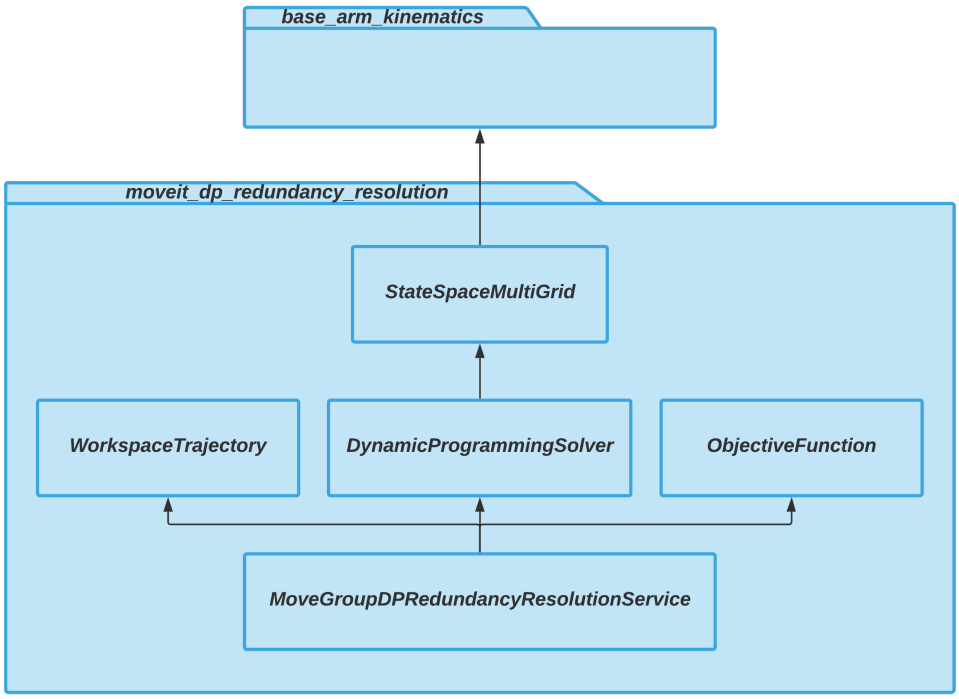


Figure 6.2: UML class diagram depicting the design structure of the service.

6.2 Test Suites

The verification of the previously analyzed concepts is carried out by means of unit tests. *Unit testing* is a level of software testing where individual components of a software are tested [53]. It is the first level of software testing and is performed prior to any other software tests, such as Integration Testing, System Testing and Acceptance Testing. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. In object-oriented programming, the smallest unit is a method, which may belong to a base/super class, abstract class or derived/child class.

Unit tests are usually performed by using the *White Box Testing* method [54]. White box testing (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure of the item being tested is known to the tester. The tester chooses inputs to exercise paths through the code, and determines the appropriate outputs. This method is named so because the software program, in the eyes of the tester, is like a white (transparent) box, inside which one clearly sees.

There are some reasons [53] that make unit tests vital for a proper writing of software:

1. Unit testing increases confidence in changing or maintaining code. If good unit tests are written and if they are run every time any code is changed, it will be easier to catch any defects introduced due to the change.

2. Codes are more reusable because they need to be modular to make unit testing possible.

3. The effort required to find and fix defects found during unit testing is much less in comparison to the effort required to fix defects found during system testing or acceptance testing. Also, the cost of fixing a defect detected during unit testing is less than the one of defects detected at higher levels.

4. Debugging is easy. When a test fails, only the latest changes need to be debugged. With testing at higher levels, changes made over the span of several days, weeks or months need to be scanned.

5. Codes are, in general, more reliable if they are provided of unit tests.

In this section, unit tests are executed with the aid of *gtest* (or googletest [55]), which is a testing framework developed with Google's specific requirements [56].

