SAPIENZA
UNIVERSITÀ DI ROMA

Dottorato di Ricerca in "Automatica e Ricerca Operativa"
Ciclo XXVIII

Motion planning for manipulation and/or navigation tasks with emphasis on humanoid robots

Ph.D. candidate
MARCO COGNETTI

Advisor
Prof. GIUSEPPE ORIOLO

Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti

Laboratorio di Robotica

Abstract

This thesis handles the motion planning problem for various robotic platforms. This is a fundamental problem, especially referring to humanoid robots for which it is particularly challenging for a number of reasons. The first is the high number of degrees of freedom. The second is that a humanoid robot is not a free-flying system in its configuration space: its motions must be generated appropriately. Finally, the implicit requirement that the robot maintains equilibrium, either static or dynamic, typically constrains the trajectory of the robot center of mass. In particular, we are interested in handling problems in which the robot must execute a task, possibly requiring stepping, in environments cluttered by obstacles. In order to solve this problem, we propose to use offline probabilistic motion planning techniques such as Rapidly Exploring Random Trees (RRTs) that consist in finding a solution by means of a graph built in an appropriately defined configuration space. The novelty of the approach is that it does not separate locomotion from task execution. This feature allows to generate whole-body movements while fulfilling the task. The task can be assigned as a trajectory or a single point in the task space or even combining tasks of different nature (e.g., manipulation and navigation tasks). The proposed method is also able to deform the task, if the assigned one is too difficult to be fulfilled. It automatically detects when the task should be deformed and which kind of deformation to apply.

However, there are situations, especially when robots and humans have to share the same workspace, in which the robot has to be equipped with reactive capabilities (as avoiding moving obstacles), allowing to reach a basic level of safety. Here, offline techniques cannot be used and the reaction time (from the detection of the moving obstacle to start an evasive motion) should be as small as possible. This is achieved by making use of closed-form expressions throughout the proposed method, and results in an algorithm suitable for real-time implementation.

The final part of the thesis handles the rearrangement planning problem. This problem is interesting in view of manipulation tasks, where the robot has to interact with objects in the environment. Roughly speaking, the goal of this problem is to plan the motion for a robot whose assigned a task (e.g., move a target object in a goal region). Doing this, the robot is allowed to move some movable objects that are in the environment. The problem is difficult because we must plan in continuous, high-dimensional state and action spaces. Additionally, the physical constraints induced by the nonprehensile interaction between the robot and the objects in the scene must be respected. Our insight is to embed physics models in the planning stage, allowing robot manipulation and simultaneous objects interaction.

Throughout the thesis, we evaluate the proposed planners through experiments on different robotic platforms.

Contents

Chapter 1

Introduction

Robots are starting to conquer the world! They are getting more and more importance in human society, since they can replace humans in dangerous and repetitive tasks but also in services tasks. Nowadays, it is common to use robots in factories (mainly manipulators) or in space missions (e.g., rovers on Mars) but they are starting to appear also in everyday life tasks. Some examples are aerial vehicles (e.g., Amazon wants to use quadrotors for delivering purposes) or mobile robots (e.g., the Roomba robot for cleaning the houses). Robots today are making a considerable impact on many aspects of modern life, from industrial manufacturing to healthcare, transportation, and exploration of the deep space and sea. Human society have always the dream of building autonomous and intelligent machine. Currently, in my humble opinion, we are just one or two steps behind this dream and the next few years will be fundamental for having complete autonomous robots in everyday human lives.

A robot is a device built from humans, whose goal is to take place or help humans, especially in dangerous environments or manufacturing processes. The word "robotics" was derived from the word "robot", which was firstly introduced to the public by Czech writer Karel Čapek in his play R.U.R. (Rossum's Universal Robots), which was published in 1920. The play begins in a factory that makes artificial people called robots, creatures who can be mistaken for humans – very similar to the modern ideas of androids. Karel Čapek himself did not coin the word. He wrote a short letter in reference to an etymology in the Oxford English Dictionary in which he named his brother Josef Čapek as its actual originator. In an article in the Czech journal "Lidové noviny" in 1933, he explained that he had originally wanted to call the creatures "laboři" ("workers", from the Latin word labor). The word robot comes from the Slavic word *robota*, which means labour.

The community officially referred to Isaac Asimov as the first author that uses the word "robotics" in a printed work. Just as anecdote, Asimov published in 1941 a fiction short story named "Liar!" where he was unaware that he was coining the term; since the science and technology of electrical devices is electronics, he assumed robotics already referred to the science and technology of robots. Asimov states that his first work where the word "robotics" was used is "Runaround" (Astounding Science Fiction, March 1942). Since "Liar!" was published ten

months before “Runaround”, the latter is generally referred as the word’s origin. The main contribution from Asimov are the three laws of Robotics (firstly introduced in his play “Runaround”), where it states the three fundamental rules that a robot has to obey

1. a robot may not injure a human being or, through inaction, allow a human being to come to harm;

2. a robot must obey the orders given it by human beings except where such orders would conflict with the First Law;

3. a robot must protect its own existence as long as such protection does not conflict with the First or Second Laws.

These three laws (used also in Asimov’s most famous “Robot series”) have becoming a fundamental reference for the robotics community and are still referred nowadays to indicate what are the fundamental skills that a robot has to have.

Following the ideas of the three laws, there are many definition of a robot. According to the Encyclopaedia Britannica, a robot is “any automatically operated machine that replaces human effort, though it may not resemble human beings in appearance or perform functions in a human-like manner”. Merriam-Webster describes a robot as a “machine that looks like a human being and performs various complex acts (as walking or talking) of a human being”, or a “device that automatically performs complicated often repetitive tasks”. Finally, Oxford Dictionaries defines as “a machine capable of carrying out a complex series of actions automatically, especially one programmable by a computer”.

As it is evident, there is no unique definition accepted from the robotics community. For example, Joseph Engelberger, a pioneer in industrial robotics, said ones: “I cannot define a robot, but I know one when I see one”. In my humble opinion, the main reason for these multiple definitions comes from the merging between the idea of a robot coming from fictions and plays and what a robot really is, i.e. a programmable complex machine.

From a historical point of view, Leonardo Da Vinci designs in 1495 some drawings of a mechanical knight in armour that was able to move its arms, jaw and head. In 1533, Johannes Müller von Königsberg created an automaton iron eagle that was both to fly. The next reference to a robotics project was placed around 1700, when several automatons showed up. These automatons were capable of drawing, flying and even playing music (e.g., the automaton flute player and the digesting duck – powered by weights – from Jacques de Vaucanson in 1737). In the last decades of the 19th century, the first remote controlled systems appeared, mainly torpedos. Some famous examples of those controlled systems are the pneumatic torpedo from John Ericsson, the electric wire torpedos from John Louis Lay and Victor von Scheliha. Archibald Low (creator of “radio guidance systems”) worked on guided rockets and planes during the First World War. In 1917, he demonstrated a remote controlled aircraft to the Royal Flying Corps and in the same year he built the first wire-guided rocket.

After the formulation of the above-mentioned robotics laws from Asimov, the robotics performed another important step in its conceptual formulation when Norbert Wiener dictated in 1948 the principles of cybernetics, the basis of practical robotics.

An important year for the robotics community is 1950 when Alan Turing in his paper "Computing Machinery and Intelligence" stated "I propose to consider the question, Can machines think?". He is the creator of the Turing machine, a mathematical model that manipulates symbols on a strip of tape according to a table of rules. The goal of the Turing machine is to answer to the following questions: i) does a machine exist that can determine whether any arbitrary machine on its tape is "circular" (e.g. freezes, or fails to continue its computational task) ii) does a machine exist that can determine whether any arbitrary machine on its tape ever prints a given symbol[1]. Turing stated that a computer, by shuffling symbols as simple as the binary code (made only by zeros and ones), are able to replicate any mathematical deduction. Another important contribution from the same author was the "Turing test", a test aimed to check if a machine is intelligent or not. Even if the test was formulated in the next years (both for problems in the original formulation and for the definition of an intelligent machine), it gave the first definition of an intelligent machine and enabled the community to think about what an intelligent machine is and how it can be defined.

Thanks to the industrial and computers improvements, the numerical control machines and teleoperated manipulators started to appear in the early years of 1950s. The former are high-precision machines capable of performing repetitive tasks over and over again while the latter are rigid bodies connected by joints directly controlled by a human. Just in the 1960s and 1970s, the features of the two above-mentioned categories were merged, bringing to the first installed industrial robot from KUKA (1973). In 1974, ABB (a Swiss company leader in building manipulator robots) builds the world's first microcomputer controlled electric industrial robot, named IRB 6, that was delivered to a small mechanical engineering company in southern Sweden. In the same year, David Silver designed "The Silver Arm", a robot capable of replicating the movements of the human hand. It was equipped with touch and pressure sensors that can be processed via a computer.

The real breakthrough for the robotics world came in the 1980s, when theoretical contributions coming from control theory (linear and non-linear systems and control laws), electronics (integrated circuits), computer science (digital computers) and mechanics gave the opportunity to program and design robots. These robots (mainly manipulators) had a wide usage in general industry (e.g., chemical, electrical and food industries). In the same years, the first formal definition was given as "the intelligent connection between perception and action" [102]. This definition is particularly important because it summarizes, in few words, the core components of a robot. In fact, the action of a robotic system is to transfer commands to a locomotion apparatus (wheels, legs, propellers, crawlers) and/or to a manipulation apparatus (hands, end-effectors,

[1]Definition taken from https://en.wikipedia.org/wiki/Turing_machine

arms) through actuators, that animate the mechanical structure of the robot. In addition, perception is achieved through sensors that can provide information about the internal state of the robot (proprioceptive sensors) or information about the surrounding environments (exteroceptive sensors). The intelligent connection is demanded to a planning and control architecture that, given the information coming from the sensors and available model of the robot, exploits the commands to be given to the actuators. In this context, Takeo Kanade created, in 1981, the first "direct drive arm", a manipulator arm whose motors were contained in the robot itself, avoiding long external cables. In 1984, Wabot-2 was released. It had 10 fingers and its aim is to play an organ. The Carnegie Mellon University developed, in 1989, two chess playing programs (HiTech and Deep Thought) that were able to defeat chess masters (this was a real contributions in the artificial intelligence field). Just as anecdote, Deep Thought was the father of Deep Blue, the famous computer algorithm developed by IBM that defeated in 1996 the chess world championship Garry Kasparov.

In 1986, Honda began its research in humanoid robotics, whose aim is to develop robots that are able to interact with humans. This is a really relevant fact since it was the first example of modern humanoid robots. Three years later, a walking robot named Genghis was developed from MIT. It was famous for the way it walks. It was so famous that its walking was defined as the "Genghis' gait". In the same year, Rodney Brooks and A. M. Flynn (from MIT) published the paper "Fast, Cheap and Out of Control: A Robot Invasion of the Solar System" in the Journal of the British Interplanetary Society. This paper makes the idea that a rover should not be a huge, expansive robot but that it can be accessed to the average people by building many (little) cheap ones.

In 1993, a 8-legged walking robot named Dante (from Carnegie Mellon University) was developed for descending into Mt. Erebrus, Antarctica. The idea was to collect data from a harsh environment, similar to what it might found into another planet (its final mission was in space robotics). However, the mission failed when, after a short 20 foot decent, Dante's tether snaps dropping it into the crater. In the next year, DanteII was built and it successfully descent the Alaskan volcano Mt. Spurr. In 1994, the robot RoboTuna was built at MIT. Its aim was to study and replicate how fishes swim. As the name suggests, it was designed to replicate the swimming behaviour of a blue fin tuna. In the same year, the Cyberknife offered an alternative treatment in the surgery performed by human doctors. In 1996, Human developed the P2 (Prototype Model 2) humanoid robot. It had a more human-like walking behaviour w.r.t. its predecessors. It was also the first self-regulating, bipedal humanoid robot. The 1990s was also an important decade for space robotics. In fact, the Sojourner rover shut down after 83 operating days (with a starting expectation of just seven days). It did semi-autonomous operations on the Mars surface as part of the PathFinder space mission. It was equipped with an obstacle avoidance algorithm to autonomously navigate on the Mars surface. It was also able to plan motion to study the surface of the planet. In 1998, Honda released P3, while in the next year Sony produced AIBO, a robotic dog able to interact with humans. The first models had a so huge impact that they were sold in

the first 20 minutes in Japan. This robot was also used a lot in research projects. As example, the first version of the RoboCup[2] made usage of this particular robot. In the same year, LEGO developed the MINDSTORMS robotic development line. It enables the user to build his/her own robot by composing LEGO bricks.

In 2000, the famous ASIMO was introduced from Honda. It was able to run, walk, interact with humans and the environment. It was also equipped with facial, voice and posture recognition. ASIMO is still a work-in-progress project and it is aimed to be a personal assistant. In 2001, the Canadarm2 (a manipulation arm) was launched in orbit and reached the International Space Station. In 2002, Roomba from iRobot appeared for the first time in the market, as a robotic vacuum cleaner device. In 2003, Spirit and Opportunity rovers was launched by NASA whose destination was Mars (the latter successfully landed on the planet surface the next year and it is still operating). Robonaut 2 is another example of space robotics. It was the first humanoid in space and its goal was to teach to engineers how dexterous robots behave in space. In 2005, Cornell University created the first robot that is able to self-replicate. It is made by a set of cubes that are able of attaching or detaching and the robot is able to create copies of itself. In 2007, Aldebaran Robotics released NAO, a small humanoid used also in the RoboCup challenge from 2008 to the last competition. NAO will be described in details in 4.4.1, since it is the main robotic platform used in this thesis.

Self-driving cars are another example of nowadays increasing field of robotics application. Google is currently working on this topic and there are some prototypes of autonomous cars that are driving in real-life cities.

There are other important dates and facts that should be mentioned in this brief overview of the robotics history. As example, there is a huge amount of omitted facts regarding aerial vehicles and medical robotics. Since this thesis is focused on humanoid and manipulator robots, the idea was to privilege historical details about these two kinds of robots. However, a complete historical description is out of the scope of this thesis. The aim of this description is to give an idea of how the concept and the features of a robot from the society changed over the years. Just few decades ago, robots were just seen as machines used in factories and out from people lives. Nowadays, they are starting to move out from factories. Some examples are the above-mentioned Roomba vacuum cleaner robot and the Parrot ArDrone, an aerial vehicle available on the market. For this reason, it is really important nowadays to build robots that are able to autonomously plan and execute motion, given a task. This can be the first, rough definition, of the motion planning problem, core topic of this thesis. In other words, the problem we want to face in this thesis is how to generate the movements for a robot that must execute an assigned task (e.g., opening a door).

Since this thesis faces the motion planning problem using different robotics platforms, it is important to define the different types of existing robots.

[2]Currently its website is: http://www.robocup2015.org/

Figure 1.1 Some examples of manipulator robots. From left to right: an industrial robot from ABB, the KUKA LWR and the KINOVA JACO arm.

1.1 Manipulators

A manipulator can be defined as "a sequence of rigid bodies (links) interconnected by means of articulations (joints)" [104]. In this section, as well in the next ones, we give a brief overview about the fundamental principles of robotics. An expert can easily skip this chapter while a reader can find more details about this topic in [102, 104].

A manipulator is composed by an arm that provides mobility, a wrist that gives to it the dexterity and an end-effector to fulfil the task given to the robot. Some examples of this kind of robot is given in Figure 1.1. Its structure is made by the serial of open kinematic chains. An kinematic chain is referred to open when there is a unique sequence of links that connects the two terminals of the chain. On the contrary, the chain is named closed when there is a loop in the chain itself.

The movements of a manipulator (or a robot as general reference) are possible through joints, i.e. the articulation between two consecutive links. Each joint provide to the robot a Degree Of Freedom (DOF). There are two main joint types: prismatic and revolute. A prismastic joint provides a translational motion between the two consecutive links, while a revolute joint provides a rotational motion between the two consecutive links. In an open kinematic chain, the number of joints and DOFs are equal each other, while they are not equal in case of a closed kinematic chain (in details, the DOFs are less than the number of the joint due to the closure constraint).

The *workspace* of the robot is the portion of the environment that is reachable from the end-effector of the robot. It heavily depends on the structure of the robot, as well from where the joints are placed on the robot itself.

1.2 Mobile robots

A mobile robot is an automatic machine that is capable of locomotion. The industrial robots (like the ones presented in the previous section) are fixed-based robots capable of very accurate and repetitive capabilities. Their main limitation relies in the fact that they have a very limited workspace. On the contrary, mobile robots are able to roam freely in an unstructured, uncertain

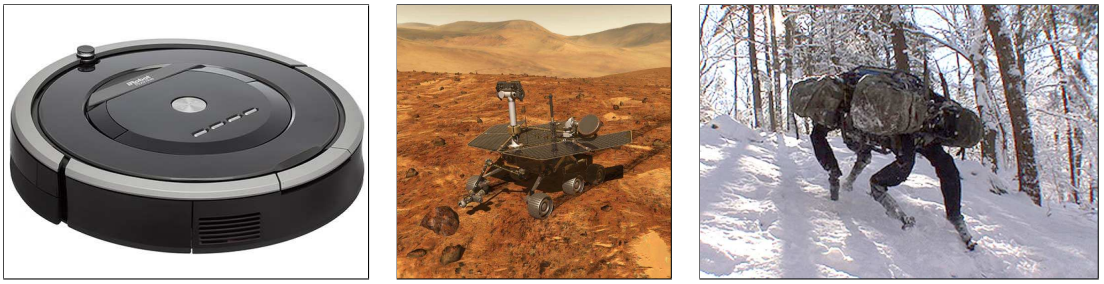


Figure 1.2 Some examples of mobile robots. From left to right: the Roomba vacuum cleaner robot from iRobot, the Spirit rover from NASA and the legged BigDog robot from Boston Dynamics.

and dynamic world. In other words, a mobile robot has a mobile base that enable it to freely navigate in the environment. Some examples of mobile robots are depicted in Figure 1.2.

A mobile robot can be used both in service (if the environment is structured) and field robotics (if the environment is unstructured). In the first category, robots are used for transportation (e.g., movements in factories), customer assistance (e.g., autonomous guide in museum) or cleaning (e.g., the Roomba robot depicted in Figure 1.2). In the other category, a mobile robot is used for exploration (e.g., the rovers used in space robotics), monitoring (e.g., aerial vehicles are used for monitoring forests and seas), rescue, agriculture, transportation, entertainment (e.g., the ArDrone from Parrot).

We already mentioned that the main difference between fixed-base and mobile robots is the ability of the latter to move and explore the environment. This ability comes with a price that is the spreading of the following problems

- localization: the robot should be able to localize in the environment it is exploring using or not an initial guess or a map;
- planning: plan the motion of the robot by defining a path or a trajectory, respectively;
- motion control: control the robot to track the desired behaviour defined in the previous point;
- perception: perceive the environment from sensors and process data coming from those sensors.

Figure 1.3 summarizes the main blocks needed to control a mobile robot. Each block correspond to the solution of the above-mentioned problems.

As it is evident from Figure 1.2, there are several different kinds of robots that can be defined as mobile robots. In my opinion, they can be group in the following main categories

- underwater robots;
- aerial robots;
- wheeled robots;

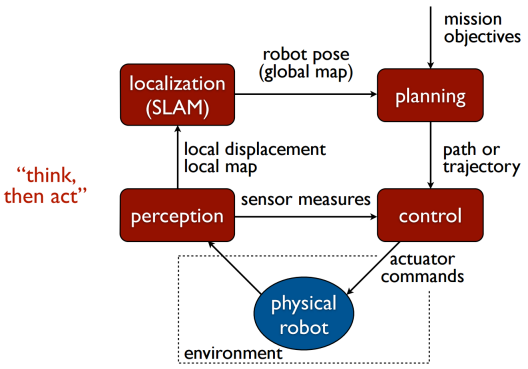


Figure 1.3 The principal blocks needed to control a mobile robot: localization, planning, perception and control (courtesy of Oriolo's slides).

- legged robots.

Even if each of the above-mentioned group are really interesting and promising from a research point of view (and it would be worthwhile to describe all of them in details), we will focus on the last two categories, since the planning algorithms presented in this thesis will be applied on these kinds of robots.

1.2.1 Wheeled robots

A wheeled robot is typically formed by a main body (or *chassis*), and some wheels. There are three types of wheels that a robot can be equipped with: *fixed wheel*, *steerable wheel*, *caster wheel*. The fixed wheel can rotate about a single axis and it is usually not actuated. The wheel is rigidly attached to the robot and its orientation is constant with respect to the body whose it is attached. The steerable wheel has two main rotation axes: one is the same as the fixed wheel while the second is orthogonal. This second rotation axis allows the wheel to have a different orientation w.r.t. the main body of the robot. The caster wheel has two axes of rotation, as the steerable wheel. Unlike the steerable wheel, the vertical axis does not meet the center of the wheel but is has an offset. The latter enables the wheel to automatically rotate when the robot is rotating, letting the wheel to quickly align with the direction assumed by the main body of the robot.

By combining these wheel types and modifying their positions along the robot's body, several kinematic structures can be defined. The most famous and common one is the *differential drive*, where there two fixed wheels (typically actuated) that share the same axis of rotation. The wheel set is completed by a caster wheel (typically not actuated) whose main contribution is to provide statical balance to the structure. As example, please refer to Fig. 1.4. An alternative kinematic structure is the *car-like*. It has two fixed wheels on a rear axle and two steerable wheels on a front axle. It embeds two motors: the first (as in the differential drive scheme) is the one that provides traction while the second steers the two wheels in the front. The last structure we would like to mention is the omnidirectional one. In this kinematic structure, three wheels are placed around a

Figure 1.4 Examples of different wheeled robot types. From left to right: differential drive, car-like and omnidirectional wheeled robot.

circle, having 120° of angle offset. There is one motor in each wheel that provides to the structure the possibility to move instantaneously along any Cartesian direction. There exist other types of wheels. A remarkable one is the Macanum (or Swedish) wheels, as the ones in rightmost robot in Fig. 1.4. They consist in a fixed wheel with passive rollers, placed around a circular rim. The axis of rotation of each roller has an inclination angle of 45° w.r.t. the plane of the wheel. Usually, three or four Swedish wheels are employed for a robot, providing omnidirectional capabilities to the robot itself.

In the rest of this thesis, we will refer to the differential drive as the kinematic structure for a wheeled robot.

Any wheeled vehicle is subject to *kinematic constraints* that reduces its mobility while leaving the possibility to reach any point in the workspace of the robot by composing appropriate maneuvers. As example, let think about a driving car. Everybody knows that a car cannot move along the orthogonal to its forward direction. For this reason, it is important to analyze in details the constraints that apply to a wheeled robot.

Let define $\boldsymbol{q}$ as the *configuration* of the robot and as $\mathcal{C}$ (here $\mathbb{R}^n$) its *configuration space*. The robot will be subjected to several constraints that can be summarized into the form

$$h_i(\boldsymbol{q}) = 0 \forall i \in [1, \dots, k < n].$$

These constraints are called *holonomic* or *integrable* constraints. Their effect on the robot dynamics is to reduce the space of accessible configurations. A trivial solution to solve the holonomic constraints would be to use the implicit function theorem for expressing the k coordinates in function of the remaining $n - k$ coordinates. Although this is a valid, this is a local solution and it might introduce singularities.