6.2.1 Test Suite for Base-Arm Kinematics Plugin

The performed unit tests that verify the reliability of *BaseArmKinematicsPlugin* class methods are the following:

- *SolutionFound_getPositionIK_AllSolutions*: verifies whether the method *getPositionIK*, which returns all the solutions to the inverse kinematics, returns true or false.

- *SolutionFound_getPositionIK_MinDistanceSeedSolution*: for arms with less than 7-DOM, it checks that the method *getPositionIK* returns the joint space solution closest to the initial seed; for arms with more than 6-DOM, it verifies that the method correctly throws an exception, as an implementation is not available for such arms.

- *SolutionFound_getPositionFK_AllSolutions*: verifies that *getPositionIK*, returning all the solutions to the inverse kinematics, and *getPositionFK* methods work correctly together; the test fails if the error between the goal pose and computed pose is greater than a given absolute error.

- *8SolutionsFound_getPositionIK_AllSolutions*: this very specific test checks that *getPositionIK* returns at least 8 solutions to the inverse kinematics for a given end-effector target pose and fixed base joint coordinates.

- *SetJointIndices_getRedundantJoints*: checks that the method *getRedundantJoints* correctly retrieves the redundant joints of the base-arm chain.

The class has been tested on mobile manipulators composed of arms, such as Franka Emika's Panda (7-DOM) and Kinova Mico (6-DOM), mounted on a 6-DOM base.

As a result of the execution of these tests, a remarkable flaw emerged: one of the arm analytic inverse kinematics solvers, obtained from the official repository of the manipulator, retrieved a wrong solution to the inverse kinematics for that manipulator. A fix to this issue has been provided by regenerating the inverse kinematics plugin for the concerned arm by means of MoveIt! IKFast plugin generator.

6.2.2 Test Suite for Redundancy Resolution Service

The performed unit tests that verify the reliability of *StateSpaceGrid* and *StateSpaceMultiGrid* classes are the following:

- *ComputeGrids*: verifies that method *computeGrids()* from class *StateSpaceMultiGrid* returns true.
- *EnableNode*: verifies that methods *isNodeValid()*, *enableNode()* and *isNodeEnabled()* from class *StateSpaceGrid* return true when 8 solutions to the inverse kinematics are expected at given node.
- *CompareGrids*: verifies that grids binary files computed with method *exportToBinary()* from class *StateSpaceGrid* are byte-wise equal to those calculated with previous implementation in [33].

The performed unit test that verifies the reliability of *DynamicProgrammingSolver* class is the following:

- *ReturnRobotTrajectory*: verifies that the DP-inspired generalized solver is able to return an optimal solution, that is equivalent to check that method *solve()* from class *DynamicProgrammingSolver* returns true.

The classes have been tested on Franka Emika's Panda (7-DOM) manipulator and verified that obtained results match those in [33].

Chapter 7

Conclusions

7.1 Results

Future planetary mission scenarios are shifting from robotic exploration towards colonization of satellites and planets. As a result, robots capable of supporting tasks for construction activities are required. In these circumstances, engineers can rely on cooperative robots that can collaborate with each other to accomplish tasks that individual robotic systems cannot achieve.

Major companies interested in space exploration and colonization are developing advanced technologies for ground control centers to support planning and control of complex robots. Mobile manipulators, consisting of robotic arms mounted on moving platforms, are used in the present dissertation as an example of representatively complex robotic systems that are not trivial to control.

The aim of the present dissertation is to contribute to the development of these technologies, setting two objectives:

1. to build a plugin for the inverse kinematics of mobile manipulators
2. to generalize a redundancy resolution service for optimal trajectory planning

The present activity has resulted in the attainment of both the predefined objectives in the measure of:

- building a kinematics plugin for the handling of forward and inverse kinematics of mobile manipulators
- providing a basic interface for platform joints transformation due to the terrain shape
- allowing the management of multi-dimensional grids through an appropriate data structure
- generalizing a DP-inspired algorithm for the handling of multiple redundancy parameters

7.2 Future Works

Results presented in this dissertation pave the way to many future developments. Some suggested improvements and analyses concern the:

- creation of use case with mobile manipulator to acquire data on final performance indices values
- analysis through test suite of base-arm kinematics plugin with mobile manipulators having different base models, such as Clearpath's Jackal
- set of additional terrain shapes or take in numerical terrain structure when handling the motion of the platform in presence of non-flat terrains
- analysis of 3-dimensional grids through slices at various waypoints
- improvement of base-arm kinematics plugin for the generation of homogeneous grids
- parallelization of the reading process of grid nodes inside the DP algorithm in order to decrease computational time

Appendix A

Calculus of Variations

A.1 Principal Problems in Calculus of Variations

The inverse kinematic problem for non-redundant manipulators brings to a finite set of solutions. If the manipulator is redundant, there can be infinite solutions and it is possible to select one among them by applying a criterion as an additional constraint to the problem, such as, the minimization of a cost functional.

The *optimal solution* to the non-redundant kinematic inversion problem is supported by calculus of variations. In a broader sense, calculus of variations deals with functional optimization problems. Before listing some of the main problems found in calculus of variations, a key quantity has to be defined.

Considering a function $x(t)$ defined in a certain time interval, then the scalar variable $\boldsymbol{F}$ is called a functional of $x(t)$ if for each value of $x(t)$ corresponds a scalar value of $\boldsymbol{F}$:

$$\boldsymbol{F} = \boldsymbol{F}(x(t)) \tag{A.1}$$

A classic example of a functional $\boldsymbol{F}$ is the definite integral:

$$\boldsymbol{F} = \int_{t_0}^{t_1} x(t)dt \tag{A.2}$$

For a fixed interval of integration $[t_0, t_1]$, $\boldsymbol{F}$ only depends on $x(t)$.

A.1.1 Fixed endpoint problem

The first problem explored here, proper of calculus of variations, consists in finding the trajectory $x(t)$ that minimizes a cost functional at a fixed endpoint.

Given a function $x(t)$ defined over the time interval $[t_0, t_1]$, the problem can be formulated this way:

$$\min_{x(t)} F(x) = \int_{t_0}^{t_1} f(t, x(t), \dot{x}(t))dt \tag{A.3}$$

assuming $f(t, x(t), \dot{x}(t))$ integrable over $[t_0, t_1]$ and differentiable with respect to all its arguments. The trivial necessary condition for a function $x^*(t)$ to be the minimizer of a functional $\boldsymbol{F}$ is that $\boldsymbol{F}(x) \geq \boldsymbol{F}(x^*), \forall x(t)$. However, this condition is not useful from an application point of view: it is essential to find an alternative condition that allows to determine the minimizing solution $x^*(t)$.

First-order necessary condition

The optimal solution to the preceding problem, assuming free boundary values, is the one that satisfies the Euler-Lagrange equation (A.4) with corresponding boundary conditions (A.5):

$$\frac{\partial f}{\partial x} - \frac{d}{dt}\left(\frac{\partial f}{\partial \dot{x}}\right) = 0 \tag{A.4}$$

$$\delta x(t_0)\left(\frac{\partial f}{\partial \dot{x}}\right) = \delta x(t_1)\left(\frac{\partial f}{\partial \dot{x}}\right) = 0 \tag{A.5}$$

where $\delta x(t_0)$ and $\delta x(t_0)$ are the variations at endpoints. On the contrary, if boundary values $x(t_0)$ and $x(t_1)$ are constrained, then the boundary conditions for (A.4) take values $\delta x(t_0) = \delta x(t_1) = 0$.

Second-order necessary condition

The previous result comes from an analysis on the derivative (or, *first variation*) of $\boldsymbol{F}$ and gives a necessary condition at endpoints. However, it does not ensure that $x^*(t)$ is a minimizer for $\boldsymbol{F}$, so it is essential to study the *second variation* of it.

A non-negative second variation of $\boldsymbol{F}$ guarantees a second-order necessary condition for a minimum, that is, the variation $\Delta\boldsymbol{F}$ is always positive. The non-negativity condition is satisfied if function f is convex in both variables $(x(t), \dot{x}(t))$.