There is another type of constraints that involves both the coordinates and its derivative (velocity). It can be expressed as

$$a_i(\boldsymbol{q}, \dot{\boldsymbol{q}}) = 0 \forall i \in [1, \dots, k < n].$$

These kind of constraints are called *kinematic* and they constraint the admissible motion of the robot by reducing the set of velocities that can be attained at each configuration. Each constraint

is usually expressed in Pfaffian form

$$\boldsymbol{a}_i^T \dot{\boldsymbol{q}} = 0,$$

with $\boldsymbol{a}_i$ assumed to be smooth. Stacking the single constraint, the complete Pfaffian form is

$$\boldsymbol{A}^T \dot{\boldsymbol{q}} = 0.$$

It is obvious that if a robot has k holonomic constraints, then it embeds also k kinematic constraints

$$\frac{dh_i(\boldsymbol{q})}{dt} = \frac{\partial h_i(\boldsymbol{q})}{\partial \boldsymbol{q}} \dot{\boldsymbol{q}} = 0, \forall i \in [1, \dots, k < n].$$

It is important to underline that the opposite is not true in general: k kinematic constraints do not correspond to k holonomic constraint since some of those can be not integrable or *non-holonomic*. A non-holonomic constraint modifies the mobility of the robot, but in a different way w.r.t. a holonomic constraint. To convince the reader, if a Pfaffian constraint is integrable, it can be expressed as $h_i(\boldsymbol{q}) = c$, with c an integration factor. This means that there is a loss of accessibility in the configuration space, since the motion in the configuration space $\mathcal{C}$ is limited to a particular level surface of the function h_i. In case the function $h_i(\boldsymbol{q})$ is a dimension d, the surface has dimension $n - d$. On the contrary, if the constraint $h_i(\boldsymbol{q})$ is non-holonomic, the velocity are constrained to be in a subspace of dimension $n - d$, where d is the dimensionality of the constraint. In this case, there is no accessibility loss in the configuration space for the robot but the number of DoFs are reduced to $n - d$ while the generalized coordinate are still the same as in the original problem and they cannot be reduced even in a local representation of the configuration space. In other words, if a system has k non-holonomic constraint ($h(\boldsymbol{q})$) it has access to the whole configuration space $\mathcal{C}$ but its velocity is constrained to lie in a subspace of dimensionality $n - k$, i.e. the null space of $h(\boldsymbol{q})$.

Assume to have a system with k Pfaffian constraints. As stated before, this means that the system have access to the whole configuration space $\mathcal{C}$ but its velocity are constrained in a subspace of dimensionality $n - k$, i.e. the null-space of $\boldsymbol{A}^T(\boldsymbol{q})$. The same system can be expressed in the following form

$$\dot{\boldsymbol{q}} = \sum_{z=1}^{n-k} \boldsymbol{g}_z(\boldsymbol{q}) u_z = \boldsymbol{G}(\boldsymbol{q}) \boldsymbol{u},$$

with $\boldsymbol{g}_z(\boldsymbol{q})$, $z \in 1, \dots, n - k$ is a base vector of the null-space of $\boldsymbol{A}^T(\boldsymbol{q})$ and $\boldsymbol{u}$ is the input vector.

As example of application of what we presented up to this point, there is the well-known *unicycle* model. It is composed by a single orientable wheel. Its configuration $\boldsymbol{q}$ is defined as $\boldsymbol{q} = [x\ y\ \theta]^T$ where (x, y) are the Cartesian coordinates of the contact point between the wheel and the ground (in world frame) while θ is its heading w.r.t. the x-axis. It has a single constrain

(named pure rolling constraint) that is expressed as a Pfaffian form

$$\dot{x}\sin\theta - \dot{y}\cos\theta = [\sin\theta \ -\cos\theta \ 0] \ \dot{\boldsymbol{q}} = 0. \tag{1.1}$$

In simple words, this constraint imposes a zero velocity for the contact point along the orthogonal plane. This constraint is obviously non-holonomic, since it does not imply any loss of reachability for the system but it constrains the velocity that can be apply on the system itself. The same expression in eq. (1.1) can be rewritten as

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos\theta \\ \sin\theta \\ 0 \end{pmatrix} v + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \omega = \begin{pmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} v \\ \omega \end{pmatrix}, \tag{1.2}$$

that is the kinematic model of the unicycle. Moreover, the input v and ω has a physical interpretation. The former is the driving velocity while the latter is the angular velocity along the vertical axis of the contact point, respectively. Computing the Lie bracket over the two columns of the last matrix in eq. 1.2, it can be demonstrated that these two columns and their Lie bracket are linearly independent. Then, the unicycle is controllable with a degree of non-holonominity equal to 2.

If one looks to the unicycle model in a strict manner (i.e., a robot composed by just a wheel), it is obvious that it has a real serious problem regarding its stability. However, there are many robotic system whose kinematic model is equivalent to a unicycle. The differential drive presented in this section is an example of such systems. Regarding this robot, denote with (x, y) the position of the midpoint between the wheels w.r.t. a world reference frame and denote with θ the common orientation of the fixed wheels. The kinematic model in eq. 1.2 applies also in this case, if the inputs are expressed are

$$v = \frac{r(\omega_R + \omega_L)}{2} \qquad \omega = \frac{r(\omega_R - \omega_L)}{d},$$

where ω_R and ω_L are the angular speed of the left and right wheel, respectively. In addition, d is the distance between the wheels centres and r is the radius of the wheels.

This model will be useful at the end of this thesis, when the problem of the rearrangement planning is presented. The reader is referred to [104] for further details about the kinematics of mobile robots.

1.2.2 Legged robots

A legged robot, as the name suggests, is a mobile robot equipped with a variable number of legs. Even if bipedal legged robots will be the main topic of this thesis, it is important to mention that there are other types of legged robots. In particular, quadrupedal robots and hexapod robots

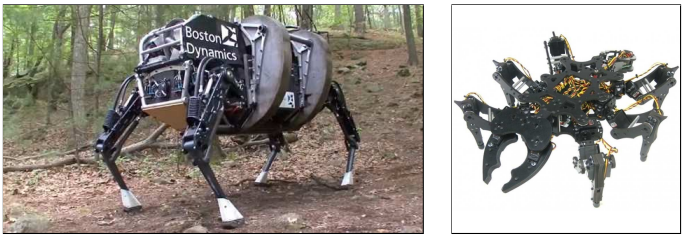


Figure 1.5 Examples of different legged robots. Left: quadrupedal robot; right: hexapod robot.

are one examples of those, as depicted in Figure 1.5. The main difference between the above-mentioned kinematic structures and a bipedal robot is in the way they maintain balance during walking. Having a greater number of legs, they have a wider choice of foot placing to maintain static balance.

In this thesis we will focus on bipedal robots and, in particular, on humanoid robots. A humanoid robot is, by definition, a robot whose shape resemble a human body. This kind of robot is attracting more and more attention by the robotics community for a variety for reasons. The first is an historical reason: humans were always attracted to create a human-like robot. The reader can be convinced of that thinking about movies and plays. Starting from the Capek's play "R.U.R." mentioned in the previous section to some movies such as "Bicentennial Man" (directed by Chris Columbus, 1999) and "I, Robot" (directed by Alex Proyas, 2004)[3], it is clear that, if one asks to a human how he/she imagines his/her ideal robot, there is an high probability that he/she will reply by designing a humanoid robot. Moreover, it is well-known that people are more inclined to accept a human-like robot w.r.t. a machine-like one (e.g., a manipulator or a wheeled robot). In my opinion, this is due to the fact that people, unconsciously, find a "common point" in the physical appearance of the robot and, for this reason, they are more prone to accept it in every-day life activities. Finally, humans are social animals that generally like to observe and interact with one another. Then, a robot whose appearance looks like a human has an higher probability for a social interaction, if compared to other robotic platforms.

Second, a humanoid robot might be able to accomplish, in principle, every task that a human can fulfil. In addition, they can be employed in activities where other kinds of robots cannot be used. Consider the case where a robot should perform several tasks in an airplane cargo having different levels, in an environment where the robot has to share its workspace with humans (that will be treated as research case in this thesis). Here, a legged robot might outperform other robotic platforms since it can handle stairs in a human-like way, without introducing any external infrastructure such as ramps that would be needed for a wheeled robot.

Third, humanoid robots are challenging from a research point of view due to the high number of DOFs with which they are equipped with. They are typically redundant in performing a task and the management of the redundancy is fundamental to achieve a human-like accomplishment

[3]There would be hundreds of other movies that confirm the same concept.

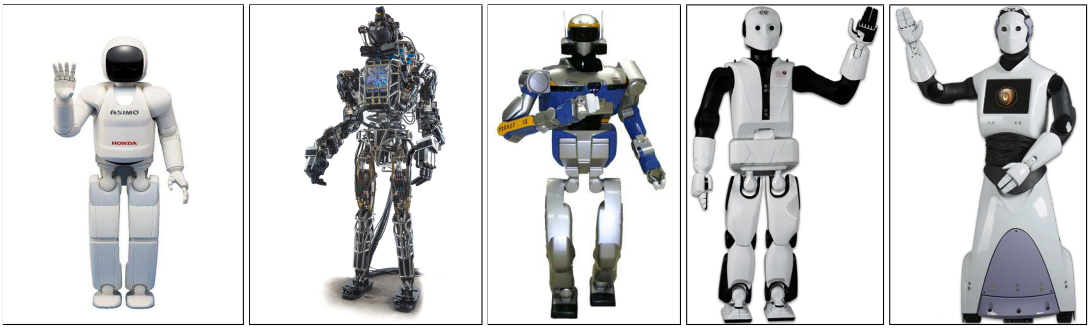


Figure 1.6 Examples of different humanoid robots. Left to right: Asimo from Honda; Atlas from Boston Dynamics; HRP-2 from METI; REEM-C from PAL Robotics; REEM from PAL Robotics.

of the task. Furthermore, the maintenance of the equilibrium during locomotion should be ensured at all time and it is not a trivial problem, mostly if compared with other robotic platform where the problem of balance is not so central. With that we do not want to assert that the humanoid is the best robotic platform but that there are cases in which it is more suitable to use these robots, as in the airplane scenario described above.

Today, there exist humanoid robots having different sizes and shapes. Even the number of DOFs with which they are equipped with can vary but there are some common points that defines what is a humanoid robot: a head, two arms and a torso. Typically they have also two legs, like the first four robots depicted in Fig. 1.6, but there is a fuzzy line in the definition of a humanoid robot. In fact, someone defines as humanoid robot also the last robot robot in Fig. 1.6 where the legs are replaced by a mobile platform. In this case, the locomotion problem is completely different and the balance constraint (formally defined in the next chapter) is easier to be solved, since the Center of Mass (CoM) of a mobile base is by definition in the support polygon defined by the base itself. Then, a mobile robot is typically stable (at least statically). In this thesis we will focus on legged humanoid robots.

The rest of this thesis is organized as follows.

- Chapter 2 is devoted to the introduction to the motion planning problem, core of this thesis while Chapter 3 introduces the sampling-based motion planning methods;
- Part I describes a framework for solving the task-constrained motion planning problem for a humanoid robot. In particular, Chapter 4 describes the basic version of the planner, while Chapter 5 presents a modified version of the planner that generalizes the previous one and allows a generic representation of the assigned task (a task can be formalized even as the combination of tasks of different nature, e.g., manipulation and navigation tasks). Finally, Chapter 6 uses the planner in Chapter 5 in a framework that is able to deform the task, if it is needed;
- Part II handles the case in which the humanoid robot dows not have the time for running an offline planner, as the ones in Part I. Here, an evasive motion should be executed as fast

as possible, in order to prevent a collision with a moving obstacle (e.g., a human). This is a basic level of safety with which each humanoid should be equipped with;

- Part III is devoted to solving the rearrangement planning problem. This problem is particularly important in case the robot has to fulfil manipulation tasks. Robot-centric and object-centric primitives are combined in order to find fast solutions to a variety of scenarios;
- Chapter 9 ends the thesis with some considerations about the proposed planners and some hints about future works.

Chapter 2

Motion planning

The main problem faced in this thesis is the *motion planning* problem. The chapter starts with a brief introduction about this problem. Then, its particularization in presence of a task is presented. We will refer to this as *task-constrained motion planning* problem. Finally, the case of humanoid robots is discussed. The reader is referred to [76, 79, 104] for additional information about the concepts described in this chapter.

2.1 The motion planning problem

Informally, the motion planning problem can be defined as planning the motion of the robot in such a way it is able to fulfil an assigned task while *avoiding collisions* with obstacles that are in the environment. In this really rough definition of the planning problem, there are some keywords that should be analized in more details. First, a robot operates in a workspace where there might be obstacles. In particular, obstacles might be *static* (i.e., they do not move during the execution of the motion), *moving* (i.e., they move during the execution of the motion but their motions are known in advance by the motion planner) and *dynamic* (i.e., they move during the execution of the motion and their motions are not known in advance by the motion planner). As example, consider the scenario in Figure 2.1 and assume one wants to plan the motion of the robot on the left. In this scenario, there are static obstacles (e.g., central body of the robot and its vision system), moving obstacles (e.g., the other robot might move in a predefined way such that it can be taken into account in a planning stage) and dynamic obstacles (e.g., the human whose motion is not predictable). In such a situation, the robot should plan its motion in such a way it is able to accomplish a task (e.g., pick and place objects in the example of Figure 2.1) while avoiding collisions with all the obstacles in the environment.

In case there are only static and moving obstacles in the environment, the planning is referred to as *off-line* planning. In case there are also dynamic obstacles, then the planning is said to be *on-line*. In the former case, the robot can perform all the computations before executing the motion; in the latter case, the planning should be performed *on-line*, by means of on-board sensors. Just

Figure 2.1 Example of a scenario where the motion planning is fundamental for fulfilling the pick and place task.

as reminder, in this thesis, we will first face the off-line motion planning problem and then we will extend our framework towards the on-line motion planning.

Moving to a more formal definition, a *robot* $\mathcal{R}$ is made by a single rigid body (as in the case of a mobile robot) or by a multiple kinematic chains whose base is fixed (as in the case of a manipulator) or mobile (as in the case of a humanoid robot). The robot moves in a Euclidean *workspace* $\mathcal{W}$. The description of the environment is completed by the *obstacle set* $\mathcal{O} = \{\mathcal{O}_1 \cup \cdots \cup \mathcal{O}_p\}$, with p the number of obstacles in the environment. The strongest assumption in motion planning is that the geometries of the robot $\mathcal{R}$ and the obstacles $\mathcal{O}$ are assumed to be known in advance. Moreover, even the poses of the obstacles in $\mathcal{O}$ w.r.t. $\mathcal{W}$ are supposed to be known. Even if these assumptions seem confining, the information required are retrievable via a scan of the environment, that it is possible to obtain through exteroceptive sensors of the robot. In other words, the motion planning assumes that there exists an external module that provides the information described above, but how to build this module is out the scope of this thesis.

In its basic formulation, the motion planning problem can be described as follows: find a *path* (defined as a sequence of postures) for a robot $\mathcal{R}$ that has to move from a given initial to a given goal posture of $\mathcal{R}$ in $\mathcal{W}$, while avoiding collisions between $\mathcal{R}$ and $\mathcal{O}$. In case a path does not exist, report a failure. This formulation of the motion planning problem is known in literature as *piano mover's problem*. An implicit assumption of this problem is that the robot is not subjected to any kinematic constraint, i.e., it is a free-flying system. Obviously, the above-mentioned formulation has several assumptions that may be not applicable for a real system: first, the free-flying assumption does not hold in case of non-holonomic constraints, such as for mobile robots. Second, manipulation tasks cannot be included in this formulation, since a contact with obstacles is required in order to manipulate objects in the environment. Finally, the assumption that the robot is the only entity that is moving in the environment limits the formulation to static obstacles, without having the opportunity to consider moving obstacles (such as humans or other robots). All these assumptions are introduced to reduce the problem to a pure geometrical path planning problem. In order to generalize the previous formulation, the concept of *configuration space* should be introduced.

2.2 Configuration space

The configuration space is the set of the possible transformations that could be applied to the robot. Usually, it is composed by the Cartesian product of the following spaces

- Cartesian coordinates: they describe the Cartesian position of selected points on the robot;
- angular coordinates: they describe the orientation of bodies of the robot.

As example, the configuration space for a fixed-base manipulator having n joints can be defined as a subset of $(\mathbb{R}^3 \times SO(3))^n$. If the dimension of the configuration space is n, then a configuration can be defined as $\boldsymbol{q} \in \mathbb{R}^n$. However, this is valid just locally. In fact, the configuration space $\mathcal{C}$ can be more complex than a Euclidean space. When the configuration of a robot includes angular coordinates (as in the case of a humanoid), its configuration space is properly described by a manifold, i.e., a space in which a neighbourhood of a selected point has a correspondence with $\mathbb{R}^n$ through a homeomorphism.

As example, assume one wants to define a configuration space for a 3D rigid body ($SE(3)$). We choose this example since it will be the main type of entities we will treat in this thesis. The configuration space is $\mathbb{R}^3 \times \mathbb{RP}^3$. The reason is straightforward: three dimensions from translation; three dimensions from rotation. The real challenge is how to define the representation for the angular part. A first choice would be to select three angles to represent the orientation, such as roll, pitch and yaw. This choice is convenient since it is easy to describe and visualize a kinematic chain using this parametrization. Furthermore, it is efficient from a computational point of view, thanks to linear algebra libraries. However, this comes with a price. As everybody knows, any Euler angle representation has a singularity, i.e., there exist multiple angle values that refer to the same rotation matrix. These problems destroy the topology, that might bring to both theoretical and practical issues. Obviously, other choices are possible, such as unit quaternions. It is a four-dimension vector with the constraint that its norm is equal to 1. This representation is indeed more general than the Euler angles one but it is not unique. If one denotes with $\boldsymbol{h}$ the quaternion and with $\boldsymbol{R}$ its rotation matrix, it can be proved that $\boldsymbol{R}(\boldsymbol{h}) = \boldsymbol{R}(-\boldsymbol{h})$. In other words, this means that a rotation of θ about the axis d is equivalent to a rotation of $2\pi - \theta$ about the axis $-d$. Fortunately, this is the only problem is using quaternions, that can be fixed by the identification trick. As mentioned before, a quaternion is an element of $\mathbb{R}^4$, with the unitary constraint. This projects the quaternion on $\mathbb{S}^3$. Using identification, declare $\boldsymbol{h} \sim -\boldsymbol{h}$ for all unit quaternions. With this assumption the antipodal points[1] of $\mathbb{S}^3$ are uniquely identified. It can be proved that, if the antipodal points are known, $\mathbb{RP}^n \cong \mathbb{S}^n / \sim$. Finally, $SO(3)$ can be defined as the set of all lines through the origin of $\mathbb{R}^4$. As summary, the previous discussion was for saying that $SO(3)$ can be parameterized by picking points in $\mathbb{S}^3$. This is an analogy with $SO(2)$

[1] To give an idea of what an antipodal point, let focus on the sphere case. The antipodal point of a point on the surface of a sphere is the point which is diametrically opposite to it, i.e., it is situated by intersecting the line drawn from the one to the other passes through the center of the sphere and forms a true diameter. Source: Wikipedia.

where the same concept can be applied by picking points in $\mathbb{S}^1$ and by ignoring the antipodal identification problem for $SO(3)$.

The reason for which the quaternion representation was introduced relies in its ability to compose transformations, that is really simple. Then, quaternion multiplications can be used instead of matrix computed (even more efficient from a computational point of view). Having defined a valid representation for $SO(3)$, the derivation of a representation for $SE(3)$ is trivial. A matrix in $SE(3)$ is an homogeneous matrix in the form

$$\begin{pmatrix} \boldsymbol{R} & \boldsymbol{v} \\ \boldsymbol{0} & 1 \end{pmatrix} ,$$

with $\boldsymbol{R} \in SO(n)$ and $\boldsymbol{v} \in \mathbb{R}^n$. Since it is well known that $SO(3) \cong \mathbb{PR}^3$ and the translation component can be chosen freely, the *C-space* (that is the Configuration space) for a rigid body in 3D (that is translates and rotates in $\mathbb{R}^3$) is defined as

$$\mathcal{C} = \mathbb{R}^3 \times \mathbb{PR}^3 ,$$

that is a manifold of dimension six. As one might notice, the dimension of this manifold is equal to the number of DOFs of a free-flying system that navigates in a three-dimensional space. Obviously, the configuration space can be generalized in case of multiple bodies as $\mathcal{C} = \mathcal{C}_1, \dots, \times \mathcal{C}_m$, with m the number of rigid bodies that composes the kinematic chain that one is considering. Note that there is no general rule for deriving the C-space for a rigid body and the definition of the configuration space has to be faced case by case. In fact, the whole range can be not reachable for some kind of joints. As example, consider a revolute joint, whole range is rarely equal to $[0, 2\pi)$. If this happen, the C-space is homomorphic to $\mathbb{R}^1$ instead of $\mathbb{S}^1$, obtaining a similar situation than in the roll, pitch, yaw angles, that is the same arm configuration can be expressed in multiple ways. In this case, the configuration space of a revolute joint is typically $\mathbb{R}$, as we will use in the next chapters.

2.3 Configuration space obstacles

Having defined the configuration space $\mathcal{C}$, we can turn our attention to the obstacles. In particular, the configuration space obstacles $\mathcal{CO}$ is the image of the obstacles, defined in the workspace $\mathcal{W}$ of the robot, in the configuration space $\mathcal{C}$. In other words, it is the set of all the configurations of the robot in which it collides with itself (self-collisions) or with an obstacle in the environment $\mathcal{W}$. Assume to have an obstacle $\mathcal{O}_i, i = 1, \dots, l$ in $\mathcal{W}$. The image $\mathcal{CO}_i$ of the obstacle in $\mathcal{C}$ is given by

$$\mathcal{CO}_i = \{\boldsymbol{q} \in \mathcal{C} : \mathcal{R}(\boldsymbol{q}) \cap \mathcal{O}_i \neq \emptyset\} ,$$

where we recall that $\mathcal{R}$ is the robot (composed by a single rigid body or a set of rigid bodies $\mathcal{R} = \mathcal{R}_1, \ldots, \mathcal{R}_n$ connected each other in a kinematic chain) and $\mathcal{R}(\boldsymbol{q})$ is the kinematic map (direct kinematics) that projects a configuration of the robot in the workspace. Note that, in the previous definition, we implicitly assumed that each obstacle is closed (i.e., it includes its boundaries). The extension to the multiple obstacles case is pretty straightforward

$$\mathcal{CO} = \cup_{i=1}^{l} \mathcal{CO}_i \ ,$$

that defines completely the *C-obstacle region*, with l the number of obstacles. The part of the space that is not included in $\mathcal{CO}$ is called *free space* and it is formally defined as

$$\mathcal{C}_{\text{free}} = \mathcal{C} - \mathcal{CO} = \left\{ \boldsymbol{q} \in \mathcal{C} : \mathcal{R}(\boldsymbol{q}) \cap \left(\cup_{i=1}^{l} \mathcal{O}_i \right) = \emptyset \right\} \ ,$$

that is the portion of the configuration space where there are no self-collisions nor collisions with obstacles. Even in the case in which $\mathcal{C}$ is a connected space, there is the chance that $\mathcal{C}_{\text{free}}$ is not connected, due to the occlusions provided by the C-obstacle region. In the canonical problem introduced before, the assumption of a free-flying system means that it is able to follow any path in $\mathcal{C}_{\text{free}}$.

Thanks to the introduction of the new concepts, it is possible to introduce a more formal definition of the canonical motion planning problem. Given a starting $\boldsymbol{q}_{\text{start}}$ and a goal $\boldsymbol{q}_{\text{goal}}$ configuration (they can be also provided by inverse kinematics of postures of the robot $\mathcal{R}$ in $\mathcal{W}$), the aim of the motion planning is to define a collision-free path that lies entirely in $\mathcal{C}_{\text{free}}$. Otherwise, the motion planner should report a failure.

Even if the previous definition of the C-obstacle region seems straightforward, it is really hard to have an explicit representation of this space in real applications. In fact, there is no general algorithm that is able to construct $\mathcal{CO}$. In the case in which the obstacles are polygons, the star algorithm can be used. Actually, this algorithm works in 2D and it assumes that the obstacles are convex. It can be extended to the non-convex case since a non-convex object can be approximated as a composition of multiple convex shapes. More details about this approach can be found in [79]. We mentioned this algorithm to assert that, even if there are works whose aim is to build an explicit representation of $\mathcal{CO}$, it is really hard in general to create an explicit representation for $\mathcal{CO}$, due also to the computational effort that this representation requires. In addition, even assuming that a representation exists, it would be not valid anymore in case the robot or an obstacle moves, leading to the necessity of building from scratch a new representation. For this reason, a complementary approach is used in this thesis (as in the majority of approaches used today in the literature): use a collision-checker. It will be described in the next chapter.

Chapter 3

Sampling-based motion planning

The aim of this chapter is to introduce some concepts about the sampling-based methods for motion planning. The structure and the main concepts are taken from [76, 79, 104]. Then, the reader is referred to these books for further details. As for the previous chapter, the reason for which we mention these concepts is to create a self-contained thesis.