Actually, it is possible to demonstrate that a non-negative second variation of $\boldsymbol{F}$, along the optimal trajectory, requires the convexity of f only in $\dot{x}(t)$. Therefore, the second-order necessary condition for function $x^*(t)$ to be a minimizer of $\boldsymbol{F}$ is that, along that whole trajectory, the following inequality holds:

$$\frac{\partial f^2}{\partial x^2} \geq 0 \tag{A.6}$$

A.1.2 Fixed endpoint constrained problem

The second problem examined consists in finding a trajectory that satisfies a certain constraint and minimizes a specified cost functional.

Considering a function $x(t)$ defined over the time interval $[t_0, t_1]$, the problem is formulated as follows:

$$\begin{aligned} \min_{x(t)} F(x) &= \int_{t_0}^{t_1} f(t, x(t), \dot{x}(t))dt \\ s.t. \quad & v(t) = g(x(t)) \end{aligned} \tag{A.7}$$

Then, defining a modified function f' that includes the constraint $v(t)$ by means of the *Lagrange multiplier* $\lambda \in \mathbb{R}$:

$$f'(t, x(t), \dot{x}(t)) = f(t, x(t), \dot{x}(t)) + \lambda \left(v(t) - g(x(t))\right) \tag{A.8}$$

the previous problem is equivalent to:

$$\begin{aligned} \min_{x(t)} F(x) &= \int_{t_0}^{t_1} f'(t, x(t), \dot{x}(t))dt \\ s.t. \quad & v(t) = g(x(t)) \end{aligned} \tag{A.9}$$

which can be seen as a non-constrained fixed endpoint problem (A.3) plus an additional constraint $v(t) = g(x(t))$.
At this point, assuming free boundary values, the minimizing solution satisfies the following set of differential algebraic equations (DAE):

$$\begin{gathered} \frac{\partial f'}{\partial x} - \frac{d}{dt}\left(\frac{\partial f'}{\partial \dot{x}}\right) = 0 \\ \delta x(t_0)\left(\frac{\partial f'}{\partial \dot{x}}\right) = \delta x(t_1)\left(\frac{\partial f'}{\partial \dot{x}}\right) = 0 \\ v(t) = g(x(t)) \end{gathered} \tag{A.10}$$

A.1.3 Fixed endpoint multidimensional problem

Considering the trajectories vector $\boldsymbol{x}(t) = [x_1(t), x_2(t), \ldots, x_n(t)]$ defined over the time interval $[t_0, t_1]$, and supposing non-constrained boundary values, the third problem investigated consists in finding the optimal vector function $\boldsymbol{x}^*(t)$ that minimizes the functional $F(\boldsymbol{x})$:

$$\min_{\boldsymbol{x}(t)} F(\boldsymbol{x}) = \int_{t_0}^{t_1} f(t, \boldsymbol{x}(t), \dot{\boldsymbol{x}}(t))dt \tag{A.11}$$

where f is a scalar function in $\boldsymbol{x}(t)$, $\dot{\boldsymbol{x}}(t)$ and t.

As in the scalar case, it is possible to prove that the minimizer of $F(\boldsymbol{x})$ satisfies the boundary conditions of the Euler-Lagrange equation:

$$\frac{\partial f}{\partial \boldsymbol{x}} - \frac{d}{dt}\left(\frac{\partial f}{\partial \dot{\boldsymbol{x}}}\right) = 0 \tag{A.12}$$

$$\delta\boldsymbol{x}(t_0)\left(\frac{\partial f}{\partial \dot{\boldsymbol{x}}}\right) = \delta\boldsymbol{x}(t_1)\left(\frac{\partial f}{\partial \dot{\boldsymbol{x}}}\right) = 0 \tag{A.13}$$