The main idea under the sampling-based motion planning algorithms relies in avoiding the explicit construction of $\mathcal{CO}$. Instead, they sample configurations in the configuration space $\mathcal{C}$, checking if they are in collision or not through repetitive calls to a *collision checker*. The idea is that, by iteratively sampling $\mathcal{C}$, a finite set of collision-free configurations is found. In this set, there can be found several paths that solve a given instance of the motion planning problem. This approach is in contrast with the one described at the end of the previous chapter, where an explicit representation of $\mathcal{CO}$ was built. The sampling approach was proved to be more effective in real applications not limited to robotics but also to manufacturing and biological applications. In these cases, it is impossible to build an explicit representation of $\mathcal{CO}$, due to its complexity and for the huge amount of time needed for that (even in the simplest cases). This is the reason for which, in the last decades, sampling-based algorithms have been widely used.

This comes with a price that is the *completeness* of the algorithm. An algorithm is said to be *complete* if, for any input, it tells if a solution exists in a finite amount of time. In other words, it means that a solution should be returned in a finite amount of time, assuming that a solution exists. In the sampling-based approaches, it cannot be ensured. For these algorithms, we talk about *resolution completeness* for deterministic approaches while we said that an algorithm is *probabilistically complete* if the algorithm is probabilistic. For the former, we mean that the algorithm will find a solution in a finite amount of time, if it exists; in case a solution does not exist, it may loop forever. On the other side, a probabilistically complete algorithm will find a solution with a probability that goes to one as the time approaches infinity. In this thesis, we will propose probabilistically complete algorithms. Fundamental for these algorithm is the concept of *distance* in the configuration space that measures the distance between two configurations. In case the C-space is $\mathbb{R}^n$, a trivial choice is to define the distance as the Euclidean distance.

Anyway, other choices are possible, even for $\mathbb{R}^n$. A metric space (X, d) is a topological space X embedding a metric function $d : X \times X \to \mathbb{R}$ such that

- $d(\boldsymbol{x}_1, \boldsymbol{x}_2) > 0, \forall \boldsymbol{x}_1, \boldsymbol{x}_2 \in X$;
- $d(\boldsymbol{x}_1, \boldsymbol{x}_2) = 0 \iff \boldsymbol{x}_1 = \boldsymbol{x}_2$;
- $d(\boldsymbol{x}_1, \boldsymbol{x}_2) = d(\boldsymbol{x}_2, \boldsymbol{x}_1), \forall \boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathcal{X}$;
- $d(\boldsymbol{x}_1, \boldsymbol{x}_2) + d(\boldsymbol{x}_2, \boldsymbol{x}_3) \geq d(\boldsymbol{x}_1, \boldsymbol{x}_3), \forall \boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3 \in X$.

Due to the generality of this definition, multiple choices are possible for the metric function. Just as reference, one important and widely used metrics is the L_p

$$d(\boldsymbol{x}_1, \boldsymbol{x}_2) = \left(\sum_{i=1}^{n} \|\boldsymbol{x}_{1,i} - \boldsymbol{x}_{2,i}\|^p \right)^{1/p} ,$$

where $\boldsymbol{x}_{j,i}$ is the ith component of $\boldsymbol{x}_j$. Changing the value of p, some of the most famous metrics can be obtained. In fact, if p is set to 2, the Euclidean distance is obtained; if p is 1, the Manhattan distance is the result; finally, if p is ∞, the L_∞ is the metrics that one wants to use. A nice property of the metric functions is that they nicely generalized in case the C-space is composed by several subspaces. In fact, it can be composed by the Cartesian product. As example, assume the C-space is composed by the union of two metric spaces (X, d_x) and (Y, d_y). The general metric function (Z, d_z) can be chosen as

$$d_z(\boldsymbol{z}_1, \boldsymbol{z}_2) = \alpha \; d_x(\boldsymbol{x}_1, \boldsymbol{x}_2) + \beta \; d_y(\boldsymbol{y}_1, \boldsymbol{y}_2) \; ,$$

where $\boldsymbol{z}_i = (\boldsymbol{x}_i, \boldsymbol{y}_i)$ and $\alpha, \beta > 0$ are weighted scalars, and $\boldsymbol{x}_i \in X$ and $\boldsymbol{y}_i \in Y, i = 1, 2$.

As before, consider the case of $SE(3)$ since it will be the space most used in this thesis. The main challenge here is to define a proper metric function for the orientation part, i.e. to define a proper distance function for $SO(3)$. In case the orientation is represented by quaternions, note that there is the identification problem, as discussed in the previous chapter. Recall in fact that the unit quaternion lies in a subset $\mathbb{S}^3$ of $\mathbb{R}^4$. Anyway, a metric defines for $\mathbb{S}^3$ will not be valid for the quaternion case, due to the identification problem of the antipodal points. A possible choice for this metric is

$$d(\boldsymbol{h}_1, \boldsymbol{h}_2) = \min \left\{ d_s(\boldsymbol{h}_1, \boldsymbol{h}_2), d_s(\boldsymbol{h}_1, -\boldsymbol{h}_2) \right\} , \tag{3.1}$$

where $d_s(\boldsymbol{h}_1, \boldsymbol{h}_2) = \cos^{-1}(a_1 a_2 + b_1 b_2 + c_1 c_2 + d_1 d_2)$, $\boldsymbol{h}_i = (a_i, b_i, c_i, d_i)$ is the ith quaternion and the reason for which there is the min term is due to the antipodal point constraint.

Having defined a distance function for $SO(3)$, we can turn our attention to its definition in $SE(3)$. A trivial extension would be to use eq. (3.1) for the orientation component and another metric for $\mathbb{R}^3$, introducing weights. Even if that is correct, it means that one has to tune these weights. Other choices are possible, as the ones we will propose in the next chapter of this thesis.

3.1 Single-query versus multi-query algorithms

First, the sampling-based algorithms can be grouped into two main categories

1. multi-query algorithms;
2. single-query algorithms.

In a single-query algorithm, the motion planning problem is solved from scratch every time the algorithm is invoked. In other words, given a starting $\boldsymbol{q}_{\text{start}}$ and a goal $\boldsymbol{q}_{\text{goal}}$ configurations, the motion planner searches for a solution, without having any information from previous runs of the algorithm. For this reason, these algorithms are suitable for situations in which one wants to solve the problem for a particular choice of $\boldsymbol{q}_{\text{start}}$ and $\boldsymbol{q}_{\text{goal}}$. Then, it is clear that any form of pre-computation is not useful. The main advantage in using this kind of algorithm relies in the speed of convergence, usually faster than multi-query algorithms. On the other side, the main disadvantage is the fact that it has to start from scratch when finding a new solution; in case it has to be done for the same environment, it might be not efficient. The Rapidly-exploring Random Trees (RRTs) [78] are one example of such algorithms. They will be used in the next chapter, for solving the motion planning problem for humanoids.

On the contrary, a multi-query algorithm is used when several calls to the same motion planning problem has to be solved. Since one knows a priori that the motion planner has to be invoked more than one time on the same robot model and the same environment, it makes sense to invest some time in creating a sort of "preprocessing" information. In this way, future queries will be answered efficiently. The main idea is then to solve the problem in two phases: in the first phase, called learning phase, a *roadmap* is built; in a second phase, also referred to as query phase, a given a couple of nodes $\boldsymbol{q}_{\text{start}}$, $\boldsymbol{q}_{\text{goal}}$ is connected to the roadmap. A path is then found, since the built roadmap is connected and a path can be found that connects $\boldsymbol{q}_{\text{start}}$ and $\boldsymbol{q}_{\text{goal}}$. This approach is known in literature as *probabilistic roadmap*. More in details, the learning phase consists in randomly sampling the $\mathcal{C}$-space and checking if these configurations lie in $\mathcal{C}_{\text{free}}$. At each iteration, a random sample $\boldsymbol{q}_{\text{rand}}$ is extracted and checked for collisions. In case it lies in $\mathcal{C}_{\text{free}}$, it is attempted to be connected in the roadmap to its neighbours[1] through a local planner. As example, a really simple local planner consists in connecting two nodes by a straight line and checking if this line lies in $\mathcal{C}_{\text{free}}$ (this planner is valid only in case no kinematic constraints act on the robot). Once the roadmap is built, a pair $\boldsymbol{q}_{\text{start}}$, $\boldsymbol{q}_{\text{goal}}$ is given. The query phase then consists in connecting those configurations to nodes of the roadmap. To this aim, the algorithm searches for neighbours, that are listed in increasing order of distance w.r.t. $\boldsymbol{q}_{\text{start}}$ (resp. $\boldsymbol{q}_{\text{goal}}$). The local planner is again invoked to connect one configuration in this list and $\boldsymbol{q}_{\text{start}}$ (resp. $\boldsymbol{q}_{\text{goal}}$). If a valid path is found, both for the connection of $\boldsymbol{q}_{\text{start}}$ and $\boldsymbol{q}_{\text{goal}}$ to the roadmap, then a solution for the motion planning problem is found. The main advantage using a multi-query algorithm relies

[1] A neighbour can be identified as a node in the roadmap whose distance is less than a predefined threshold.

Figure 3.1 An example of task-constrained motion planning problem. The robot has to open a cabinet. Courtesy of [8].

in the roadmap, that enables to find quick solution for future queries. The main disadvantage is in the learning phase, that needs time to build the roadmap. The density of the roadmap is another key point. If it is too dense, the path search in the query phase may be not so trivial and might need time. On the other hand, the more sparse it is, the less classes of homotopy paths are given as output. Both RRTs and probabilistic roadmaps are probabilistically complete.

In this thesis, we will focus on single-query algorithms since we want to extend the proposed approaches towards the on-line case, where the environment is highly dynamic.

3.2 Task-constrained motion planning

Here, the motion planning problem defined in the previous sections is particularized in the case in which there is an explicit task the robot has to fulfil. This problem is referred to as Task-Constrained Motion Planning problem (TCMP in the rest of this thesis). As example, consider Figure 3.1, where a humanoid robot has to open a cabinet. In this case, a trajectory for the right hand of the robot should be defined and tracked.

Formally, a TCMP problem can be defined as generating the motion for a robot that should satisfy some task space constraints (e.g., moving the right hand along a predefined trajectory in Figure 3.1) while avoiding collisions with parts of itself (self-collision) or with workspace obstacles. In this problem, the robot is usually redundant w.r.t. the task that it has to accomplish. Just for sake of completeness, a robot is said to be *redundant* w.r.t. a task if its number of DOFs n is strictly greater than the dimension of the task m. Obviously, if the robot is non-redundant ($n = m$), the motion planning is not needed, since there is just one way to fulfil the task. In this case, a standard Jacobian-based inverse kinematic scheme solves the problem. However, there is no way to modify the joint motion in case of collisions, since there is just one joint motion that fulfil the task. For this reason, we will assume, here and in the following, that the robot is redundant w.r.t. the task it has to fulfil.

Up to the author's knowledge, the first explicit reference to a TCMP problem is in [92] where the MPEP (Motion Planning along End-effector Paths) problem was introduced for redundant robots. Note that the anachronisms MPEP and TCMP refer to the same planning problem. Here and in the following, we will use TCMP, to be consistent with the rest of the thesis. At the very

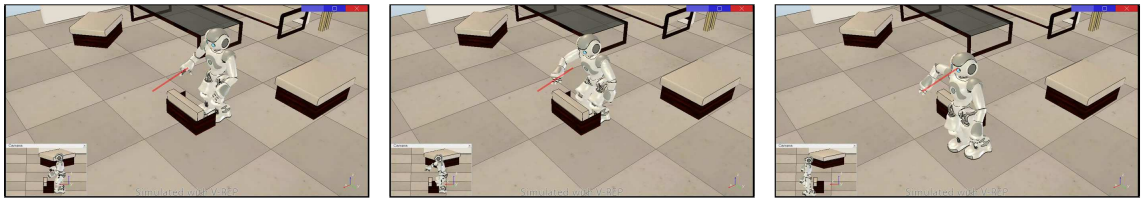


Figure 3.2 An example where a pure kinematic control is not able to solve the TCMP problem while a motion planning approach does. (left) The planning problem with the desired trajectory for the right hand (red). (center) Final solution of the kinematic control. (right) Final solution of a motion planning algorithm.

beginning, the TCMP problem was tackled and labelled as special case of redundancy resolution. An example is in [101] where kinematic schemes are used to solve the problem. Even if this might sound trivial, it is important to underline that a pure kinematic scheme is not suggested for facing the TCMP problem. In fact, it is inherently local and only suited for generating reactive behaviours. Moreover, it has no backtracking capabilities and it may fail in finding a solution for complex problems. To convince the reader, consider the scenario in Figure 3.2. In this figure, the right hand of the humanoid robot is constrained along a predefined trajectory, depicted as a red line. Using a kinematic control, the robot enters into the black box without having any opportunity to move out from it, since the task will constraint the robot to move forward (in addition, the robot cannot overcome the lateral obstacle with a step due its locomotive limitations). The result is that the robot is not able to complete the task, even stretching its arm. On the other side, this problem is solved by a motion planning algorithm, as in the right of Figure 3.2. The result is that the robot completes the task by taking some lateral steps. Using really rough words, the motion planner explores multiple paths, finding one that fulfils the task, embedding also backtracking capabilities. This is the real reason for which we will focus on motion planning algorithms in this thesis. Another approach for dealing with the TCMP problem is through the optimal control. Using this approach, the TCMP is formalized as a non-linear Two-Point Boundary Value Problem (TPBVP). Just for completeness, a BVP can be defined as a system of ordinary differential equations with solution and derivative values specified at more than one point. In case the solution and derivatives are specified at just two points, then it is defined as a TPBVP. A solution can be found via numerical techniques. The main drawiback of this approach is that it has no guarantee of success.

In its first formulation in [90, 92], the TCMP was formulated as finding a joint path $\boldsymbol{q}(\sigma)$ that satisfies

$$\boldsymbol{y}(\sigma) = \boldsymbol{f}(\boldsymbol{q}(\sigma)) \; \forall \sigma \in [0, 1] \; ,$$

where $\boldsymbol{q}$ is the configuration of the robot, $\boldsymbol{y}$ is the desired task in the dexterous task space, σ is the parametrization of the path, $\boldsymbol{f}$ is the forward kinematic map. In addition to the fulfilment of the task, the robot is requested to not collide with itself (self-collision) nor with an obstacle that are in the environment. Furthermore, joint limits and velocity joint limits should not be violated. In this formulation, an initial configuration $\boldsymbol{q}(0)$ can be assigned or not. The former is the case in

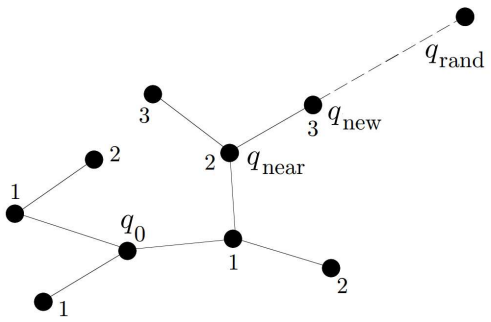


Figure 3.3 Basic step of the RRT algorithm: (i) extract a random configuration $\boldsymbol{q}_{\text{rand}}$; (ii) find the nearest configuration $\boldsymbol{q}_{\text{near}}$; (iii) find $\boldsymbol{q}_{\text{new}}$ in the direction of $\boldsymbol{q}_{\text{rand}}$. The numbers on each node refers to the depth of the node itself.

which the planning is invoked from sensor information, that defines the robot current state. The latter is the case the planner is invoked off-line, letting to the planner itself the role to identify a proper initial configuration. The problem was faced for fixed-based redundant manipulators (e.g., snake robots) in [92] and for mobile manipulators in [90]. The main idea in [92] (that will be revised for humanoid robots in the next chapter) is to seek for a solution as the linear interpolation of collision-free joint configurations $\boldsymbol{q}(\sigma_0), \dots, \boldsymbol{q}(\sigma_s)$, with the σ_i's equally spaced and s the number of samples one wants to sample the task. Finally $\boldsymbol{y}(\sigma_i) = \boldsymbol{f}(\boldsymbol{q}(\sigma_i)), \forall i \in [0, s]$. Even if the task might be not satisfied along the path interpolating two consecutive configurations, the authors justified this by saying that the resolution can be increased by increasing the parameter s. The authors then proposed two different methods for solving the TCMP problem: greedy and RRT-like. In the first method, the algorithm favours the exploitation versus the exploration of the configuration space. In few words, it divides the configuration space into two components: the redundant and non-redundant one (the robot is supposed to be redundant w.r.t. the task it has to accomplish). At each iteration, the algorithm starts from a configuration $\boldsymbol{q}(\boldsymbol{\sigma}_i)$. Then, it samples a random configuration $\boldsymbol{q}(\boldsymbol{\sigma}_{i+1})$ for its redundant part and uses an inverse kinematic scheme to find a collision-free configuration that copes with the next task sample ($\boldsymbol{y}(\boldsymbol{\sigma}_{i+1})$). The path between the $\boldsymbol{q}(\boldsymbol{\sigma}_i)$ and $\boldsymbol{q}(\boldsymbol{\sigma}_{i+1})$ is checked for collision. If it is collision-free, then it is added to the configuration list, until the task is completely fulfilled. As it is evident from this very rough description, the algorithm works, at each iteration, on the last valid sample found. This means that a valid configuration found two or more iterations before is never expanded again. This might cause some problems, since the redundancy of the robot is not explored in details. In fact, this kind of solution works well for easy scenes, where there is (almost) no need to explore the configuration space. This is the reason for which the authors introduced the RRT-like scheme.

The RRT-like algorithm builds a tree $\mathcal{T}$ in the configuration space of the robot in an iterative way, rooted at $\boldsymbol{q}(\boldsymbol{\sigma}_0)$. In this framework, $\boldsymbol{q}(\boldsymbol{\sigma}_0)$ is computed by first choosing a random value for the redundant variables and then using an inverse kinematic scheme for the rest of the configuration vector. Then, at each iteration, a random configuration $\boldsymbol{q}_{\text{rand}}$ is extracted and the nearest one $\boldsymbol{q}_{\text{near}}$ is retrieved among the nodes in $\mathcal{T}$, as depicted in Figure 3.3. A difference

between a classical RRT and the RRT-like in [92] relies in this step. Given two configurations, the former computes the minimum distance (appropriately defined for the C-space) for the whole configuration vector while the latter computes the distance just for the redundant component, returning also the corresponding task sample k (the task is supposed to be sampled in s points, as for the greedy algorithm). The new configuration is set with an incremental distance $\Delta \boldsymbol{q}$ from $\boldsymbol{q}_{\text{near}}$, in the direction of $\boldsymbol{q}_{\text{rand}}$ (as in Figure 3.3). The RRT-like algorithm slightly modify it by doing that just for the redundant part of $\boldsymbol{q}_{\text{near}}$ and by invoking an inverse kinematic scheme to complete the definition of $\boldsymbol{q}_{\text{new}}$. A local planner is finally invoked between $\boldsymbol{q}_{\text{near}}$ and $\boldsymbol{q}_{\text{new}}$, in order to check its path. In case there are no collisions, $\boldsymbol{q}_{\text{new}}$ is added as node of $\mathcal{T}$ while the path joining $\boldsymbol{q}_{\text{near}}$ and $\boldsymbol{q}_{\text{new}}$ is added as an edge of $\mathcal{T}$. Otherwise, the process is repeated. The reader might be confused on which is the point where the algorithm ensures the fulfilment of the task. To this aim, note that, when $\boldsymbol{q}_{\text{new}}$ is computed, the inverse kinematic scheme ensures that $\boldsymbol{y}(\sigma_{k+1}) = \boldsymbol{f}(\boldsymbol{q}_{\text{new}})$ lie. In this way, the task is again fulfilled on the sampled points. The RRT-like (as the RRT) algorithm is probabilistically complete and the experiments in [92] show that this method is particularly suitable for difficult scenes, where the greedy algorithm may fail in finding a solution. On easy scenes, the greedy algorithm outperforms the RRT-like one in terms of planning time, even if the latter is able to find a solution.

The same authors generalized the above-mentioned framework in [90] for mobile manipulators. Here, the main difference is the definition of the configuration space, that is naturally partitioned in the platform configuration (for the mobile base) and in the manipulator configuration. The former plays the role of the redundant variables in [92], while the latter plays the role of the non-redundant variables in [92]. Another contribution of [90] relies in the extension of the TCMP framework for non-holonomic constraints (since the robot has a mobile base).

The TCMP framework was further extended in [91] for mobile manipulators. The main difference with the other approaches is in the introduction of a *motion generation scheme* that ensures the continued satisfaction of task constraint. Recall that the fulfilment of the task in [90, 92] is ensured only in the task samples but not during the motion between samples. The work in [91] is based on the concept of *control based* motion planning, formally defined in [16]. The main idea is that configuration samples are generated using a differential model of the robot (called *motion generation scheme*). A tree $\mathcal{T}$ is built in the configuration space in order to find a collision-free path. As for [90], the task is sampled in s points and an initial configuration $\boldsymbol{q}(\boldsymbol{\sigma}_0)$ may be assigned or not. In the latter case, a random configuration is chosen for the redundant part, while an inverse kinematic algorithm completes the definition of $\boldsymbol{q}(\boldsymbol{\sigma}_0)$. At each iteration, a random configuration $\boldsymbol{q}_{\text{rand}}$ is extracted and the nearest configuration $\boldsymbol{q}_{\text{near}}$ is retrieved as the one in $\mathcal{T}$ having the smallest distance to $\boldsymbol{q}_{\text{rand}}$. At this point, the *motion generator* is invoked, depicted in Figure 3.4. Note that, since $\boldsymbol{q}_{\text{near}}$ has been already computed, also the corresponding task sample k has been identified. The motion generation performs a *forward motion*, a *self-motion* and a *backward motion*. The forward motion produces a subpath leading to a configuration on the manifold associated with the next task sample $\mathcal{L}_{k+1}$. This path is checked for collisions and it

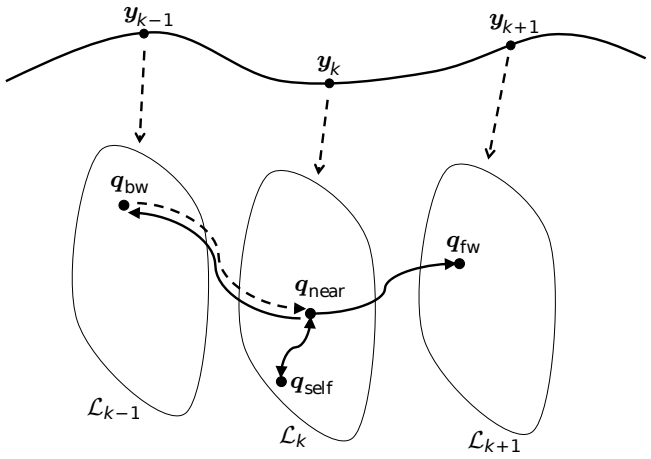


Figure 3.4 The extend procedure of [91]. Three configurations and subpaths are generated at each step: forward motion ($\boldsymbol{q}_{\text{fw}}$); self-motion ($\boldsymbol{q}_{\text{self}}$) and backward motion ($\boldsymbol{q}_{\text{bw}}$). The figure is a courtesy of [91].

is, in case, added to $\mathcal{T}$. The same happens for the self-motion that produces $\boldsymbol{q}_{\text{self}}$, a configuration that lies in $\mathcal{L}_k$. Within this motion, the end-effector is forced to remain fixed in the same pose, action that is possible thanks to the redundancy of the robot w.r.t. the task. Finally, the backward motion generates $\boldsymbol{q}_{\text{bw}}$ (lying in $\mathcal{L}_{k-1}$) that is eventually added to $\mathcal{T}$ in case of collision-free path with $\boldsymbol{q}_{\text{near}}$. The main difference w.r.t. [90, 92] relies in the generation of these subpaths. In [91] the task is exactly satisfied along these subpaths in view of a Jacobian-based inverse kinematic scheme, that will be described in details in the next chapter.

The reason for which we described in details the approach in [91] is because this thesis is highly inspired by this work and some concepts will be used also in this thesis.

In the literature, other works faced the TCMP during the years. As example, three different algorithms are proposed in [110] for solving the TCMP in case of fixed-based manipulators. This approach is extended in [111] where the kind of tasks are generalized (e.g., opening a door). On the other hand, sampled-based motion planners are proposed in [66, 105], taking into account uncertainties. This is a really promising field of research since, even if a motion planner plans a trajectory for each joint, the robot will be not able to exactly apply them. This is due to the robot actuators, that are not able to apply instantaneously a given command and to the inevitable differences between the robot and the environment models and the corresponding real counterparts. However, this problem is not faced in this thesis.

The authors in [11] proposed a method for solving the TCMP problem in case of repetitive tasks. In other words, the motion of the robot should be planned in such a way it returns in the same configuration at the end of the task. To this aim, a bidirectional RRT is built (two trees are grown respectively from the start and the goal configurations, that in this problem are equal each other) and a loop closure is proposed in order to join the two trees. A similar approach is proposed in [12], where the TCMP is proposed in case the environment is composed by moving obstacles, i.e. in the case where the trajectories of the obstacles are known in advance. The same

authors extended the framework in [12] including also the bounds on the actuator torques on the robot motors. This leads to the adoption of a acceleration-level motion generation and this is the main contribution of [10].

Most of the approached described in this section are applied for the fixed-base manipulators. In this thesis, we propose an algorithm that is able to solve the TCMP problem for humanoid robots. It is important to underline that the extension of an algorithm aimed for other robotic platforms to humanoid robots is not trivial. This is true for a number of reasons, that will be described in details in the next chapter.

I

MOTION PLANNING FOR HUMANOIDS

Chapter 4

Task-oriented whole-body motion planning for humanoids based on step generation

This chapter describes and proposes an approach for solving the problem of planning the motion of a humanoid robot that must execute manipulation tasks, possibly requiring stepping, in environments cluttered by obstacles. In other words, a method for solving the TCMP problem (introduced in the previous chapter) for humanoid robots is proposed. The reason for which we chose to face this problem was inspired from the intensive research activity on which humanoid robots were subjected to in the last two decades. The long term objective is to create highly versatile robotic platforms that can effectively assist – or even replace – humans in their (repetitive and/or dangerous) daily activities.

In the past, many researchers have focused on the design of control models and algorithms for achieving stable biped locomotion [51, 52, 53, 54, 115]. However, thanks to the high number a humanoid robot is typically equipped with, humanoids are capable to achieve other complex tasks in addition to locomotion, e.g., manipulation. To fully express their versatility, they must be able to plan and perform whole-body motions in unstructured environments that are populated by obstacles.

As already mentioned in the previous chapter, one possible approach for dealing with multiple tasks is the kinematic control [103], that can be extended in order to include collision-free requirements [57]. A clever application of this technique for humanoid footstep generation in manipulation tasks is presented in [58]. However, the kinematic control is a local technique prone to local minima and without backtracking capabilities. Here and in the rest of the thesis we will assume that the geometry of the scene is known in advance, as well as a model of the robot. Within these hypotheses, the robot should execute a task, as declared above.

The motion planning problem for humanoid robots is challenging for a number of reasons. The first is the number of degrees of freedom. A humanoid robot is typically equipped with more joints w.r.t. other robot platforms. This brings to the definition of high dimensional configuration spaces, as formally defined in the next section. Obviously, the higher is the dimension of the

configuration space, the harder is to find a solution to the motion planning problem since the space where to find a solution is wider and the cardinality of the infinite ways of fulfilling a task becomes bigger and bigger. The second is that a humanoid robot is not a free-flying system in its configuration space. The motion must be generated appropriately. Finally, the requirement that the robot maintains equilibrium, either static or dynamic, typically constrains the trajectory of the robot Center of Mass (CoM), thus reducing the dimension of the planning space. The last point is a specific problem of legged robots: in fact, the CoM is not explicitly constrained when planning the motion of, e.g., a mobile robot since this constraint is usually satisfied by construction[1]. On the contrary, maintaining the equilibrium is a fundamental problem for a humanoid robot, whose fulfilment is not as trivial as for a mobile platform. These points should convince the reader that planning the motion for humanoid robots is an hard problem. For this reason, even in the simple configuration-to-configuration case (i.e., find a collision-free motion between two configurations $\boldsymbol{q}_{\text{start}}$ and $\boldsymbol{q}_{\text{goal}}$), the motion planning problem for humanoids is usually approached by introducing simplifications at various levels of the problem formulation.

For example, one may simplify the environment by taking into account collisions, at least in a first phase, only at the footstep [15, 69, 70] or at the leg [95] level. A somewhat dual approach consists in finding first a collision-free path for a simplified geometric model of the humanoid, such as its bounding volume, and then approximating this path with a feasible locomotion trajectory. This technique is used in [96] to animate digital characters and in [126] to plan dynamically feasible motions of a humanoid. However, these methods may require to reshape the path if the feasible trajectory is found to be in collision with obstacles. The whole configuration space of a humanoid is considered in [68] to plan first collision-free, statically stable motions that are then converted to dynamically stable collision-free trajectories; in this work, however, the robot does not perform stepping motions. Due to the coarse approximation of the robot occupancy, this method may fail to find a solution in cluttered environments; for example, extending over a table to pick up an object is not possible. On the contrary, such a task can be performed using the techniques developed in this thesis, as we will show starting from the next section. The method in [39] represents one of the few motion planning methods that simultaneously generates footsteps and whole-body motions.

Most of the above works do not directly incorporate task constraints. It is clear that the problem becomes even harder to solve in the presence of a task that the robot has to accomplish (e.g., move an object from a start to an end location in a room). This is what we refer to as TCMP problem, i.e., finding collision-free motions for a humanoid that is assigned a certain task (e.g., a manipulation action) whose execution may require stepping. Some works in the literature faces this problem and they can be grouped in three main categories

1. separate locomotion from task execution [8, 42];

[1]The CoM typically lies in the support polygon in a mobile robot.

2. integrate them with a two-phase planner which first computes a collision-free, statically stable paths for a free-flying humanoid base, and then approximates it with a dynamically stable walking motion [22, 23, 56, 68, 126];

3. achieve acyclic locomotion and task execution through whole-body contact planning [5, 6, 32, 80].

The authors in [8] propose a framework for planning the motion of a humanoid robot whose goal is to manipulate objects. More in details, it builds a RRT-Connect (a bidirectional tree, originally introduced in [71]) in the configuration space of the robot (each node in the tree contains a whole-body configuration of the robot and an information about the object handle trajectory, described in the following). First, a catalogue of statically stable configurations (free of self-collisions and respecting the joint limits) are precomputed and stored in a database. Then, two trees are grown from the starting (the current state of the robot) and the goal (chosen through a inverse kinematic solver, in function of the object to be manipulated) configurations. At each step, one tree is expanded by sampling a random stable configuration from the database, computing the nearest configuration in the tree and generating a new configuration starting from the nearest one in the direction of the random sample, following the standard steps of a RRT. If the new configuration is not valid, the two trees are swapped and the procedure is repeated. On the contrary, the new configuration is added to the current tree while the other tree is expanded towards the newly added configuration. In positive case, a valid path is found and the problem is solved. Otherwise, the algorithm proceeds with a new iteration. In this approach, the hand is enforced to track a given trajectory in order to manipulate an object. This trajectory is specified at the beginning, in function of the object to be handled. When the new configuration is generated, an inverse kinematic solver finds an arm configuration that minimizes the distance to the nearest configuration found by the algorithm. This ensures that the new configuration will lie on the specified trajectory. The main limitation of this approach is that it is not trivial to extend in case the task requires stepping. Then, locomotion is not taken into account in [8]. On the contrary, stepping motions appear naturally within our framework.

The works in the second group, as mentioned, propose a two-stage approach. The work in [23] first computes a collision-free, statically stable paths for a free-flying humanoid base, and then approximates it with a dynamically stable walking motion. More in details, a RRT is built in the configuration space (defined by the robot joints and the pose of a representative point of the robot in $SE(3)$). The aim of this motion planner is to find a collision-free path for a sliding-feet robot, taking into account two constraints: (i) static balance (CoM in the support polygon); (ii) end-effector pose. Since in reality a legged robot cannot slide on a regular floor, such paths are physically unfeasible. For this reason, the statically balanced sliding path is converted to a dynamically balanced walk. Footsteps are placed corresponding to the nominal walk pattern of the robot. Given the footsteps, the Zero Moment Point (ZMP) trajectory is computed, together with the desired trajectories for the feet. From that, a CoM trajectory is generated using the

preview controller introduced in [51]. A standard Jacobian-based inverse kinematic scheme is used for computing joint velocities. In particular, a prioritized task framework is used for the following tasks, in decreasing order of priority: (i) positions and orientations of the feet; (ii) horizontal position of the CoM; (iii) height of the CoM; (iv) verticality of the waist; (v) task for a specific point of the robot. The authors prove, in the same paper, that a statically gait can be converted to a dynamically one through the well-known concept in control theory of small-space controllability.

There are two major differences between the approach in [23] and ours. First, we do not split the approach into two stages but we generate the joint motion in a single one. We believe that generating the motion in a single stage allows to fully take advantage of the robot DOFs. Second, we do not adopt a prioritized scheme but we make usage of the augmented Jacobian. This because we want to ensure that the task (assigned for a specific point of the robot) is fulfilled. With the above-mentioned tasks list, the assigned task has the lowest priority and it will be fulfilled only if the ones having higher priorities are satisfied. Consequently, there is no guarantee that it will be accomplished. Second, thanks to the fact that we simultaneously generate footsteps and whole-body motions, we can obtain plans where the robot steps over obstacles, a feature that is not possible to obtain in [23]. Moreover, both static and dynamic walking gait are contemplated within our framework, while collisions found in the first step of [23] can be avoided just with a reshaping to a dynamic walking gait.

The third category mainly includes the works from Kheddar. The main idea under these works is to plan the motion of a humanoid robot whose task is to maintain multi-contact stances with the environment. The approach in [6] uses a depth-first motion planning algorithm in order to create a path between two consecutive stances. An inverse kinematics-and-statics solver is used for finding configurations and contact stances simultaneously. Finally, an optimization problem is solved to find the path in such a way the following constraints are satisfied: (i) static equilibrium; (ii) contact forces within friction cones; (iii) joints within joint limits; (iv) torques within torque bounds; the path is collision-free. The major advantage in using our approach w.r.t. [6] is that the latter does not allow to consider task paths nor it easily generalizes for dynamic walking.

As mentioned before, our approach does not separate locomotion from task execution. In particular, using the TCMP framework developed in [91] and the concepts introduced in Section 3.2, the proposed method explores the submanifold[2] of the configuration space that is admissible with respect to tasks and possible other constraints, including humanoid equilibrium.

As we will see in the next section, our approach consists in growing a tree in the configuration space of the humanoid robot, whose expansion in the constrained manifold is obtained via a hybrid motion generation scheme that is able to generate, at the same time, feet positions and and whole-body motions, that are validated by a collision checker.

To summarize, the method we will propose in this thesis has two main contributions

[2]It is a submanifold since there are some constraints, such as equilibrium, that limits the manifold dimensionality.

1. motions are generated in a single phase;
2. walking emerges *naturally* from the solution of the planning problem as driven by the assigned task.

Just for reference, this chapter is organized as follows. In Section 4.1, we formally introduce the motion model for the humanoid robot, together with the definition of the configuration space and a rigorous formulation of the addressed problem. The hybrid motion generation mechanism, that is the core of our method, is presented in Section 4.2. The randomized planner is discussed in 4.3. Finally, planning experiments performed in V-REP for the NAO humanoid robot are reported in Section 4.4.

4.1 Problem formulation

In this section, we will formulate our planning problem. To this aim, it is convenient to first introduce a convenient motion model for a humanoid robot.

4.1.1 Humanoid motion model

As discussed in Section 3.2, it is fundamental for a motion planning algorithm to define a configuration space. Denote with n the number of joints of a humanoid robot. If we assume that the humanoid is a free-flying system, one might specify its configuration by assigning the pose (position plus orientation) of a specific body of the robot (e.g., the torso) and the values of the n joints. Since we are interested in planning motions where at least one foot touches the ground (i.e., we do not take into account jumping movements), we can choose the support foot as the above-mentioned specific body and we can define a configuration simply as

$$\boldsymbol{q} = \begin{pmatrix} \boldsymbol{q}_{\mathrm{spt}} \\ \boldsymbol{q}_{\mathrm{jnt}} \end{pmatrix}, \tag{4.1}$$

where $\boldsymbol{q}_{\mathrm{spt}} = (x_{\mathrm{spt}} \quad y_{\mathrm{spt}} \quad \theta_{\mathrm{spt}})^T \in SE(2)$ is the planar pose of the support foot in a world reference frame and $\boldsymbol{q}_{\mathrm{jnt}} \in \mathcal{C}_{\mathrm{jnt}}$ is the n-vector of joint angles, with the associated joint space $\mathcal{C}_{\mathrm{jnt}}$. The humanoid configuration space $SE(2) \times \mathcal{C}_{\mathrm{jnt}}$ has therefore dimension $n + 3$.

The above definition in eq. (4.1) of a configuration requires that the support foot is always identified, even when the humanoid is in a double support phase (both feet on the ground). In such a situation, our planner may arbitrary define the support foot as one of the two feet, since both of them touch the ground. The same holds also for the starting configuration (supposed to be assigned and corresponding to the current state of the robot), which is typically in double support. Note that the identity of the support foot vary over time. In fact, whenever a step is planned, the swing foot moves from its current to a different location on the ground, which is defined as the new pose of the support foot, as shown in Figure 4.1.

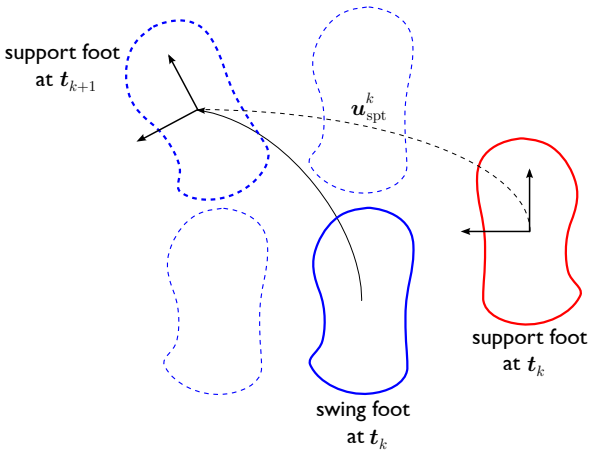


Figure 4.1 Support foot displacement after a step initiated at t_k. The swing foot moves from its initial pose to the selected final one (thick dashed blue), and becomes the new support foot. The support foot displacement $\boldsymbol{u}^k_{\text{spt}}$ is defined accordingly. Other possible final poses for the swing foot are also shown (light dashed blue).

It is important to underline that the choice of the support foot pose as the specific body has also an additional advantage. In fact, since the support foot always lies on the ground, its pose can be completely defined in $SE(2)$, rather than $SE(3)$ (as for any other body of the humanoid robot). This implies a reduction of the dimension of the configuration space. Moreover, the partitioning that we made for the configuration vector in eq. (4.1) reflects the way the motion is generated within our framework. In particular, the support foot pose $\boldsymbol{q}_{\text{spt}}$ undergoes discrete displacements achieved through steps; on the other side, $\boldsymbol{q}_{\text{jnt}}$ changes with continuity. For this reason, the specific mechanism is well represented by the following *hybrid* motion model

$$\boldsymbol{q}^{k+1}_{\text{spt}} = \boldsymbol{q}^k_{\text{spt}} + \boldsymbol{A}(\boldsymbol{q}^k_{\text{spt}})\boldsymbol{u}^k_{\text{spt}} \tag{4.2}$$

$$\dot{\boldsymbol{q}}_{\text{jnt}}(t) = \boldsymbol{v}_{\text{jnt}}(t) \qquad t \in [t_k, t_{k+1}]. \tag{4.3}$$

This model describes the evolution of the robot configuration within a generic time interval $[t_k, t_{k+1}]$ in which the robot performs a stepping motion, i.e., a whole-body motion that produces also a displacement of the support foot (see Figure 4.1). In particular:

- $\boldsymbol{q}^k_{\text{spt}} = \boldsymbol{q}_{\text{spt}}(t_k)$ and $\boldsymbol{q}^{k+1}_{\text{spt}} = \boldsymbol{q}_{\text{spt}}(t_{k+1})$ are the planar poses of the support foot respectively at the start (t_k) and the end (t_{k+1}) of the time interval;
- $\boldsymbol{A}(\boldsymbol{q}^k_{\text{spt}})$ is the homogeneous transformation matrix from the support foot frame at t_k to the world frame;
- $\boldsymbol{u}^k_{\text{spt}}$ is the pose displacement of the support foot expressed in its frame;
- $\boldsymbol{v}_{\text{jnt}}$ is the velocity command for the humanoid joints.

Recall that we used the term hybrid for referring to the above-mentioned motion model. In fact, note that $\boldsymbol{q}_{\text{spt}}$ evolves discretely in $[t_k, t_{k+1}]$, i.e., it changes instantaneously at t_{k+1} when the

final pose of the support foot is defined. On the other side, $\boldsymbol{q}_{\mathrm{jnt}}$ evolves continuously, as we will see in Section 4.2.

The discrete nature of eq. (4.2) comes from the fact that a step needs a finite (non-zero) time $T_k = t_{k+1} - t_k$ to be performed. It is important to emphasize that the joint velocity $\boldsymbol{v}_{\mathrm{jnt}}$ and the support foot displacement $\boldsymbol{u}^k_{\mathrm{spt}}$ are not independent each other but they are highly coupled. In fact, humanoid motions are generated at joint level. There is no way to assign in a decoupled way a desired foot pose for one foot and a command for the joints. On the contrary, the joint motion should be generated in such a way the foot reaches the desired pose. In other words, any desired pose displacement of the support foot from t_k to t_{k+1} will be the result of the motion of the swing leg during the time interval. This will be appropriately handled by our motion generation scheme in Section 4.2.

Up to this point, we focus on stepping motions. The motion model (4.2-4.3) is also valid in case of *non-stepping* motions, i.e., motions where the robot changes its internal posture (i.e., it moves its joints) without moving its feet. In other words, the robot remains into the double support phase. In this case, $\boldsymbol{u}^k_{\mathrm{spt}} = \mathbf{0}$ and the support foot does not move. Moreover, the duration T_k may be chosen arbitrary.

For this reason, we will use the motion model (4.2-4.3) to describe any *elementary* (i.e., stepping or non-stepping) humanoid motion. Our solution will be a composition of elementary motions.

4.1.2 Task-constrained planning

As mentioned in the rest of the thesis, we are interested in solving the TCMP problem for a humanoid robot. Having introduced a proper motion model, we can turn our attention to the assigned task. In this chapter we will focus on manipulation tasks, expressed as the trajectory (position and, possibly, the orientation) for one hand of the humanoid robot. Tasks that may be described in this way include, e.g, opening a door, turning a valve or picking up an object.

Collect the task variables in a vector $\boldsymbol{y}$ which takes values in an appropriate space. The task coordinates are related to configuration coordinates by a forward kinematic map

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{q}_{\mathrm{spt}}, \boldsymbol{q}_{\mathrm{jnt}}) \ . \tag{4.4}$$

This kinematic map is supposed to be known in advance. As example, in the manipulation case for one hand, this would be the forward kinematics from the support foot to the hand, as depicted in Figure 4.2. Note that our approach is general, in the sense that the constraint can be assigned to any point of the robot. As it appears obvious from eq. (4.4), any point of the humanoid robot can be constrained, assuming that a proper forward kinematic map is defined. We decide to constrain one hand since manipulation is a common task for a humanoid robot. As we will see in the next chapter, the same framework holds in the case of a navigation task, where the point of interest is the middle point between the feet of the robot.

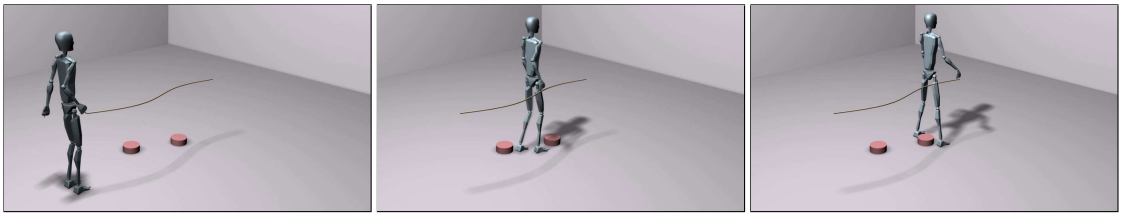


Figure 4.2 An example of solution for the TCMP problem. (left) The initial state of the robot with the task trajectory assigned for the right hand, aimed for a manipulation task. The motion planner should avoid the obstacles on the ground (red cylinders) while fulfilling the assigned task. (center) An intermediate posture assumed by the robot to fulfil the task and before approaching an obstacle. (right) The final state that completes the assigned task.

Suppose that a desired task trajectory $\boldsymbol{y}^*(t)$, $t \in [t_i, t_f]$, is assigned and that it is continuous and differentiable over time. It is assumed that the initial configuration $\boldsymbol{q}(t_i)$ of the robot is given, and that $\boldsymbol{f}(\boldsymbol{q}_{\text{spt}}(t_i), \boldsymbol{q}_{\text{jnt}}(t_i)) = \boldsymbol{y}^*(t_i)$.

Informally speaking, a solution of the TCMP problem for humanoid robots consists in finding a *feasible* (in a sense that will be clear soon) humanoid motion over $[t_i, t_f]$ that realizes the assigned task while avoiding collisions with workspace obstacles, whose geometry is known in advance. In general, a solution will consist of a sequence of elementary motions, either stepping or non-stepping, fully described by the hybrid model (4.2-4.3) and starting at $t_1 = t_i$.

It is worthless to mention that the choice of the appropriate sequence of $\boldsymbol{u}^k_{\text{spt}}$ among the others, at each step, is totally left to the planner. The same holds also for the duration T_k of each elementary motion. Pay attention that this does not mean that the planner should be greedy, i.e., it selects an action to take for a state and it will not never expand that state anymore. In fact, we will derive a probabilistic planner and the reader is referred to Section 4.3 for further details. For simplicity, and to keep the notation light, we consider all durations to be equal, i.e., $T_k = T$ for all k and $t_f - t_i = N \cdot T$. It is important to underline that this assumption is not needed and our framework can be naturally extended for elementary motions having different durations. The only reason of introducing such an assumption is to present in a more clear way the planner to the reader. This assumption will be removed in the next chapter, where a more general planner will be introduced.

Finally, we have all the ingredients for formally define a solution for the TCMP problem in case of humanoid robots. As already mentioned, once a motion is determined at the joint level, the resulting sequence of steps (and in particular, the placements $\boldsymbol{q}^k_{\text{spt}}$ of the support foot, for $k = 1, 2, \dots$) is completely determined. In other words, the robot is able to move by assigning values for its joint variables. When the joints move (in a proper way), the feet (in particular the swing foot that varies over time) move as well. This movement generates the footprints that completely defines the movements of the support and swing feet over time. Therefore, a solution consists of a trajectory $\boldsymbol{q}_{\text{jnt}}(t)$, $t \in [t_i, t_f]$, such that

- the assigned task trajectory is realized; that is,

$$\boldsymbol{y}(t) = \boldsymbol{f}(\boldsymbol{q}^k_{\mathrm{spt}}, \boldsymbol{q}_{\mathrm{jnt}}(t)) = \boldsymbol{y}^*(t),$$

  for all $t \in [t_k, t_{k+1}]$, $k \in [1, \ldots, N]$;

- self-collisions and collisions with workspace obstacles are avoided;

- position and velocity of the joints are within their bounds, respectively in the form $\boldsymbol{q}_{\mathrm{jnt,m}} < \boldsymbol{q}_{\mathrm{jnt}} < \boldsymbol{q}_{\mathrm{jnt,M}}$ and $\boldsymbol{v}_{\mathrm{jnt,m}} < \boldsymbol{v}_{\mathrm{jnt}} < \boldsymbol{v}_{\mathrm{jnt,M}}$;

- the robot is in equilibrium at all times.

The last two requirements express what we mean by *feasibility* of the humanoid motion. In particular, the last may be declined differently, depending on the desired kind of equilibrium. We will come back to this in the next section.

4.2 Motion generation

Our planner, which works in an iterative fashion, makes use of a *motion generator*. At each iteration, the motion generator is invoked to produce a feasible (in the sense described at the end of Section 4.1.2), collision-free elementary motion that realizes a portion of the assigned task trajectory.

Our motion generation scheme reflects the way the configuration vector $\boldsymbol{q}$ is divided. In fact, due to the hybrid nature of the humanoid model, motion for the two parts of the configuration $\boldsymbol{q}$ is generated using two interleaved procedures. The first is called *step generation* and it is in charge of deciding if and where to displace the support foot. Moreover, it produces an appropriate trajectories for the swing foot and the CoM of the humanoid. The second is referred to as *joint motion generation* and it computes the joint velocity commands so as to realize these trajectories and, at the same time, comply with a portion of the assigned manipulation task.