The generalization of these results is straightforward assuming a constrained trajectory vector $\boldsymbol{x}(t)$, i.e., $\boldsymbol{v}(t) = \boldsymbol{g}(\boldsymbol{x}(t))$. With the introduction of the modified function f', where $\boldsymbol{\lambda} \in \mathbb{R}^n$ is the *Lagrange multipliers vector*:

$$f'(t, \boldsymbol{x}(t), \dot{\boldsymbol{x}}(t)) = f(t, \boldsymbol{x}(t), \dot{\boldsymbol{x}}(t)) + \boldsymbol{\lambda}\left(\boldsymbol{v}(t) - \boldsymbol{g}(\boldsymbol{x}(t))\right) \tag{A.14}$$

the minimizer $\boldsymbol{x}^*(t)$ satisfies the following set of DAEs:

$$\begin{gathered} \frac{\partial f'}{\partial \boldsymbol{x}} - \frac{d}{dt}\left(\frac{\partial f'}{\partial \dot{\boldsymbol{x}}}\right) = 0 \\ \delta\boldsymbol{x}(t_0)\left(\frac{\partial f'}{\partial \dot{\boldsymbol{x}}}\right) = \delta\boldsymbol{x}(t_1)\left(\frac{\partial f'}{\partial \dot{\boldsymbol{x}}}\right) = 0 \\ \boldsymbol{v}(t) = \boldsymbol{g}(\boldsymbol{x}(t)) \end{gathered} \tag{A.15}$$

A.1.4 Isoperimetric problem

The fourth problem analyzed is the isoperimetric problem which consists in finding the function $x(t)$ that minimizes the integral functional $F(x)$ and, at the same time, satisfies another integral constraint. The problem can be formalized as follows:

$$\begin{aligned} \min_{x(t)} F(x) &= \int_{t_0}^{t_1} f(t, x(t), \dot{x}(t))dt \\ s.t. \quad G(x) &= \int_{t_0}^{t_1} g(t, x(t), \dot{x}(t))dt = c \end{aligned} \tag{A.16}$$

where c is a constant and the boundary conditions are written in the form $x(t_0) = x_0$, $x(t_1) = x_1$. It is also possible to prove that a necessary condition for the minimizer to be a solution to the isoperimetric problem takes place if $x^*(t)$ is an extremal solution of the functional:

$$\int_{t_0}^{t_1} \left(f(t, x(t), \dot{x}(t)) + \lambda g(t, x(t), \dot{x}(t))\right) dt \tag{A.17}$$

in which λ is a constant and its value depends on the considered problem.

Bibliography

[1] Altec. [Online]. Available: https://www.altecspace.it/en

[2] B. Bona, *Modellistica dei robot industriali*, ser. Strumenti per l'ingegneria. CELID, 2002.

[3] Robotics simulation. [Online]. Available: https://www.intorobotics.com/robotics-simulation-softwares-with-3d-modeling-and-programming-support/

[4] J. W. Burdick, "On the inverse kinematics of redundant manipulators: Characterization of the self-motion manifolds," in *Advanced Robotics: 1989*, K. J. Waldron, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 25–34.

[5] B. Siciliano and O. Khatib, *Springer Handbook of Robotics*. Berlin, Heidelberg: Springer-Verlag, 2007.

[6] L. Barinka and R. Berka. (2002, March) Inverse kinematics - basic methods. [Online]. Available: http://old.cescg.org/CESCG-2002/LBarinka/paper.pdf

[7] C. R. Carignan, "Trajectory optimization for kinematically redundant arms," *Journal of Robotic Systems*, vol. 8, no. 2, pp. 221–248, 1991. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.4620080206

[8] A. Nedungadi and K. Kazerouinian, "A local solution with global characteristics for the joint torque optimization of a redundant manipulator," *Journal of Robotic Systems*, vol. 6, no. 5, pp. 631–654, 1989. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.4620060508

[9] K. Kazerounian and Z. Wang, "Global versus local optimization in redundancy resolution of robotic manipulators," *Int. J. Rob. Res.*, vol. 7, no. 5, pp. 3–12, Oct. 1988. [Online]. Available: http://dx.doi.org/10.1177/027836498800700501