As already mentioned, note that these two steps are consecutive and, then, interleaved. The step generation (presented in Section 4.2.1) generates two major outputs, i.e., desired trajectories for the swing foot and for CoM. These trajectories should be fulfilled by the joint motion generation module (presented in Section 4.2.2), together with a portion of the assigned task.

4.2.1 Step generation

The step generation is invoked with reference to a humanoid configuration $\boldsymbol{q}^k = (\boldsymbol{q}^k_{\mathrm{spt}} \ \boldsymbol{q}^k_{\mathrm{int}})^T$ at time t_k. Its goal is to generate a trajectory for the swing foot and a trajectory for the CoM of the humanoid.

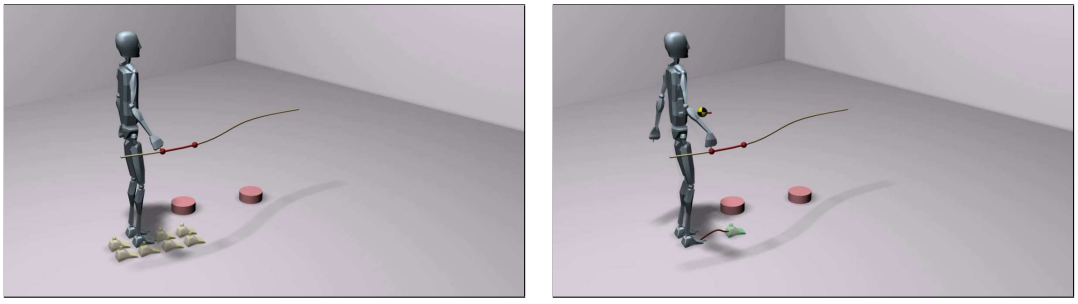


Figure 4.3 Example of the step generation procedure. (left) Once $\boldsymbol{q}_{\text{near}}$ has been selected, the planner has to choose among the set of right primitives. (right) Once the primitive has been chosen, the SPG is invoked in order to generate a feasible trajectory for the CoM and for the swing foot.

First, a displacement $\boldsymbol{u}^k_{\text{spt}}$ is chosen for the support foot from the following set of *step primitives*

$$\mathbf{0} \cup \left\{ \begin{pmatrix} \pm\alpha\,\delta_x \\ d_{\min} + \beta\,\delta_y \\ \pm\gamma\,\delta_\theta \end{pmatrix}, \alpha, \beta, \gamma \in \{0, 1, \ldots, M\} \right\} \tag{4.5}$$

where $\mathbf{0}$ is the null displacement corresponding to a non-stepping motion. In any other case, $\boldsymbol{u}^k_{\text{spt}}$ is the linear combination of three basic displacements

1. a forward displacement of length δ_x;
2. a lateral displacement of length δ_y (to which one should add $d_{\min}$, the minimum lateral distance between the feet);
3. a rotation by an angle δ_θ,

where M determines the size of the maximum displacement. As it is evident from eq. (4.5), there are multiple primitives for the same motion (e.g., the forward step). In fact, if one changes the value of α, the resulting primitive has a different step length in the forward direction. Moreover, α, β, γ might have a non-zero value at the same time. In this way, a generic step can be obtained, as shown in Figure 4.3. Their values should be defined in function of the robot gait capabilities, as we will do in the experimental Section 4.4. Obviously, the richer is the set of primitives, the more types of steps (and then movements) the robot can perform. On the other side, the motion planner has more primitives with a richer set. Then, when the step generation module is called, it is harder to find the most suitable primitive within the set, typically causing an increase of the planning time. For this reason, a compromise between the richness of the primitive set and the planning time should be found.

The primitive vector, $\boldsymbol{u}^k_{\text{spt}}$ might also be state-dependent, i.e. the support foot displacements may depend on the position of both feet or on the primitive that has generated $\boldsymbol{q}^k$. As example, the algorithm should avoid to take the same decision (for instance a forward step) twice or take a lateral step after a forward one. This is confirmed also by Figure 4.1 in which the lateral step in the right direction is neglected to avoid that the robot assumes a undesirable configuration.

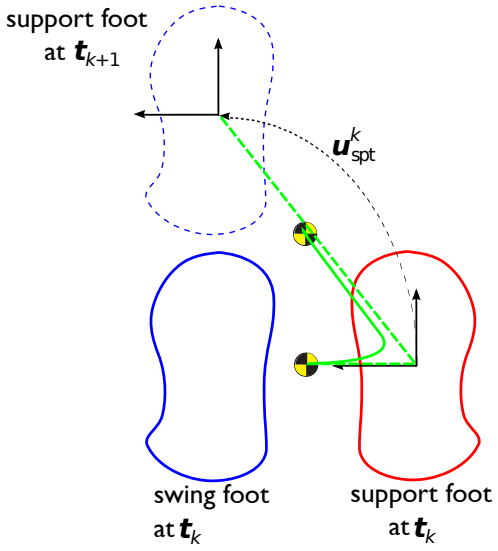


Figure 4.4 Example of computing the CoM given the support foot at t_k and a foot displacement $\boldsymbol{u}^k_{\text{spt}}$, that defines the location of the support foot at t_{k+1}. The green solid line represents the CoM trajectory.

To explore the space of possible solutions, the choice of $\boldsymbol{u}^k_{\text{spt}}$ in the set of primitives may be performed either randomly or based on an appropriate heuristic. For example, minimizing the angle between the final orientation of the support foot and the Cartesian tangent to the assigned task trajectory at t_k would lead to privileging foot placements that are locally aligned with the manipulation task. In our experiments, we choose the former, since we saw that this choice explores in a more uniform way the space of possible solutions.

After a foot displacement $\boldsymbol{u}^k_{\text{spt}}$ has been identified by selecting a primitive, the new pose of the support foot is uniquely identified through eq. (4.2). At this point, if $\boldsymbol{u}^k_{\text{spt}} \neq \boldsymbol{0}$, a *Stepping Pattern Generator*[3] (SPG for short) is invoked. The SPG is an external module that takes as input the current configuration $\boldsymbol{q}^k$ and the chosen displacement $\boldsymbol{u}^k_{\text{spt}}$ of the support foot, and provides as output (as shown in Figure 4.3)

1. a reference trajectory $\boldsymbol{z}^*_{\text{swg}}$ in $[t_k, t_{k+1}]$ for the swing foot (position and orientation);
2. a reference trajectory $\boldsymbol{z}^*_{\text{CoM}}$ in $[t_k, t_{k+1}]$ for the center of mass of the humanoid.

In this chapter, we say that the robot is in equilibrium in a configuration $\boldsymbol{q}^k$ if and only if the ground projection of the CoM (computed in $\boldsymbol{q}^k$) lies in the support polygon defined by the feet. This is known in literature as static equilibrium. Just for completeness, we recall that the support polygon is equal to the convex hull of the two feet, in case of double support. In case of single support, it reduces to the support foot sole only. The static equilibrium will be replaced

[3] We do not discuss in detail the structure of this module, which is usually available as part of the basic software suite of humanoid robots.

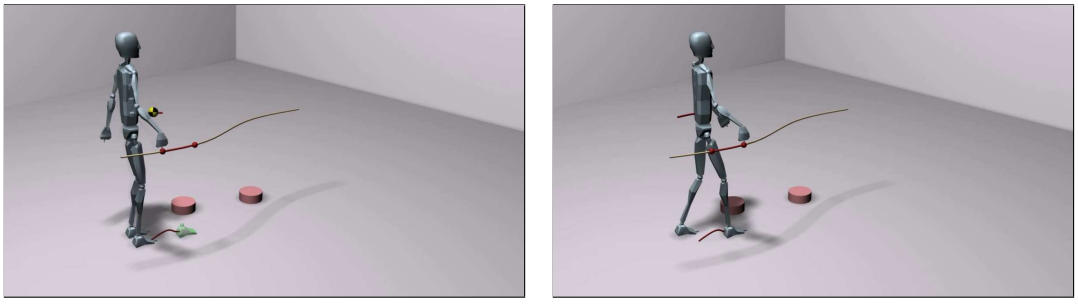


Figure 4.5 Example of the joint motion generation procedure. (left) The output of the step generation, i.e., a trajectory for the CoM and for the swing foot. (right) The generation of a feasible motion from the joint motion generator. Its aim is to fulfil these trajectory, as well a portion of the assigned task.

by the dynamic equilibrium in the next chapter, where a more general motion planner will be introduced.

The procedure (for a SPG) in charge of computing the CoM, given the footprints, is depicted in Figure 4.4. Roughly speaking, given $\boldsymbol{u}^k_{\text{spt}}$ and the actual poses of the swing foot and the support foot at t_k, the pose of the support foot is uniquely determined at t_{k+1}, as mentioned before. When performing a step (i.e., $\boldsymbol{u}^k_{\text{spt}} \neq \boldsymbol{0}$), the CoM is first moved on the support foot, in order to not fall when stepping. Then, the line joining the support foot at t_k and t_{k+1} is computed. The CoM trajectory can be easily computed by a way-point trajectory, as the solid green line in Figure 4.4, stopping in the middle point of this second line. The trajectory of the CoM is then smoothed by means of a way-point technique, in order to generate a continuous and differentiable trajectory, a feature that will be important in the next section.

If $\boldsymbol{u}^k_{\text{spt}} = \boldsymbol{0}$ (a non-stepping motion has been chosen) the SPG is not invoked. The reference trajectory for the swing foot in $[t_k, t_{k+1}]$ is simply $\boldsymbol{z}^*_{\text{swg}}(t) \equiv \boldsymbol{z}_{\text{swg}}(t_k)$, whereas the equilibrium constraint will be directly taken into account during the joint motion generation.

4.2.2 Joint motion generation

The joint motion generation is in charge of computing $\boldsymbol{q}_{\text{jnt}}$ in $[t_k, t_{k+1}]$. It starts from $\boldsymbol{q}^k = (\boldsymbol{q}^k_{\text{spt}} \;\; \boldsymbol{q}^k_{\text{jnt}})^T$ at t_k and its goal is to realize the portion of the assigned task trajectory $\boldsymbol{y}^*(t)$ between t_k and t_{k+1}. At the same time, it has to fulfil the reference trajectory $\boldsymbol{z}^*_{\text{swg}}$ for the swing foot and (if $\boldsymbol{u}^k_{\text{spt}} \neq \boldsymbol{0}$) the reference trajectory $\boldsymbol{z}^*_{\text{CoM}}$ for the center of mass in the same interval.

Assume $\boldsymbol{u}^k_{\text{spt}} \neq \boldsymbol{0}$. Let stack the three above-mentioned tasks and let define the augmented task vector $\boldsymbol{y}_a = (\boldsymbol{y}^T \;\; \boldsymbol{z}^T_{\text{swg}} \;\; \boldsymbol{z}^T_{\text{CoM}})^T$. Let $\boldsymbol{J}_a$ be the Jacobian matrix of $\boldsymbol{y}_a$ with respect to $\boldsymbol{q}_{\text{jnt}}$.

Since $\boldsymbol{q}^k_{\text{spt}}$ represents the support foot pose and it remains constant throughout the interval $[t_k, t_{k+1})$, all the kinematic chains described above are referred to the support foot reference frame. Note that, while the swing foot trajectory and the CoM trajectory are typically expressed in this reference frame, thanks to the procedure described in Section 4.2.1, an additional explanation is needed for the assigned task. This task is typically expressed in world reference frame, as introduced in Section 4.1.2. Since the support foot pose is known at t_k (from the step generation

module), it is trivial to compute the relative transformation w.r.t. this reference frame and, then, express all the tasks in the support foot reference frame.

Define the augmented task error as $\boldsymbol{e} = \boldsymbol{y}_a^* - \boldsymbol{y}_a$, where $\boldsymbol{y}_a^*(t)$ is the reference value of the augmented task between t_k and t_{k+1}. Joint velocities can be computed through

$$\boldsymbol{v}_{\text{jnt}} = \boldsymbol{J}_a^{\dagger}(\boldsymbol{q}_{\text{jnt}})\left(\dot{\boldsymbol{y}}_a^* + \boldsymbol{K}\boldsymbol{e}\right) + \left(\boldsymbol{I} - \boldsymbol{J}_a^{\dagger}(\boldsymbol{q}_{\text{jnt}})\boldsymbol{J}_a(\boldsymbol{q}_{\text{jnt}})\right)\boldsymbol{w}, \tag{4.6}$$

where $\boldsymbol{J}_a^{\dagger}$ is the Moore-Penrose pseudoinverse matrix of $\boldsymbol{J}_a$, $\boldsymbol{K}$ is a positive definite matrix and $\boldsymbol{w}$ is a n-vector whose value does not affect the execution of the tasks in $\boldsymbol{y}_a$. In fact, $\boldsymbol{I} - \boldsymbol{J}_a^{\dagger}(\boldsymbol{q}_{\text{jnt}})\boldsymbol{J}_a(\boldsymbol{q}_{\text{jnt}})$ is an orthogonal projection matrix in the null space of $\boldsymbol{J}_a$. It is easy to prove that, using eq. (4.6), $\dot{\boldsymbol{e}} = -\boldsymbol{K}\,\boldsymbol{e}$. In fact, since $\boldsymbol{e} = \boldsymbol{y}_a^* - \boldsymbol{y}_a$, its derivative is

$$\dot{\boldsymbol{e}} = \dot{\boldsymbol{y}}_a^* - \dot{\boldsymbol{y}}_a = \dot{\boldsymbol{y}}_a^* - \boldsymbol{J}_a(\boldsymbol{q}_{\text{jnt}})\dot{\boldsymbol{q}}_{\text{jnt}} = \dot{\boldsymbol{y}}_a^* - \boldsymbol{J}_a(\boldsymbol{q}_{\text{jnt}})\boldsymbol{v}_{\text{jnt}}\ , \tag{4.7}$$

since $\boldsymbol{v}_{\text{jnt}} = \dot{\boldsymbol{q}}_{\text{jnt}}$ from eq. (4.3). Substituting eq. (4.6) in eq. (4.7), one obtains $\dot{\boldsymbol{e}} = -\boldsymbol{K}\,\boldsymbol{e}$, i.e., the augmented task reference trajectory is exponentially stable. This means that, in the case $\boldsymbol{J}_a$ is full row-rank, all the three tasks are fulfilled. We prefer to use this approach with respect to a prioritized task framework because, in the latter, we have no guarantee of convergence for all the tasks in $\boldsymbol{y}_a$.

Regarding eq. (4.6), the last term to be discussed is $\boldsymbol{w}$. This is the term that explores the redundancy of the system. To uniformly explore this space, we set for the whole $[t_k, t_{k+1}]$ interval

$$\boldsymbol{w} = \boldsymbol{w}_{\text{rnd}}, \tag{4.8}$$

where $\boldsymbol{w}_{\text{rnd}}$ is a bounded-norm randomly generated n-vector.

Assume now $\boldsymbol{u}_{\text{spt}}^k = \boldsymbol{0}$, i.e., a non-stepping motion has been chosen. In this case, a trajectory for the CoM cannot be defined as in Figure 4.4, since the robot does not have to perform a step. For this reason, we let the CoM free to move in this case. Then, we refer to this primitive also as `free_CoM`. Recall that such a primitive corresponds to a double support phase with both feet in the same initial pose for all the time interval. Moreover, its duration is not defined a priori and this feature will be fundamental in our planner, topic of the next section. Since the CoM is not constrained anymore, eq. (4.6) is still used but $\boldsymbol{y}_a = (\boldsymbol{y}^T \quad \boldsymbol{z}_{\text{swg}}^T)^T$ is the augmented task vector. Furthermore, $\boldsymbol{z}_{\text{swg}}^*(t) = \boldsymbol{z}_{\text{swg}}(t_k), \forall t \in [t_k, t_{k+1}]$. The vector $\boldsymbol{w}$ is now chosen as

$$\boldsymbol{w} = -\eta \cdot \nabla_{\boldsymbol{q}_{\text{jnt}}} H(\boldsymbol{q}_{\text{jnt}}) + \boldsymbol{w}_{\text{rnd}}, \quad \eta > 0, \tag{4.9}$$

where $H(\boldsymbol{q}_{\text{jnt}})$ is the squared distance between the centroid of the support polygon and the projection of the center of mass on the ground. The meaning of eq. (4.9) should be clear: since the CoM is not constrained when $\boldsymbol{u}_{\text{spt}}^k = \boldsymbol{0}$ and one wants that it lies anyway in the support polygon, the first right term of eq. (4.9) moves $\boldsymbol{q}_{\text{jnt}}$ in the direction of the antigradient of H. In

other words, this term is aimed at keeping the center of mass as close as possible to the center of the support polygon, thereby preferring robot configurations that are statically further from instability. The second right term ($\boldsymbol{w}_{\mathrm{rnd}}$) is aimed to avoid deterministic actions. In fact, assuming a deterministic $\boldsymbol{w}$, eq. (4.6) is completely deterministic as well and the space of solutions in not explored at all. In the case in which the deterministic solution results in a collision with an obstacle in the environment, there would be no way to generate another solution. The random term has the aim of introducing different solutions for the same desired augmented task vector $\boldsymbol{y}_a^*$, then exploring the space of possible solutions.

From a robotic point of view, the joint motion generation is a Jacobian-based inverse kinematic scheme with an augmented task vector.

During the integration of eqs. (4.6–4.8), or eqs. (4.6–4.9), collision avoidance is continuously checked, together with position and velocity limits for the joints, in order to produce a feasible motion. For a non-stepping motion, the equilibrium is also checked, since the CoM is not constrained along a trajectory. If any of these conditions is violated, the current motion generation is prematurely terminated. Otherwise, integration stops when t_{k+1} is reached. At this point, a feasible joint motion $\boldsymbol{q}_{\mathrm{jnt}}(t)$, $t \in [t_k, t_{k+1}]$ has been generated and can be used by the planner.

The integration of eq. (4.6) can be performed using a numeric solver, whose integration step might be taken as small as possible in order to achieve an arbitrary accuracy of trajectory tracking. We have tested different integrators for this purpose, starting from the simple Euler one to the 4th order Runge-Kutta integrator. As it is well known, the former has the advantage of simplicity and, then, it is really fast from a computational point of view. The latter is slower but it has more accuracy than the former. Since our planner works offline, we decide to use the 4th order Runge-Kutta integrator. An important reason for which we need to integrate eq. (4.6) is that our platform accepts only joint position inputs, rather than joint velocity commands.

4.3 Planner overview

The proposed planner builds a tree $\mathcal{T}$ in the task-constrained configuration space

$$\mathcal{C}_{\mathrm{task}} = \{\boldsymbol{q} \in \mathcal{C} : \boldsymbol{f}(\boldsymbol{q}_{\mathrm{spt}}, \boldsymbol{q}_{\mathrm{jnt}}) = \boldsymbol{y}^*(t), t \in [t_i, t_f]\}$$

with the root at the initial configuration $\boldsymbol{q}(t_i)$. Nodes of the tree are configurations of the humanoid robot while arcs represent whole-body motions (both stepping or non-stepping) that join two adjacent nodes and should contain feasible motions. It is important to underline that both nodes and arcs are completely contained in $\mathcal{C}_{\mathrm{task}}$. Every node is associated to a time instant t_k, $k = 1, \ldots, N+1$. The tree structure means that multiple nodes may be associated to the same time instant t_k.

A generic iteration of the proposed planner is depicted in Figure 4.6. It implements a strategy (RRT-like) similar to the ones discussed in Section 3.2.

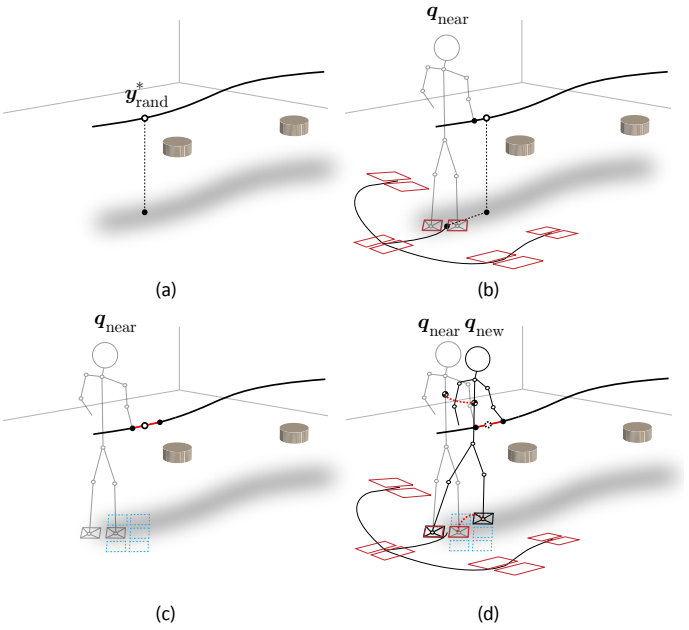


Figure 4.6 A generic iteration of the proposed planner. (a) A random sample is chosen from the assigned task trajectory and its projection on the ground is computed. (b) The most suitable configuration $\boldsymbol{q}_{\text{near}}$ is extracted from those in the tree using a probability inversely proportional to the distance between such projection and the midpoint between the feet. (c) The portion of the assigned task trajectory starting from $\boldsymbol{q}_{\text{near}}$ is extracted (red) and the step generator is called to choose a support foot displacement among the set of primitives (dashed blue). (d) The SPG produces reference trajectories for the swing foot and the center of mass (both dashed red) and the joint motion generator is invoked to produce a feasible elementary motion. If the motion is feasible, its final configuration $\boldsymbol{q}_{\text{new}}$ is added to the tree as node and the motion is added as arc.

The first step consists in extracting a 3D random sample, denoted as $\boldsymbol{y}^*_{\text{rand}}$, located on the assigned task trajectory. The ground projection of this point is then computed, as shown in Figure 4.6a. The next step is to compute the most suitable configuration, $\boldsymbol{q}_{\text{near}}$, defined as the configuration in the tree $\mathcal{T}$ picked with a probability inversely proportional to the Euclidean distance between such projection and the midpoint between the feet (Figure 4.6b). The idea behind this metric is to expand with more probability configurations where this distance is small. These configurations are less prone to failure during the expansion, since the robot is not so close to the boundary of its support polygon. In fact, it is reasonable to assume that configurations where this distance is large are likely to be close to its joint limits or to losing equilibrium. The reason for which we chose this metric instead of computing the usual nearest configuration is the completeness. In fact, assume to have two configurations associated with the same time instant t_k having different distances w.r.t. the ground projection. If the nearest criteria is used, the configuration with the larger distance will never be selected, compromising the completeness of the algorithm.

Once $\boldsymbol{q}_{\text{near}}$ has been selected, even the starting time t_k for the current iteration is identified as well. In fact, at the beginning, the tree $\mathcal{T}$ contains just the starting configuration $\boldsymbol{q}(t_i)$, associated with the starting time t_i. When a new configuration $\boldsymbol{q}_{\text{new}}$ is generated (as it will be described

next) and inserted, the time interval $[t_k, t_{k+1}]$ has been identified and $\boldsymbol{q}_{\text{new}}$ is associated with the time t_{k+1}. Then, at each iteration, the starting time t_k is always available whenever $\boldsymbol{q}_{\text{near}}$ has been identified.

At this point, the planner chooses a duration T_k for the current iteration. For simplicity, we assumed all durations to be equal, i.e., $T_k = T$ for all k and $t_f - t_i = N \cdot T$. Then, the time interval is uniquely identified as $[t_k, t_{k+1}]$, with $t_{k+1} = t_k + T$. Finally, the portion of the assigned task $\boldsymbol{y}^*(t)$ between t_k and t_{k+1} is acquired and the motion generation module is invoked. As explained in Section 4.2.1, the step generation module chooses a displacement $\boldsymbol{u}^k_{\text{spt}}$ for the support foot in the set of primitives in eq. (4.5). This procedure in depicted in Figure 4.6c. Then, the SPG is invoked and to produce reference trajectories for the swing foot and the CoM of the humanoid robot. In case $\boldsymbol{u}^k = \boldsymbol{0}$ (non-stepping), the SPG is not called since the CoM is free to move and the swing foot is simply constrained in its initial pose, as explained in Section 4.2.1. Finally, the joint motion generation is invoked to compute the joint motions, as mentioned in Section 4.2.2. This motion is able to fulfil the portion of the assigned task as well these reference trajectories in $[t_k, t_{k+1})$. If the motion is feasible, the final humanoid configuration $\boldsymbol{q}_{\text{new}}$ is added to the tree (Figure 4.6d) as node while the motion is added as arc connecting $\boldsymbol{q}_{\text{near}}$ and $\boldsymbol{q}_{\text{new}}$. On the contrary, the procedure is repeated.

We recall that a motion is feasible if the configurations that compose the motion never exceed the joint limits as well the joint velocity limits. In addition, they do not have to contain collisions with obstacles nor self-collisions, as clearly stated at the end of Section 4.1.2.

4.4 Planning experiments

The proposed planner has been implemented in V-REP (a robot simulator developed by Coppelia Robotics) on a MacBook Pro dual-core running at 2.66 GHz. The chosen robotic platform is NAO by Aldebaran Robotics. Section 4.4.1 will give an overview of the chosen robotic platform, while Section 4.4.2 does the same for the simulator. Finally, Section 4.4.3 is devoted to describe the planning experiments that validate and test the proposed planner.

4.4.1 The NAO robot

The NAO[5] is a small (58 cm) humanoid robot, developed by Aldebaran Robotics. It has 5 degrees of freedom in each leg, 5 in each arm, 1 in the pelvis, 2 in the neck, and 1 in each finger, for a total of 25 DOFs. From a sensor point of view, NAO has two cameras (top and down) on its head, as depicted in Figure 4.7. However, just one camera can be used at time, obtaining a monocular vision system. This because the two camera cannot be activated at the same time due to the power consumption that reduces a lot the autonomy of the robot. Moreover, the limited CPU (1.6 GHz) does not allow to process data streaming from both cameras, together with data coming from

[4]http://doc.aldebaran.com/2-1/family/nao_dcm/actuator_sensor_names.html
[5]https://www.aldebaran.com/en/humanoid-robot/nao-robot

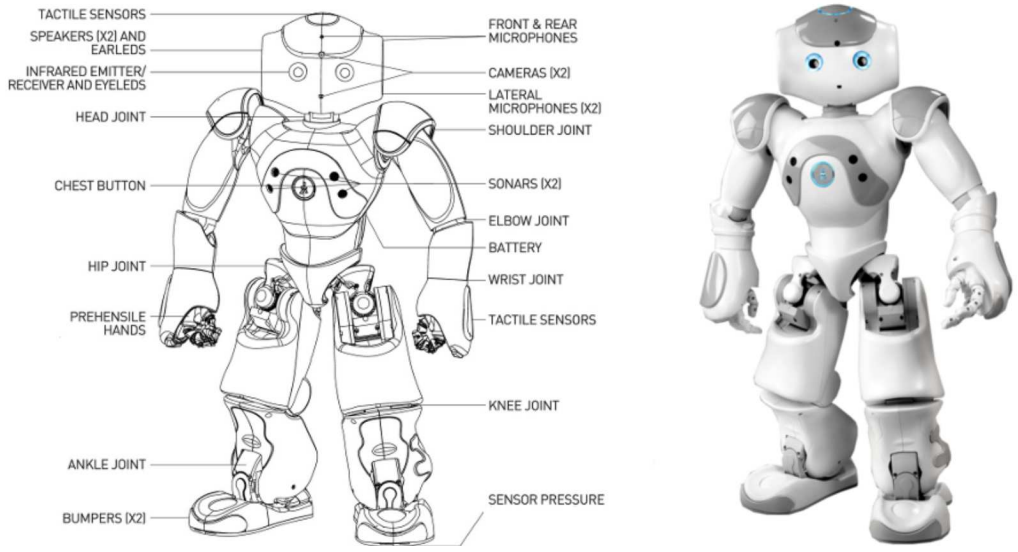


Figure 4.7 An overview of the NAO robot. (left) A schematic representation of NAO with its main features and sensors: two cameras, four directional microphones, two sonar range-finders, one IMU, two infrared emitters and receivers, three touch sensors, two bumpers, eight FSRs. Image courtesy of Aldebaran Robotics[4]. (right) a figure of the real NAO robot.

the other sensors. Finally, NAO has problems with heating. Since each of its 25 motors and each sensor need power, the more sensors are on, the more probability that the system goes to overheating.

The sensor equipment is completed by

- four directional microphones (on its head), whose aim is to detect audio input, such as vocal commands;
- two sonar range-finders (on its chest), whose aim is to compute the distance to the closest obstacle. Its frequency is 40 Hz, its resolution is 0.1 m and its detection range is $[0.25, 2.55]$ m;
- two infrared emitters and receivers;
- one Inertial Measurement Unit (IMU), composed by three axis accelerometer (1% precision with an acceleration of $\sim 2G$) and two axis gyrometers (5% precision with an angular speed of $\sim 500^\circ/s$). It is mounted approximately inside the robot torso and it provides linear accelerations of the main body, together with a measurement of the angular speed and an estimation of the orientation of the robot, i.e. the roll, pitch and yaw angles;
- three touch sensors (one over its head and one for each hand), whose goal is to detect physical interactions with the robot. They give an ON/OFF signal as output;
- two bumpers (in the tip of the feet). whose goal is to detect collision between the feet and an object. They give an ON/OFF signal as output;

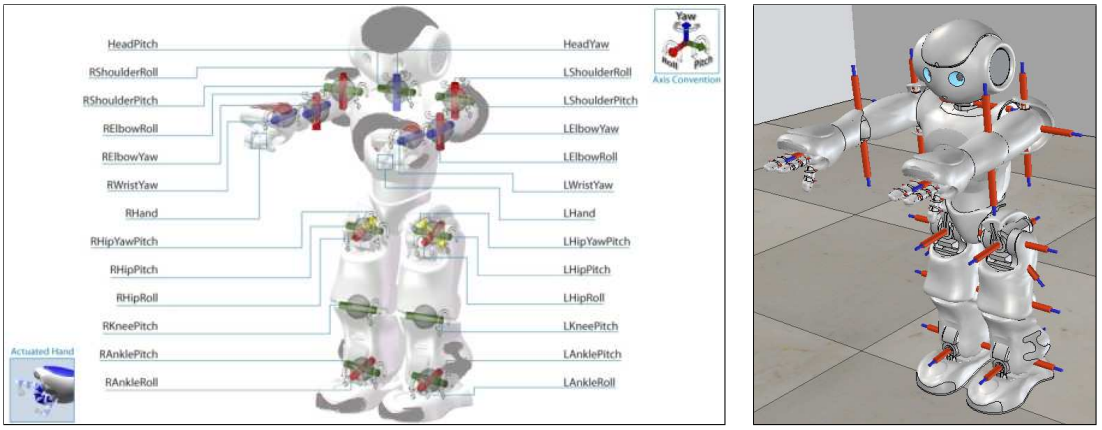
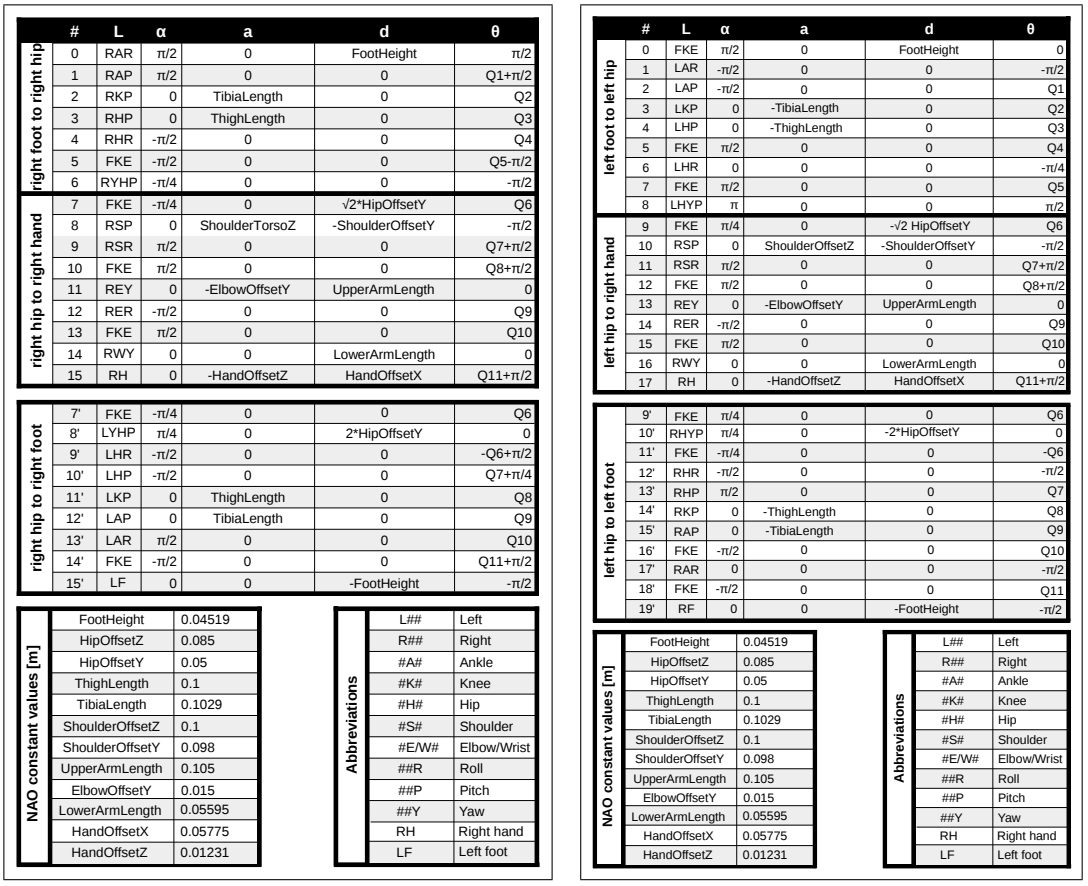


Figure 4.8 The list of joints of the humanoid robot NAO. (left) names of the joints of NAO. The first letter refers to the left (L) or right (R) leg or arm. (right) the joints of the NAO robot in the V-REP simulator.

- eight Force Sensitive Resistors (FSRs) (under its feet), whose goal is to measure a resistance change according to the pressure applied. Its range is $[0, 25]$ N

In order to run the motion planner proposed in Section 4.3, one needs some forward kinematic chains and their relative Jacobian matrices (to compute the augmented task vector $\boldsymbol{y}_a$ in eq. (4.6)). More in details, the following kinematic chains are requested: (i) from the support foot to the specific point of the robot that has to accomplish the assigned task (in our case, the right hand); (ii) from the support foot to the swing foot; (iii) from the support foot to the CoM.

In Figure 4.9, we report the kinematic chains starting from the support foot. We decided to report both kinematics starting from both feet (instead of single ones from a common point, e.g., the torso) since they are ready to be used in the framework presented in Section 4.3. An additional comment regarding the kinematics of NAO should be given. As a clever reader might notice, NAO has two particular joints. We are referring to RHYP and LHYP, i.e., the right and left hip yaw pitch joints. These are the two joints in the hip of the robot, pointing in a oblique direction, as shown in Figure 4.8. They are not independent, but highly coupled. In fact, just one motor commands these two joints, forcing them to assume a mirrored value at each time. This reflects in the kinematics expressed in Figure 4.9, where the RHYP and LHYP are reduced to a single joint value. In particular, this affects the kinematic chain that starts in one leg and ends into the other leg (i.e., the one for controlling the swing foot), since it is the only one that involves these two joints. The major effect relies in the Jacobian derivation. In case one wants to compute the analytical Jacobian, there is no additional work to do since it is sufficient to derive the forward kinematics assuming the same joint variable for the two joints. On the other side, if one wants to compute the geometric Jacobian, one has first to compute the Jacobian in the standard way, i.e. considering RHYP and LHYP as two different joints. The result is a Jacobian matrix $\bar{\boldsymbol{J}}$ having dimensions $n \times (n+1)$. For this reason, we introduce a matrix $\boldsymbol{E} \in \mathbb{R}^{(n+1)\times n}$



	#	L	α	a	d	θ
right foot to right hip	0	RAR	π/2	0	FootHeight	π/2
	1	RAP	π/2	0	0	Q1+π/2
	2	RKP	0	TibiaLength	0	Q2
	3	RHP	0	ThighLength	0	Q3
	4	RHR	-π/2	0	0	Q4
	5	FKE	-π/2	0	0	Q5-π/2
	6	RYHP	-π/4	0	0	-π/2
right hip to right hand	7	FKE	-π/4	0	√2*HipOffsetY	Q6
	8	RSP	0	ShoulderTorsoZ	-ShoulderOffsetY	-π/2
	9	RSR	π/2	0	0	Q7+π/2
	10	FKE	π/2	0	0	Q8+π/2
	11	REY	0	-ElbowOffsetY	UpperArmLength	0
	12	RER	-π/2	0	0	Q9
	13	FKE	π/2	0	0	Q10
	14	RWY	0	0	LowerArmLength	0
	15	RH	0	-HandOffsetZ	HandOffsetX	Q11+π/2
right hip to right foot	7'	FKE	-π/4	0	0	Q6
	8'	LYHP	π/4	0	2*HipOffsetY	0
	9'	LHR	-π/2	0	0	-Q6+π/2
	10'	LHP	-π/2	0	0	Q7+π/4
	11'	LKP	0	ThighLength	0	Q8
	12'	LAP	0	TibiaLength	0	Q9
	13'	LAR	π/2	0	0	Q10
	14'	FKE	-π/2	0	0	Q11+π/2
	15'	LF	0	0	-FootHeight	-π/2

NAO constant values [m]	
FootHeight	0.04519
HipOffsetZ	0.085
HipOffsetY	0.05
ThighLength	0.1
TibiaLength	0.1029
ShoulderOffsetZ	0.1
ShoulderOffsetY	0.098
UpperArmLength	0.105
ElbowOffsetY	0.015
LowerArmLength	0.05595
HandOffsetX	0.05775
HandOffsetZ	0.01231

Abbreviations	
L##	Left
R##	Right
#A#	Ankle
#K#	Knee
#H#	Hip
#S#	Shoulder
#E/W#	Elbow/Wrist
##R	Roll
##P	Pitch
##Y	Yaw
RH	Right hand
LF	Left foot

	#	L	α	a	d	θ
left foot to left hip	0	FKE	π/2	0	FootHeight	0
	1	LAR	-π/2	0	0	-π/2
	2	LAP	-π/2	0	0	Q1
	3	LKP	0	-TibiaLength	0	Q2
	4	LHP	0	-ThighLength	0	Q3
	5	FKE	π/2	0	0	Q4
	6	LHR	0	0	0	-π/4
	7	FKE	π/2	0	0	Q5
	8	LHYP	π	0	0	π/2
left hip to right hand	9	FKE	π/4	0	-√2 HipOffsetY	Q6
	10	RSP	0	ShoulderOffsetZ	-ShoulderOffsetY	-π/2
	11	RSR	π/2	0	0	Q7+π/2
	12	FKE	π/2	0	0	Q8+π/2
	13	REY	0	-ElbowOffsetY	UpperArmLength	0
	14	RER	-π/2	0	0	Q9
	15	FKE	π/2	0	0	Q10
	16	RWY	0	0	LowerArmLength	0
	17	RH	0	-HandOffsetZ	HandOffsetX	Q11+π/2
left hip to left foot	9'	FKE	π/4	0	0	Q6
	10'	RHYP	π/4	0	-2*HipOffsetY	0
	11'	FKE	-π/4	0	0	-Q6
	12'	RHR	-π/2	0	0	-π/2
	13'	RHP	π/2	0	0	Q7
	14'	RKP	0	-ThighLength	0	Q8
	15'	RAP	0	-TibiaLength	0	Q9
	16'	FKE	-π/2	0	0	Q10
	17'	RAR	0	0	0	-π/2
	18'	FKE	-π/2	0	0	Q11
	19'	RF	0	0	-FootHeight	-π/2

NAO constant values [m]	
FootHeight	0.04519
HipOffsetZ	0.085
HipOffsetY	0.05
ThighLength	0.1
TibiaLength	0.1029
ShoulderOffsetZ	0.1
ShoulderOffsetY	0.098
UpperArmLength	0.105
ElbowOffsetY	0.015
LowerArmLength	0.05595
HandOffsetX	0.05775
HandOffsetZ	0.01231

Abbreviations	
L##	Left
R##	Right
#A#	Ankle
#K#	Knee
#H#	Hip
#S#	Shoulder
#E/W#	Elbow/Wrist
##R	Roll
##P	Pitch
##Y	Yaw
RH	Right hand
LF	Left foot

Figure 4.9 The main kinematic chains needed for the running the TCMP framework presented in Section 4.3. The kinematics follows the well known Denavit-Hartenberg convention [25]. (left) Kinematic chains w.r.t. right foot support. (right) Kinematic chains w.r.t. left foot support. The constant values reported are referred to NAO V4.0, since we have this version of the robot in our lab.

that adapts the n-joint velocity vector $\dot{\boldsymbol{q}}$ for the Jacobian $\bar{\boldsymbol{J}}$

$$\dot{\boldsymbol{y}} = \bar{\boldsymbol{J}}\ \boldsymbol{E}\ \dot{\boldsymbol{q}} = \boldsymbol{J}\ \dot{\boldsymbol{q}}\ .$$

Suppose that $\dot{\tilde{\boldsymbol{q}}}$ is the $n+1$-vector that considers the two coupled joints as two different joints and assume i is the index of the first of these joints in $\dot{\tilde{\boldsymbol{q}}}$ while j is the index corresponding to the second. The matrix $\boldsymbol{E}$ is composed by the identity matrix with an additional row inserted at the j-th row. This row is composed by all zeros but with a 1 (or -1 if the joint is mirrored) at the i-th column. In the case of NAO, since the two joints are consecutive in the support to swing

foot kinematic chain, the result is

$$\boldsymbol{E} = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & -1 & & & & \\ & & & 1 & & & \\ & & & & \ddots & & \\ & & & & & 1 & \end{pmatrix},$$

where the missing components in the previous equation is completed by zero elements. To convince the reader, assume to have a simple robot having four joints, where the second and the third are forced to assume the same value. In this case, the first step would compute a 4×4 Jacobian matrix. The $\boldsymbol{E}$ matrix is equal to

$$\boldsymbol{E} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

where the second and third row expresses the kinematic constraint.

Using the Denavit-Hartenberg parameters, one can easily compute the forward kinematics and their relative Jacobian matrices for all the chains needed but the CoM one. This is true if one does not want to make the approximation of the CoM with the torso. Some authors, in fact, make this assumption, justifying it by saying that the torso is the body having the biggest mass and the contribution of other links can be neglected. It would be trivial to compute the CoM using this formulation, given the Denavit-Hartenberg parameters in Figure 4.9. In fact, it is sufficient to apply a rigid translation after computing the transformation matrix up to the last joint of the hip ($Q6$ in Figure 4.9). Even if this is acceptable in some cases (e.g., the robot does not move its arm or leg laterally), we prefer to use a more accurate estimation of the CoM.

More in details, our approach makes usage of the partial CoMs[7]. The main idea is to consider a single link and compute the CoM considering this link as the root of a kinematic chain. Rotating this vector in the support foot reference frame, its corresponding Jacobian matrix can be computed. If this procedure is repeated for each link, one is able to compute the CoM Jacobian matrix, as formally described below. In this procedure, we made the assumption that each link has a uniform mass distribution and that a joint links two rigid bodies. Formally, the partial CoM

[6]This image can be found at http://doc.aldebaran.com/1-14/family/nao_h25/links_h25.html

[7]A nice explanation of the partial CoMs is given at http://www.elysium-labs.com/robotics-corner/learn-robotics/biped-basics/com-jacobian/

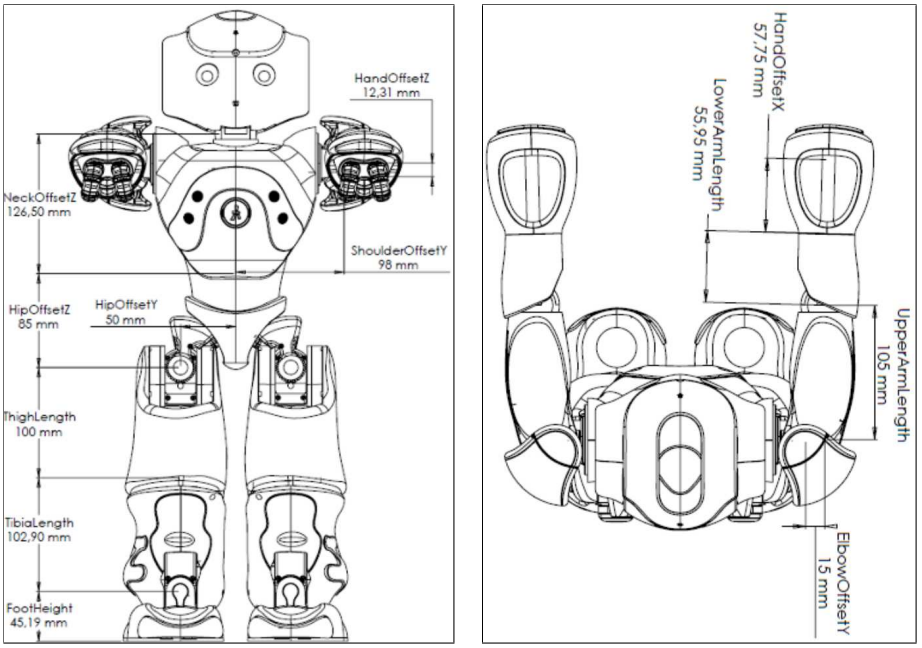


Figure 4.10 The dimensions of the NAO robot. The names are consistent with the NAO constant values reported in Figure 4.8. In this figure, the dimensions are reported in mm. This image is a courtesy of Aldebaran Robotics[6].

$\boldsymbol{z}^j_{CoM}$ of the j-th link is computed as

$$\boldsymbol{z}^j_{CoM} = \boldsymbol{R}_j \frac{\sum_{i=j}^{n} m_i \boldsymbol{y}_i}{\sum_{i=j}^{n} m_i} \ , \tag{4.10}$$

where $\boldsymbol{R}_j$ is the rotation matrix that rotates a vector expressed in the j-th link reference frame into the support foot reference frame, m_j is the mass of the j-th link and $\boldsymbol{y}_j$ is the forward kinematic position up to the base of the j-th link, computed using the Denavit-Hartenberg parameters in Figure 4.9. In other words, eq. (4.10) computes the CoM as seen by the j-th link and as depicted in Figure 4.11. The geometric Jacobian of the partial CoM is computed as

$$\boldsymbol{J}^j_{CoM} = \begin{pmatrix} \frac{\sum_{i=j}^{n} m_i}{\sum_{i=0}^{n} m_i} \left(\boldsymbol{r}_j \times \boldsymbol{z}^j_{CoM} \right) \\ \boldsymbol{r}_j \end{pmatrix} \ ,$$

where $\boldsymbol{r}_j$ is the third column of the rotation matrix $\boldsymbol{R}_j$. In the previous equation, we made the implicit assumption that the robot is composed by revolute joints, as for NAO. Stacking all the partial CoM Jacobian matrices, the resulting CoM Jacobian is obtained as

$$\boldsymbol{J}_{CoM} = \left[\boldsymbol{J}^1_{CoM} \dots \boldsymbol{J}^n_{CoM} \right] \ .$$

Note that, from the point of view of a link, a humanoid robot is composed by multiple kinematic chains ending in different robot points (e.g., the head, the two hands, the leg from the point of view of a link of one leg). The procedure described above can be trivially generalized for the

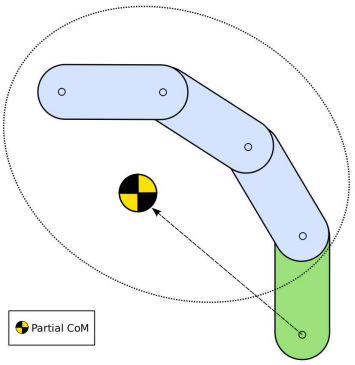


Figure 4.11 An example of computing a partial CoM for the link depicted in green.

humanoid case, as we did for the NAO robot.