[10] Y. Nakamura and H. Hanafusa, "Optimal redundancy control of robot manipulators," *Int. J. Rob. Res.*, vol. 6, no. 1, pp. 32–42, Mar. 1987. [Online]. Available: http://dx.doi.org/10.1177/027836498700600103

[11] H. Hanafusa, T. Yoshikawa, and Y. Nakamura, "Analysis and control of articulated robot arms with redundancy," *IFAC Proceedings Volumes*, vol. 14, no. 2, pp. 1927 – 1932, 1981, 8th IFAC World Congress on Control Science and Technology for the Progress of Society, Kyoto, Japan, 24-28 August 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1474667017637546

[12] K. J. Kyriakopoulos and G. N. Saridis, "Minimum jerk path generation," in

Proceedings. 1988 IEEE International Conference on Robotics and Automation, April 1988, pp. 364–369 vol.1.

[13] L. Tsai and A. Morgan, "Solving the kinematics of the most general six- and five-degree-of-freedom manipulators by continuation methods," *Journal of Mechanisms Transmissions and Automation in Design*, vol. 107, p. 189, 06 1985.

[14] J. Duffy and C. Crane, "A displacement analysis of the general spatial 7-link, 7r mechanism," *Mechanism and Machine Theory*, vol. 15, p. 153–169, 12 1980.

[15] H.-Y. Lee and C.-G. Liang, "Displacement analysis of the general spatial 7-link 7r mechanism," *Mechanism and Machine Theory*, vol. 23, pp. 219 – 226, 1988.

[16] R. Manseur and K. L. Doty, "A robot manipulator with 16 real inverse kinematic solution sets," *The International Journal of Robotics Research*, vol. 8, pp. 75–79, 1989.

[17] D. E. Whitney, "The mathematics of coordinated control of prostetic arms and manipulators," *Journal of Dynamic Systems, Measurement, and Control*, vol. 94, 12 1972.

[18] A. Liegeois, "Liegeois, a.: Automatic supervisory control of the configuration and behavior of multibody mechanisms. ieee trans. syst. man cybern. 7(12), 868-871," *IEEE Transactions on Systems, Man, and Cybernetics - TSMC*, vol. 7, pp. 868–871, 12 1977.

[19] D. N. Nenchev, "Redundancy resolution through local optimization: A review," *Journal of Robotic Systems*, vol. 6, no. 6, pp. 769–798, 1989. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.4620060607

[20] P. Chiacchio, S. Chiaverini, L. Sciavicco, and B. Siciliano, "Closed-loop inverse kinematics schemes for constrained redundant manipulators with task space augmentation and task priority strategy," *International Journal of Robotic Research - IJRR*, vol. 10, pp. 410–425, 08 1991.

[21] G. C. Calafiore and L. El Ghaoui, *Optimization Models*. Cambridge University Press, 2014.

[22] M. Uchiyama, K. Shimizu, and K. Hakomori, "Performance evaluation of manipulators using the jacobian and its application to trajectory planning," *International Journal of Robotic Research - IJRR*, 01 1985.

[23] J. M. Hollerbach and K. C. Suh, "Redundancy resolution of manipulators through torque optimization," *IEEE Journal on Robotics and Automation*, vol. 3, 08 1987.

[24] Boundary value problem. [Online]. Available: http://www.scholarpedia.org/article/Boundary_value_problem

[25] E. Ferrentino and P. Chiacchio, "Redundancy parametrization in globally-optimal inverse kinematics," *The 16th International Symposium on Advances in Robot Kinematics (ARK)*, pp. 47–55, 01 2019.

[26] E. Ferrentino and P. Chiacchio, "Topological analysis of global inverse kinematic solutions for redundant manipulators," *CISM International Centre for*

Mechanical Sciences, Courses and Lectures, pp. 69–76, 01 2019.