There is also another way for computing the CoM and its relative Jacobian matrix by using a symbolic tool. Here, one has to compute the kinematic chain starting from the support foot reference frame to the CoM of each link. Again, we made the assumption that the mass distribution of each link is uniform such that the CoM can be easily modelled as a rigid translation in the link reference frame. The CoM is computed as

$$\boldsymbol{z}_{CoM} = \frac{\sum_{j=0}^{n} m_j \boldsymbol{z}_{CoM}^{j}}{\sum_{j=0}^{n} m_j} \ ,$$

where $\boldsymbol{z}_{CoM}^{j}$ comes from the Denavit-Hartenberg parameters in Figure 4.9, by applying a rigid translation to the kinematic chain that ends into the base of the j-th link. Once $\boldsymbol{z}_{CoM}$ is computed in a symbolic way, one can compute also its Jacobian by deriving this expression, again symbolically.

We implemented both the above-mentioned procedures. Even if they are valid, we prefer to use the second formulation since it seems to be computationally faster.

An important remark about NAO is that it can be controlled only at joint level, meaning that it does not accept joint velocity commands. This means that we need to integrate the joint velocities coming from eq. (4.6), in order to control the robot using the framework presented in Section 4.3.

4.4.2 The V-REP simulator

4.4.2.1 V-REP overview

Now that we have all the kinematic information for running the TCMP framework, explained in Section 4.3, we can turn our attention to V-REP[8], a simulator from Coppelia Robotics, that we use for our planning experiments. There are two main reasons for which we chose it

1. collision-checker. V-REP has an embedded collision checker, a fundamental tool for each motion planning algorithm;

[8] http://www.coppeliarobotics.com/

2. physics engine. V-REP allows to run both kinematic (i.e., with no physics engine running) and dynamic playbacks of planned motions.

V-REP is a cross-platform, open-source robot simulator with integrated development environment. It is based on a decentralized framework: each object/model can be individually controlled by an embedded script, a plugin, a ROS node, a remote API client or a custom solution. There are already a number of sensors and robots that comes with V-REP. One good point of this simulator is its completeness. In fact, manipulators, mobile robots and humanoid are in it, as well as vision, force and proximity sensors.

V-REP is composed by three central elements

1. scene objects (how you build a robot);
2. calculation modules (how you simulate the robot);
3. control mechanisms (how you can interface with the robot).

There are 12 scene objects that can be combined in V-REP. The most important ones (the ones that we use for building the NAO model) are

- shape. It is a rigid mesh objects that is composed of triangular faces. There are different types: (i) random shape: it not optimized but any mesh can be modelled with it; (ii) convex shape: slightly optimized; (iii) pure simple shape: the most optimized shape. Strongly suggested for dynamic simulation; (iv) heightfield: use for modelling terrain with different heights;
- joint. There are four different types: (i) revolute: rotational movement; (ii) prismatic: translational movement; (iii) screw: translational while rotational movement; (iv) spherical: three rotational movements;
- tree. The tree gives the way one builds a robot in V-REP. Each object has its own dynamic properties. Kinematic chains are built using a parent-child relationship. Two shapes are connected with a joint using this relationship. Finally, there is the possibility to decouple the visual and the dynamic part of shape. This is important because complex shapes can be just visually displayed while a simple convex shape can be used for the dynamic simulation. We use that approach for developing the NAO model in the simulator;
- camera. It is used for tracking an object while moving or for fixed visualization. There are two camera models: (i) perspective; (ii) orthographic projection.

Regarding the central modules, they can be grouped in

- forward/inverse kinematics. It can be used to find a inverse kinematic solution for any kinematic chain including redundant, closed and branched. It includes also different techniques

	embedded script	add-on	plugin	remote API	ROS node
control entity is internal	no	no	no	yes	yes
implementation difficulty	easiest	easiest	easy	easy	hard
programming language	LUA	LUA	C/C++	Matlab/Python/etc	any
# of APIs	> 280	> 270	> 400	> 100	> 100
simulation customization	no	yes	yes	no	no
code execution speed	slow	slow	fast	variable	variable
communication lag	none	none	none	yes	yes
API mechanism	regular API	regular API	regular API	remote API	ROS
API extensible	LUA scripts	LUA scripts	C++ code	remote API	ROS
control relies on	nothing	nothing	nothing	sockets[a]	sockets/ROS[b]
synchronous operation[c]	no delay	no delay	no delay	comm lag	comm lag
asynchronous operation	yes[d]	no	no	yes	yes

[a] the sockets use the remote API plugin
[b] the sockets use the ROS plugin and the ROS framework
[c] synchronous means that each simulation step runs synchronously with the control entity
[d] possible with threaded scripts

Table 4.1 A comparison between the different control mechanisms in V-REP.

for finding an inverse kinematic solution such as pseudoinverse (faster, more unstable) and damped least square (slower, more stable). When finding a solution, both joint limits and obstacle avoidance might be taken into account;

- minimum distance computation. The minimum distance can be computed between any meshes and it is very fast and optimized;

- dynamics. There are four embedded physics engine: (i) Bullet. An open source physics library featuring 3D collision detection, rigid body dynamics. Mostly used for gaming applications; (ii) ODE. An open source physics engine composed by rigid body dynamics and collision detection; (iii) Vortex. A closed source, commercial physics engine producing high fidelity physics simulations. It is a really good physics engine but the free licence is only for 20 seconds; (iv) Newton Dynamics. It is a cross-platform life-like physics simulation library. It implements a deterministic solver, which is not based on traditional LCP or iterative methods, but possesses the stability and speed of both respectively. In our experiments, we have used the ODE physics engine, since it is a good compromise between high fidelity physics simulations and payload introduced by running a simulation with a physics engine running (in addition there is no limitation for the simulation duration);

- collision detection. It can be computed between any mesh. One can select collidable objects. The calculation is an exact mesh-mesh (or collection of meshes) interference calculation. This is a fundamental tool for our motion planning algorithm, in particular when the joint motion generator is called (Section 4.2.2);

- path planning. It enables holonomic/non-holonomic path planning using a starting and a goal position, a set of obstacles and a robot model. Even if it seems promising for what we want to do, it is hard to generalize for a humanoid robot.

The last component to be described is the control mechanism. They are several ways for controlling a robot that can be grouped

- embedded script. It is written in LUA[9] code and its power relies in the simplicity with which one might control a robot. It is a script that can be attached to any scene object and it is fast (but not the fastest).

- add-on. As the previous approach, it is written in LUA. It allows to quickly customize the simulator itself. An add-on can start automatically and run in the background, or it can be called as functions. Unlike the previous approach, it is not linked to any model in the scene, but they are a global feature of the simulator;

- plugin. It is written in C++. It is the fastest way to control a robot in V-REP and it is the reason for which we used it in our experiments. It can be used for customizing the simulator and/or a particular simulation or for providing a simulation with custom LUA commands, and so are used in conjunction with scripts. Roughly speaking, a plugin is a shared library that is run at runtime by V-REP. This library should include three main function. The first is called *v_repStart*, that is an initialization function, called once at simulation start; the second, named *v_repEnd* is the function called by the simulator before exiting; the last one, *v_repMessage* is the loop function, i.e., the function called at each step by the simulator. It is used for writing your own functionalities and for detecting/creating/destroying object callbacks;

- remote API. It is aimed for controlling a model (or the simulator itself) from from an external application or a remote hardware (e.g. real robot, remote computer, etc.). The communication with V-REP is performed via socket communication in a way that reduces lag and network load (this appears as a black box for the user);

- ROS interface. This method allows to connect a model to be interfaced via ROS[10]. There are more than 30 publisher types as well as more than 25 subscriber types and extensions are possible.

The first 3 methods are internal, i.e. the control entity is internal while the remaining one are remote, meaning that the control entity is external. We reported in Table 4.1 a comparison between the control mechanisms, reporting their most important features.

We decided to develop our motion planning algorithm as plugins of V-REP. The main reason is for efficiency, since it is the fastest method to control a robot in the simulator. Moreover, it gives to opportunity to integrate the code with other libraries, e.g., Eigen[11] and one has the complete control of all the entities in the simulator at once.

[9] http://www.lua.org/
[10] http://www.ros.org/
[11] http://eigen.tuxfamily.org/

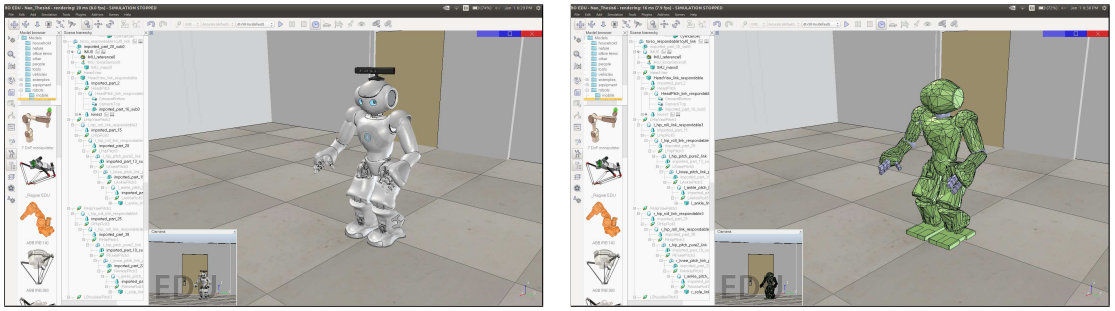


Figure 4.12 A snapshot of the NAO robot in V-REP. (left) Figure of the visual aspect of NAO, as it appears in the simulator. (right) Figure of the dynamic aspect of NAO, as it appears from the physics engine point of view.

4.4.2.2 NAO in V-REP

The first step is the development of the NAO model in V-REP, since such a robot was not in the simulator library, at the beginning. A model for the NAO robot already exists in a simulator called Webots[12]. However, the dynamic behaviour of the robot is not so reliable, in my humble opinion. Moreover, Webots is not an open-source software while V-REP does. A snapshot of the obtained model is depicted in Figure 4.12.

We tried to replicate, at the best of our knowledge, all the features and some of the sensors of the humanoid robot NAO $H25\ V4.0$. First, we got the meshes from Aldebaran Robotics and we optimized them for V-REP though a CAD tool (we use Blender[13] for this purpose). Then, we created the robot tree in V-REP developing, for each body, two layers. The first is a visual layer, i.e., it is how the body is seen from a sensor and/or by the user in V-REP. The second is a dynamic layer, i.e., it is how the body is seen by the physics engine. The reason for introducing two layers is the efficiency. In fact, a body might be simulated in principle as one single layer. However, a mesh, usually, it not convex and its propagation in a physics engine has a huge computational payload. Using the two layers is a good compromise between visual appearance and computational efficiency in a physics propagation. The result is shown in Figure 4.12, where the two layers are shown. From a user point of view, the robot appears as in the left part of Figure 4.12, while the physics engine propagates the robot depicted in the right part of Figure 4.12.

Note that the dimensions[14] were taken from the official NAO documentation, as well the joint limits[15] and the masses and the inertial matrices[16] for each body. We also simulated the motors by introducing a PID controller on each joint. When a desired value is given to a joint, the joint cannot apply it instantaneously, but it is reached through a PID controller, as for the real robot.

We simulated also some of the sensors the real robot is equipped with. We have to say that these sensors were just native with V-REP. We just applied them for NAO. In details, we set a

[12] https://www.cyberbotics.com/
[13] https://www.blender.org/
[14] http://doc.aldebaran.com/1-14/family/nao_h25/dimensions_h25.html
[15] http://doc.aldebaran.com/1-14/family/nao_h25/joints_h25.html
[16] http://doc.aldebaran.com/1-14/family/robots/masses_robot.html

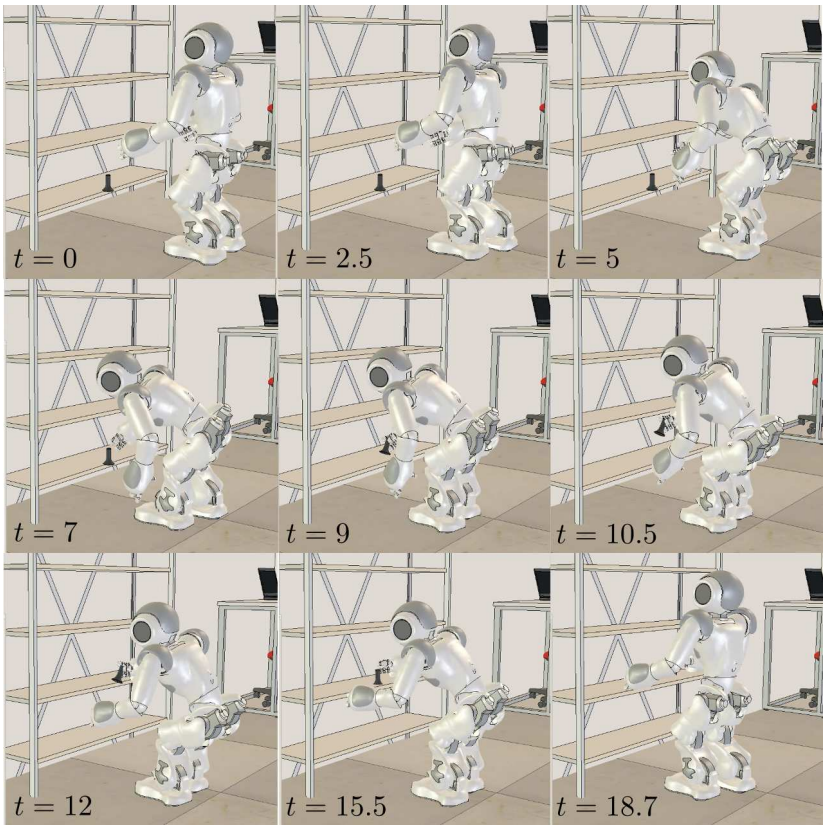


Figure 4.13 Pick and place: snapshots from a solution.

IMU on the torso of the robot, the force sensors under the feet in order to measure the pressure and the two cameras on the robot head.

4.4.3 Planning experiments

The planner proposed in Section 4.3 has been validated in V-REP, the simulator described in Section 4.4.2. The tests were performed on a MacBook Pro dual-core running at 2.66 GHz.

We have performed several experiments but here we report just two of them. In the first experiment, the robot has to pick up an object from a lower shelf of a bookcase and place it on an upper shelf. To this purpose, a trajectory is assigned for the right hand of the humanoid robot. The results are shown in Section 4.4.3.1. In a second experiment, the robot has to open a door. The constrained point is again the right hand of the humanoid. The results are shown in Section 4.4.3.2.

In both experiments, we have used the same parameter set. For step generation, uniform probability is used to extract support foot displacements from the set of primitives, in which $\delta_x = 0.03$ m, $d_{\min} = 0.1$ m, $\delta_y = 0.01$ m, $\delta_\theta = 7.5^\circ$ and $M = 2$. For joint motion generation, we use $\boldsymbol{K} = \mathrm{diag}\{2, 2, 2\}$ in eq. (4.6) and $\eta = 1.6$ in eq. (4.9), while $\boldsymbol{w}_{\mathrm{rnd}}$ in eqs. (4.8-4.9) is generated using uniform probability and a norm limit at 0.4 rad/sec. Numerical integration of joint velocities is performed with a 4th order Runge-Kutta algorithm using a step size of 0.05 s.

data	pick and place	opening a door
planning time (s)	5.41	2.61
tree size (# nodes)	81.6	55
mean task error (m)	$4.44{\cdot}10^{-4}$	$4.2761{\cdot}10^{-4}$

Table 4.2 Planner performance at a glance.

To further validate the proposed method, we have performed (for each proposed experiment) a dynamic playback of the planned motions in V-REP; i.e., we have used the generated joint trajectories as reference signals for the joint-level robot controllers and enabled a full physical simulation of the humanoid, including multibody dynamics and interaction with the environment (foot contact, object grasping and releasing). The obtained NAO motions are virtually identical to the reference motions; in particular, this confirms that static equilibrium is effectively achieved[17].

4.4.3.1 Picking and placing an object

As mentioned before, the goal of the first scenario is to pick an object from a lower shelf of a bookcase and place it on an upper shelf of the same bookcase. The trajectory assigned for the right hand lasts 18 s and the total length of the task path is 0.69 m. The duration of elementary motions is $T = 2$ s. Few snapshots of one solution found by our planner are depicted in Figure 4.13.

Since the target object is far enough from the robot, the planner needs to generate multiple steps to reach it. For this reason, it moves from its initial pose, performing four steps before grasping the object at $t = 9$ s. The object is then placed on the upper shelf at $t = 15.5$ s. Note that the robot grasps and releases the object in double support, using $\boldsymbol{u}^k_{\text{spt}} = \mathbf{0}$ in these phases. It is important to underline that the choice of which primitive to use is totally left to the planner and that step motions appear naturally, without introducing any form of biasing. Moreover, collisions with the bookcase are carefully avoided. The object is finally released at $t = 15.5$ s. The robot performs a final step completing the task at 18 s. Finally, it performs a brief self-motion that lasts 0.7 s. This is done for achieving the best possible final posture from the viewpoint of static balance. Results are reported in Table 4.2, where some data are collected in order to evaluate the performance of the proposed planner. It reports the time needed by the planner to generate a solution (planning time in Table 4.2), the number of nodes contained in the final tree (tree size in Table 4.2) and the mean value of the norm of the task error (mean task error in Table 4.2) throughout the duration of the task (not only at nodes). Since our planner is randomized, these data are averaged over 20 executions of the planner for each problem. As one might notice, the assigned task trajectory is fulfilled with high-precision. This precision can be even increased if one decreases the step size of the Runge-Kutta integrator or if one increases

[17] For further details, see also `http://www.dis.uniroma1.it/~labrob/research/HumWBPlan.html`, where there is also a video of the proposed approach.

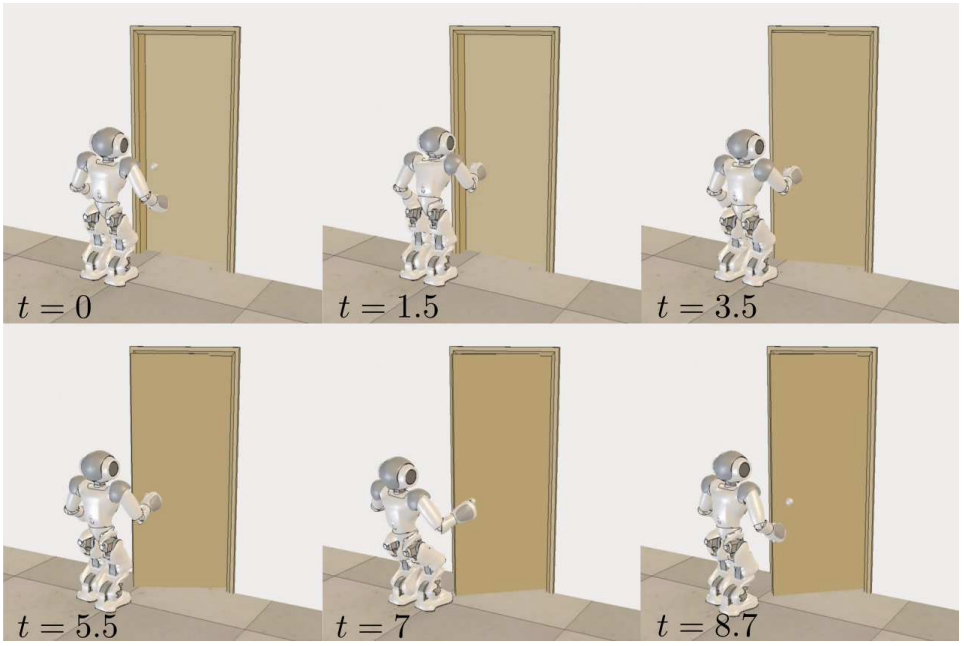


Figure 4.14 Open a door: snapshots from a solution.

the gains in $\boldsymbol{K}$. Moreover, if one substitutes the 4th order Runge-Kutta integrator with a simpler one, e.g. Euler, the planning time decreases. However, we believe that our settings are a good compromise between accuracy and planning time.

4.4.3.2 Opening a door

In the second scenario, the robot has to open a door by grasping its knob and moving it along an arc of $45°$. The duration of this task is 8 s and the length of the task path is 0.59 m. In this experiment, T was set at 1.6 s. All other parameters used by the planner are set to the same values as declared in Section 4.4.3. Figure 4.14 reports some snapshots from a solution found by our planner. At $t = 0$ s, the robot is standing in front of a door and the knob is in the workspace of the robot. For this reason, NAO does not have to take steps to grasp the knob (at $t = 1.5$ s) with its right arm. Since the feet are really close to the door, the latter cannot be opened standing in the initial pose. In fact, the robot has to take few backward steps between $t = 1.5$ s and $t = 7$ s in order to be able to open the door without colliding with it. Again, we remind that it is the planner that naturally generates the steps and the user does not provide any form of biasing. At $t = 7$ s, the door opening phase ends. Then, the robot releases the knob and takes one more backward step. Also in this case, the plan is concluded by a self-motion, as in the previous experiment. Table 4.2 reports the same data (planning time, tree size and mean task error) as for the previous scenario.

For the sake of clarity, it is worth noting that in both experiments the grasping action is purely demonstrative. Taking into account the physics of grasping is out of the scope of this thesis.

Note how both the planning time and the tree size are larger for the first problem than for the second. The reason for this is twofold. First, the task duration for the pick and place scenario

is longer. Second, the geometry of that scene requires the robot to stretch its arm to reach the object on the lower shelf. As a consequence, many candidate configurations are discarded by the planner for violating the joint limits or colliding with the bookcase, and therefore the planning time increases. In both cases, the task accuracy is very high and insensitive to the complexity of the problem.

4.5 Conclusions

In this chapter, we have presented a framework for solving the problem of planning the motion of a humanoid robot that must execute a manipulation task, possibly requiring stepping, in an environment cluttered by obstacles. The proposed method explores the submanifold of the configuration space that is admissible with respect to the assigned task and at the same time satisfies other constraints, including humanoid equilibrium. The exploration tree is expanded using a hybrid scheme that simultaneously generates footsteps and whole-body motions. The algorithm has been implemented for the humanoid robot NAO and validated through dynamic playback in the V-REP environment.

A distinctive feature of the developed planner is that, differently from the existing literature, its application is not limited to "regulation" tasks, i.e., tasks defined only in terms of a final desired value, such as a grasp posture for one hand. Indeed, by handling tasks that are specified through actual trajectories, we can allow the robot to perform more complicated operations, such as opening a door. On the other hand, if only $\boldsymbol{y}^*(t_f)$ is assigned, a suitable *approach trajectory* in the task space can be easily designed and incorporated in the assigned task to recover our problem setting. Another possibility is to simply set $\boldsymbol{y}^*(t) = \boldsymbol{y}^*(t_f)$ for all t; in this case, the motion generation scheme (4.6) will naturally produce as approach trajectory a linear motion in the task space with exponential convergence speed. This feature will be formally explored in the next chapter.

Another interesting aspect of our approach is its independence from the Stepping Pattern Generator, which is in fact assumed to be external to the planner. This means that both statically and dynamically balanced walking can be embedded in our plan, without the need of any post-processing phase. An actual Walking Pattern Generator may be also included to produce longer sequences of stepping motions directly; this can be desirable when the assigned task implicitly requires long transfers ('open the door at the end of the corridor').

Although we have essentially focused on manipulation tasks, we emphasize that the proposed framework is general and therefore can be applied also to tasks of different nature, such as navigation. This, together with the introduction of the CoM movement primitives, will be the subject of the next chapter that presents another motion planner that generalizes the one presented so far.

The planner presented in this chapter has been published in [17][18].

[18] For further details, see also `http://www.dis.uniroma1.it/~labrob/research/HumWBPlan.html`, where there is also a video of the proposed approach.

Chapter 5

Task-oriented whole-body motion planning for humanoids based on CoM movement primitives

This chapter describes another motion planner that generalizes the one presented in Chapter 4. The idea is to replace the foot displacements that are the output of the step generation module presented in Section 4.2.1 with a more general concept, the movements of the CoM. These are movements associated with typical human actions, such as static walking, dynamic walking, and more. We assume that a catalogue of CoM movement primitives has been precomputed and it is available to the planner. A solution is composed by concatenating whole-body motions that fulfil these primitives and, simultaneously, portions of the assigned task.

An important aspect under which the new planner improves over the previous is that it can indifferently handle tasks specified as trajectories (e.g., opening a door) or as simple destinations in the task space. Moreover, the task may be assigned as a single operation (e.g., 'grasp this object', 'open the door handle') or a composite sequence of navigation and manipulation actions ('take the object on that table and bring it in the other room').