[27] E. Ferrentino and P. Chiacchio, "A topological approach to globally-optimal redundancy resolution with dynamic programming," *CISM International Centre for Mechanical Sciences, Courses and Lectures*, pp. 77–85, 01 2019.

[28] A. Guigue, M. Ahmadi, R. Langlois, and M. J. D. Hayes, "Pareto optimality and multiobjective trajectory planning for a 7-dof redundant manipulator," *Trans. Rob.*, vol. 26, no. 6, pp. 1094–1099, Dec. 2010. [Online]. Available: http://dx.doi.org/10.1109/TRO.2010.2068650

[29] A. Dolgui and A. Pashkevich, "Manipulator motion planning for high-speed robotic laser cutting," *International Journal of Production Research*, vol. 47, no. 20, pp. 5691–5715, 2009. [Online]. Available: https://doi.org/10.1080/00207540802070967

[30] J. Gao, A. Pashkevich, and S. Caro, "Optimization of the robot and positioner motion in a redundant fiber placement workcell," *Mechanism and Machine Theory*, vol. 114, pp. 170 – 189, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0094114X16306061

[31] W. B. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality (Wiley Series in Probability and Statistics)*. New York, NY, USA: Wiley-Interscience, 2007.

[32] P. Wenger, "A new general formalism for the kinematic analysis of all nonredundant manipulators," in *Proceedings 1992 IEEE International Conference on Robotics and Automation*, May 1992, pp. 442–447 vol.1.

[33] E. Ferrentino and P. Chiacchio, "On the optimal resolution of inverse kinematics for redundant manipulators using a topological analysis," unpublished.

[34] J. W. Burdick, "On the inverse kinematics of redundant manipulators: characterization of the self-motion manifolds," in *Proceedings, 1989 International Conference on Robotics and Automation*, May 1989, pp. 264–270 vol.1. [Online]. Available: http://dx.doi.org/10.1109/ROBOT.1989.99999

[35] Optimal planning. [Online]. Available: http://ompl.kavrakilab.org/optimalPlanning.html

[36] Introduction to ros. [Online]. Available: http://wiki.ros.org/ROS/Introduction

[37] About ros. [Online]. Available: http://www.ros.org/about-ros/

[38] Esrocos. [Online]. Available: https://cordis.europa.eu/project/rcn/206157/factsheet/en

[39] Ros concepts. [Online]. Available: http://wiki.ros.org/ROS/Concepts#ROS_Computation_Graph_Level

[40] Ros nodes. [Online]. Available: http://wiki.ros.org/Nodes

[41] Ros master. [Online]. Available: http://wiki.ros.org/Master

[42] Ros parameter server. [Online]. Available: http://wiki.ros.org/Parameter%20Server

[43] Ros messages. [Online]. Available: http://wiki.ros.org/Messages

[44] Ros services. [Online]. Available: http://wiki.ros.org/Services

[45] Ros topics. [Online]. Available: http://wiki.ros.org/Topics
[46] Moveit! [Online]. Available: https://moveit.ros.org/
[47] Ompl. [Online]. Available: https://moveit.ros.org/documentation/planners/
[48] Ompl planners. [Online]. Available: http://ompl.kavrakilab.org/planners.html
[49] Kdl. [Online]. Available: http://www.orocos.org/kdl
[50] Trac-ik inverse kinematics plugin. [Online]. Available: https://traclabs.com/projects/trac-ik/
[51] Ikfast. [Online]. Available: http://openrave.org/docs/0.8.2/openravepy/ikfast/
[52] Boost.multiarray. [Online]. Available: https://www.boost.org/doc/libs/1_69_0/libs/multi_array/doc/index.html
[53] Unit testing. [Online]. Available: http://softwaretestingfundamentals.com/unit-testing/
[54] White box testing. [Online]. Available: http://softwaretestingfundamentals.com/white-box-testing/
[55] Googletest official repository. [Online]. Available: https://github.com/google/googletest
[56] Googletest documentation. [Online]. Available: https://github.com/google/googletest/blob/master/googletest/docs/primer.md