Results obtained on a NAO humanoid will confirm the higher versatility gained by the use of CoM movements primitives. For example, we obtain plans that automatically toggle between dynamic and static walking gaits when required by the characteristics of the environment (e.g., obstacles) or the task itself. In addition, the possibility of assigning destination points allows a simple definition of composite tasks. Moreover, since each primitive implicitly encapsulate the information about feasible next primitives, we shrink the size of search space, hence the planner performs better in terms of computational complexity.

Using primitives in humanoid motion planning is not new in the literature. These have been exploited, for example, in [43, 44] for planning motions on varied terrain. However, those primitives are actually whole-body motions (i.e., they specify the motion of all joints) whereas in our approach they describe only the trajectory of the CoM and can therefore give rise to different robot movements as required by the various phases of the plan; this results in a higher *plasticity*

of our planning method. Moreover, the planners [43, 44] cannot directly handle the case in which a task (e.g., manipulation) is assigned to the robot.

Some of the concepts we need to explain the motion planner were already introduced in Chapter 4. For this reason, we avoid to explain again these concepts by providing reminders to appropriate sections of the previous chapter. However, our goal is to provide a self-contained chapter. This chapter is organized as follows. In Section 5.1 the humanoid motion model is formalized since it changes w.r.t. the one presented in Section 4.1; the motion generation based on CoM movement primitives is described in Section 5.2, while a randomized planner is shown in Section 5.3. Finally, motion planning results are reported in Section 5.4 and the whole chapter is discussed in Section 5.5.

5.1 Problem formulation

As for Section 4.1, we first introduce a humanoid motion model, and then we discuss the nature of the considered tasks.

5.1.1 Humanoid motion model

As explained in Section 4.1.1, in order to specify a configuration of a free-flying humanoid, one should assign the n values of the joint angles together with the pose (position and orientation) of a reference frame linked to one of the robot bodies. Instead of linking this reference frame to the support foot (as we did in Section 4.1.1), we choose to attach it to the center of mass (CoM in the following) and oriented as the torso. The reason for choosing the torso for the orientation is that the CoM is a virtual point that does not provide any form of orientation. The orientation is represented as a unit norm quaternion. Furthermore, we motivate the choice of the CoM as part of the configuration vector by saying that it will be helpful when the CoM movement primitives will be introduced in Section 5.2. A configuration will be then defined as follows

$$\boldsymbol{q} = \begin{pmatrix} \boldsymbol{q}_{\mathrm{CoM}} \\ \boldsymbol{q}_{\mathrm{jnt}} \end{pmatrix} , \tag{5.1}$$

where $\boldsymbol{q}_{\mathrm{CoM}} \in SE(3)$ is the pose of the CoM frame and $\boldsymbol{q}_{\mathrm{jnt}} \in \mathcal{C}_{\mathrm{jnt}}$ is the n-vector of joint angles. The humanoid configuration space $SE(3) \times \mathcal{C}_{\mathrm{jnt}}$ has thus dimension $n + 6$.

As for the planner presented in Section 4.3, the way the configuration vector is partitioned reflects how we generate the motion within our motion planner. For the CoM, we shall concatenate whole movements (i.e., subtrajectories) rather than defining instantaneous motions (as for the support foot). These subtrajectories will be extracted from a catalogue of *CoM movement primitives* that are associated to typical human actions such as walking, jumping, squatting, etc. An example of these primitives are depicted in Figure 5.1. Each primitive contains a movement for the CoM. Moreover, it may actually specify the trajectory of other points of the robot: for example, a stepping primitive will include also the trajectory of the swing foot. Note that selecting

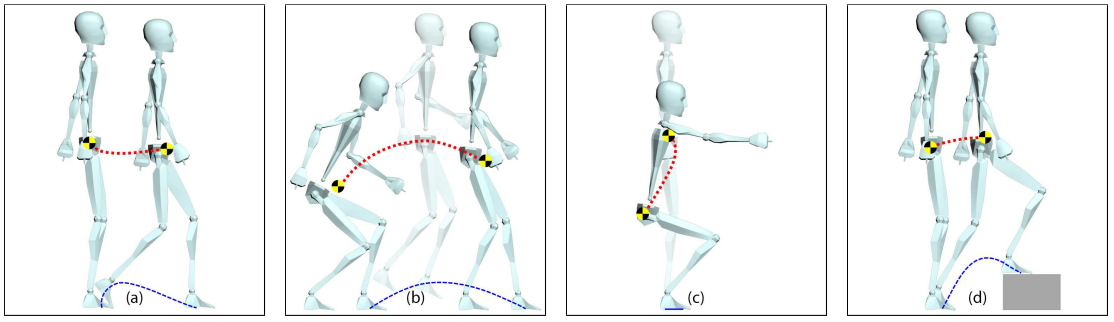


Figure 5.1 Examples of CoM movement primitives: (a) stepping (b) jumping (c) squatting (d) climbing. Each primitive specifies a trajectory for the CoM (red, dotted) and possibly other points of the robot, such as for the swing foot (blue, dashed). However, the planner can freely choose the whole-body motion among the infinite that are compatible with the primitive.

a particular CoM primitive does not specify the whole-body motion, which can be freely chosen[1] by the planner among those compatible with the primitive. This is a key point of the proposed framework and it should be discussed in more details: a CoM primitive contains a desired movement regarding the position of the CoM and the orientation of the torso, as mentioned before. Moreover, in case the primitive is a stepping primitive, it contains also the position and the orientation of the swing foot. Then, it may contain information regarding other points of the robot (in addition to the CoM) even if it is called CoM movement primitive. The reason we named it in this way is that it contains information regarding how to move the CoM. It is important to underline that each primitive contains also the duration needed to perform the primitive.

Once a CoM movement primitive has been selected, the displacement of $\boldsymbol{q}_{\mathrm{CoM}}$ is defined throughout its whole duration. At this point, the instantaneous motion of the joint coordinates $\boldsymbol{q}_{\mathrm{jnt}}$ can be chosen so as to realize the chosen primitive, together with other planning requirements. This is well-described by the following hybrid model

$$\boldsymbol{q}_{\mathrm{CoM}}(t) = \boldsymbol{q}^k_{\mathrm{CoM}} \oplus \boldsymbol{A}(\boldsymbol{q}^k_{\mathrm{CoM}})\boldsymbol{u}^k_{\mathrm{CoM}}(t) \tag{5.2}$$

$$\dot{\boldsymbol{q}}_{\mathrm{jnt}}(t) = \boldsymbol{v}_{\mathrm{jnt}}(t) \tag{5.3}$$

with $t \in [t_k, t_{k+1}]$, an interval in which the CoM performs a certain primitive movement of duration $T_k = t_{k+1} - t_k$. In the previous equations, $\boldsymbol{q}^k_{\mathrm{CoM}} = \boldsymbol{q}_{\mathrm{CoM}}(t_k)$ is the pose of the CoM reference frame at t_k, $\boldsymbol{u}^k_{\mathrm{CoM}}(t)$ is the pose displacement of the CoM frame at t relative to the pose at t_k, $\boldsymbol{v}_{\mathrm{jnt}}$ is the velocity input vector for the humanoid joints and $\oplus$ is an operator that composes two poses[2], i.e., it sums the two position vectors while it is equal to the quaternion product operator for the quaternions[3]. Finally, $\boldsymbol{A}(\boldsymbol{q}^k_{\mathrm{CoM}})$ is a transformation matrix from the

[1] In particular, repetition of the same primitive (e.g., a step) in different parts of the plan will correspond in general to different whole-body motions, depending on the local task history and obstacle placement.

[2] Here, we suppose that a pose $\boldsymbol{q}_{\mathrm{CoM}}$ is a vector composed by 7 elements: the first three are the position while the last four are the quaternion (with its unit norm constraint).

[3] We recall that we chose quaternions for representing the orientations of reference frames.

CoM frame at t_k to the world frame. In formula

$$\boldsymbol{A}(\boldsymbol{q}^k_{\mathrm{CoM}}) = \begin{pmatrix} \boldsymbol{R}(\boldsymbol{q}^k_{\mathrm{CoM}}) & \boldsymbol{0}_{3\times 4} \\ \boldsymbol{0}_{4\times 3} & \boldsymbol{I}_{4\times 4} \end{pmatrix},$$

where $\boldsymbol{R}(\boldsymbol{q}^k_{\mathrm{CoM}})$ is the rotation matrix that rotates the position component of $\boldsymbol{u}^k_{\mathrm{CoM}}(t)$ into the world frame.

The motion model in eqs. (5.2–5.3) is hybrid. In fact, the first equation is algebraic, because the CoM motion is generated by patching whole subtrajectories extracted from the catalogue of primitives; in fact, the primitive chosen for application at t_k specifies the history of the relative pose displacement $\boldsymbol{u}^k_{\mathrm{CoM}}(t)$ for all $t \in [t_k, t_{k+1}]$. The second equation is differential, as joint variables are changed instantaneously to track the CoM motion and pursue other tasks.

Note that the motion model in eqs. (5.2–5.3) has a different hybrid nature w.r.t. the one described in eqs. (4.2-4.3). In fact, even if the differential part is the same in both motion models (eq. (4.3) and eq. (5.3) are the same), we recall that the motion model in eqs. (4.2–4.3) is hybrid due to the discrete nature of the support foot pose $\boldsymbol{q}_{\mathrm{spt}}$ dynamics, while the one in eqs. (5.2–5.3) is hybrid for the algebraic nature of the pose $\boldsymbol{q}_{\mathrm{CoM}}$ of the CoM reference frame.

It should be kept in mind that the CoM displacement $\boldsymbol{u}^k_{\mathrm{CoM}}(t)$ depends on the history of the joint velocities $\boldsymbol{v}_{\mathrm{jnt}}(t)$ in $[t_k, t]$ up to time t. Indeed, any motion of the humanoid, including that of the CoM, is generated at the joint level. In other words, eqs. (5.2–5.3) are not independent but they are highly coupled. This fact must be appropriately taken into account within our motion generation scheme (see Section 5.2).

5.1.2 Task-contrained planning

Since a motion model is now available, we can formally introduce the task. Its definition is very similar to the one given in Section 4.1.2 with some minor modifications. Here, we want to generalize the concept of a task. Suppose that a task is described as the trajectory for the position (and possibly orientation) of a specific point (body) of the humanoid, as in Section 4.1.2. For instance, a manipulation task may be specified as a trajectory assigned to one hand, while a navigation task may be assigned in terms of motion of the midpoint between the feet. This simple viewpoint makes it very easy to translate a task from natural language (‘pick up that object and bring it to me’) to an assignment that can be directly used by our planner. We emphasize that the task may be a geometric path rather than a trajectory, and in particular it may reduce to a goal position in task space. The proposed planner works without any modification in these cases, as shown in Section 5.4.

Then, the task is generalized in the sense that it might be expressed as a trajectory (as for the motion planner described in the previous chapter) but also as a geometric path or even a goal position in task space. Moreover, we will show how composite tasks can be handled in our

framework, by combining tasks of different nature (e.g., manipulation and navigation tasks. See Section 5.4 for an example of a composite task).

Formally, collect the task coordinates in a vector $\boldsymbol{y}$ taking values in an appropriate space. The following kinematic map links task coordinates with configuration coordinates

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{q}_{\mathrm{CoM}}, \boldsymbol{q}_{\mathrm{jnt}}).$$

Suppose that a desired task trajectory $\boldsymbol{y}^*(t)$, $t \in [t_i, t_f]$, is assigned. In case the trajectory is assigned via a geometric path, one has to replace t with a path parameter s. On the other side, in case one wants to assign the task as a desired task set-point, its definition degenerates to a single point $\boldsymbol{y}^*(t_f)$. Our framework is able to handle without any modifications all these definitions of the assigned task.

To summarize, the planning problem considered in this chapter is to find a feasible whole-body motion of the humanoid over $[t_i, t_f]$ that realizes the assigned task while avoiding collisions with workspace obstacles, whose geometry is known in advance. In our approach, a solution is identified by a concatenation of CoM movement primitives that has been 'fleshed out' by defining collision-free whole-body motions which realize such movements while complying with the task. In the end, however, the solution can be directly described in terms of joint motions.

Formally, a solution for our TCMP problem is a trajectory $\boldsymbol{q}_{\mathrm{jnt}}(t)$, $t \in [t_i, t_f]$ that satisfies three requirements

1. the assigned task trajectory is exponentially realized; in formula,

$$\lim_{t \to \infty} (\boldsymbol{y}(t) - \boldsymbol{y}^*(t)) = \boldsymbol{0}$$

   with an exponential rate of convergence;

2. self-collisions and collisions with workspace obstacles are avoided;

3. position and velocity of the joints are within their bounds, respectively in the form $\boldsymbol{q}_{\mathrm{jnt,m}} < \boldsymbol{q}_{\mathrm{jnt}} < \boldsymbol{q}_{\mathrm{jnt,M}}$ and $\boldsymbol{v}_{\mathrm{jnt,m}} < \boldsymbol{v}_{\mathrm{jnt}} < \boldsymbol{v}_{\mathrm{jnt,M}}$;

Note that if $\boldsymbol{y}(t_i) = \boldsymbol{y}^*(t_i)$ (*matched* initial configuration, or 'the robot starts on the task'), the first requirement automatically becomes

$$\boldsymbol{y}(t) = \boldsymbol{y}^*(t),\ \forall t \in [t_i, t_f],$$

i.e., the assigned task must be exactly realized at all times.

The first requirement is another small improvement w.r.t. the framework proposed in the previous chapter. In fact, the initial configuration must be on the task in the formulation of Section 4.1.2 while here we do not require it. This is due to the CoM movement primitives and their role in determining the time interval of the generic iteration $[t_k, t_{k+1}]$ (see Section 5.2.1 for

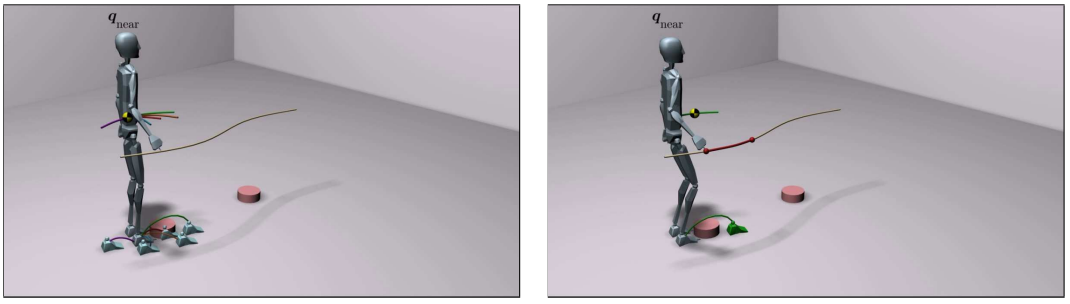


Figure 5.2 Example of the CoM movement selection procedure. (left) Once $\boldsymbol{q}_{\text{near}}$ has been selected, the CoM movement selector has to choose between the various primitive (depicted each with a different color). Each primitive embeds a duration and a trajectory for the CoM and possibly other points (here swing foot). (right) A primitive has been selected, uniquely identifying the portion of the task to be fulfilled (red).

further details), leading also to recover an initial error in a natural way. On the other side, this is not trivial to obtain by using the framework described in Chapter 4, since the time interval (for each iteration of the planner) is defined by the planner itself (the T_k in Section 4.3) and the sum of the durations might be not compatible with the total duration of the assigned task.

5.2 Motion generation

The proposed planner works in an iterative fashion by repeated calls to a *motion generator*. That is the reason for which we present first this module and then we describe the planner in Section 5.3.

As for the motion generation scheme presented in Section 4.2, we use two interleaved procedures that reflects the way the configuration vector $\boldsymbol{q}$ is composed in eq. (5.1). First, the *CoM movement selector* chooses a particular CoM movement from the set of primitives (that we assumed to be available to the planner). Then, the *joint motion generator* computes feasible collision-free joint motions that realize the chosen primitive as well as the corresponding portion of the assigned task trajectory. The definition of feasible motion is the same as given in Section 4.2, that we repeat here for completeness. A motion is feasible if the configurations that compose the motion are collision-free. Moreover, none of these configurations has to exceed the joint limits nor the joint velocity limits.

In Section 5.2.1 we describe the CoM movement selection mechanism while the joint motion generation is depicted in Section 5.2.2.

5.2.1 CoM movement selection

This procedure is the main change w.r.t. the framework described in the previous chapter. The CoM movement selector is invoked from the current configuration $\boldsymbol{q}^k = (\boldsymbol{q}^k_{\text{CoM}} \ \boldsymbol{q}^k_{\text{jnt}})^T$ at time t_k and its aim is to select a particular CoM movement in the primitive set. This is conceptually different with respect to the step generation procedure described in Section 4.2.1, where a pose displacement of the support foot was the quantity to be computed. Here, the CoM movement

selection is has to compute $\boldsymbol{u}^k_{\mathrm{CoM}}$, the pose displacement of the CoM reference frame at t relative to the pose at t_k.

For sake of illustration, we consider a precomputed catalogue of CoM movement primitives that contains only stepping movements (as well as a non-stepping motion, see below). However, it important to empathise that the proposed framework works with any set of primitives; indeed, the richer this set, the larger the set of tasks for which we will be able to plan a whole-body motion. For example, *crouching* and *crawling* primitives would allow to achieve tasks that require passing below obstacles. On the other side, the richer is the set and higher would be the planning time, since the planner has more primitive to choose and a wider space to explore.

As explained in Section 5.1, each primitive specifies the history of the relative pose displacement $\boldsymbol{u}^k_{\mathrm{CoM}}(t)$ for all $t \in [t_k, t_{k+1}]$, with $t_{k+1} = t_k + T_k$. Recall that each primitive comes with a duration T_k, that is embedded in its definition. In the following, we denote this history by $\boldsymbol{u}^k_{\mathrm{CoM}}$ for compactness, removing the temporal dependence. A CoM movement $\boldsymbol{u}^k_{\mathrm{CoM}}$ is then selected by picking one primitive from the catalogue

$$U = \left\{ U^{\mathrm{S}}_{\mathrm{CoM}} \cup U^{\mathrm{D}}_{\mathrm{CoM}} \cup \texttt{free\_CoM} \right\} \tag{5.4}$$

where $U^{\mathrm{S}}_{\mathrm{CoM}}$ and $U^{\mathrm{D}}_{\mathrm{CoM}}$ are subsets of *static* and *dynamic* steps, respectively, and `free_CoM` a *non-stepping* movement.

An example of the CoM movement selection procedure is given in Figure 5.2.

The primitives in $U^{\mathrm{S}}_{\mathrm{CoM}}$ are extracted from a *static* walking gait, where equilibrium is statically guaranteed by checking that the ground projection of the CoM lies in the support polygon of the humanoid robot. A primitive of this kind can be performed by recording the CoM (and the swing foot) trajectory from, e.g., one step described in Section 4.2.1. This set will typically include a forward step ($\boldsymbol{u}^{\mathrm{SF}}_{\mathrm{CoM}}$), a backward step ($\boldsymbol{u}^{\mathrm{SB}}_{\mathrm{CoM}}$), left ($\boldsymbol{u}^{\mathrm{SL}}_{\mathrm{CoM}}$) and right ($\boldsymbol{u}^{\mathrm{SR}}_{\mathrm{CoM}}$) steps, and possibly others.

The stepping motions in $U^{\mathrm{D}}_{\mathrm{CoM}}$ are extracted from a *dynamic* walking gait, where the equilibrium is dynamically guaranteed by checking that the Zero Moment Point (ZMP) is always contained in the humanoid support polygon. More details about the ZMP are given in [51, 52, 53, 54, 115, 122]. This set will typically include a starting step, a cruise step and a stopping step, for each direction of motion. As example, in the forward direction, the starting step will be denoted with $\boldsymbol{u}^{\mathrm{DF,start}}_{\mathrm{CoM}}$, the cruise step with $\boldsymbol{u}^{\mathrm{DF,cruise}}_{\mathrm{CoM}}$ and the stop step with $\boldsymbol{u}^{\mathrm{DF,stop}}_{\mathrm{CoM}}$. Similar notations will be used for left, right and backward steps[4]. See Figure 5.3 for examples of such primitives.

Finally, `free_CoM` is a primitive where the CoM is completely free to move as long as both feet remain fixed and the robot maintains equilibrium. This is a fundamental primitive within our framework. In fact, it is a *stretchable* primitive in the sense that its duration can be chosen

[4]The second letter in the superscript refers to the direction of motion. As example the "F" in $\boldsymbol{u}^{\mathrm{DF,start}}_{\mathrm{CoM}}$ refers to the forward direction of motion. Similarly, we use "L", "R", "B" for left, right and backward motions, respectively.

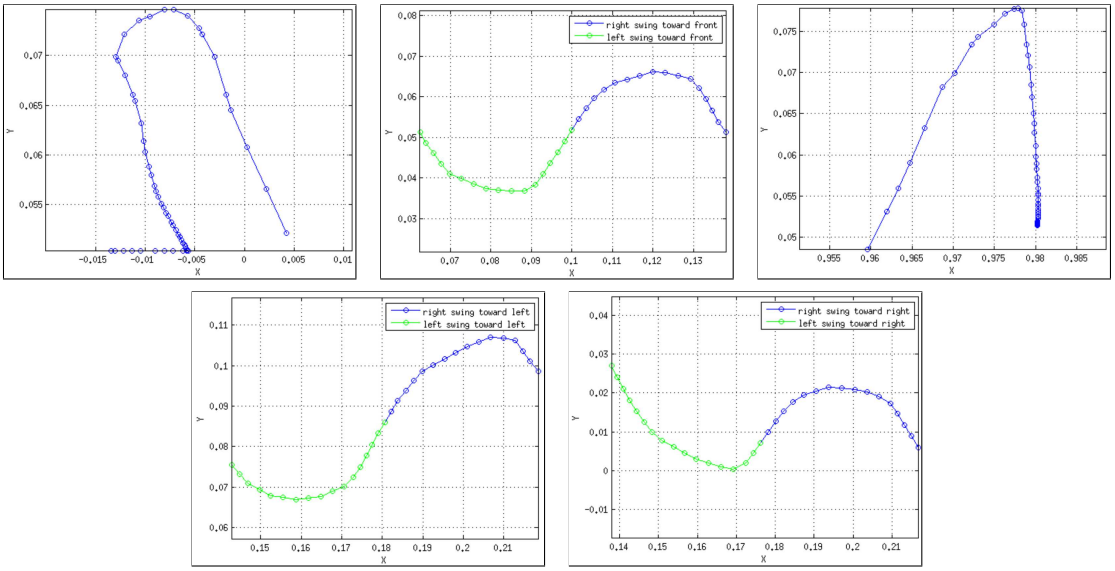


Figure 5.3 CoM movement primitives for the set U^D_{CoM}. In order: (top row) a starting forward step ($\boldsymbol{u}^{\text{DF,start}}_{\text{CoM}}$), a cruise forward step ($\boldsymbol{u}^{\text{DF,cruise}}_{\text{CoM}}$) and a stopping forward step ($\boldsymbol{u}^{\text{DF,stop}}_{\text{CoM}}$). (bottom row) a left cruise step ($\boldsymbol{u}^{\text{DL,cruise}}_{\text{CoM}}$) and a right cruise step ($\boldsymbol{u}^{\text{DR,cruise}}_{\text{CoM}}$). The green lines in the plot refer to right foot support phase; otherwise, the support foot is the left one (blue lines).

arbitrarily (as opposite to the other primitives). Then, it allows to build a sequence of movements whose total duration is $t_f - t_i$ (as specified by the task) using motion primitives whose individual durations are otherwise fixed. To convince the reader, consider the following example where U is composed by just one primitive with a duration of 2 s. Assume that the task has a duration of 5 s. Since there is just one primitive, this will be selected at each step. When arrived at $t = 4$ s, the motion planner cannot complete the task choosing a primitive (since it lasts 2 s). Introducing `free_CoM`, this primitive can be used to cope with the final portion of the task[5], adapting its duration to be 1 s. Moreover, note that

- as stated before, a CoM movement primitives specify the motion of other points on the robot body in addition to that of the CoM displacement. For example, stepping primitives assign also the swing foot trajectory within the associated time interval;
- $\boldsymbol{q}_{\text{CoM}}$ is completely defined in $[t_k, t_{k+1}]$ by plugging the initial CoM pose $\boldsymbol{q}^k_{\text{CoM}}$ and the selected primitive $\boldsymbol{u}^k_{\text{CoM}}$ in eq. (5.2);
- the above stepping movement primitives can be precomputed using suitable Walking Pattern Generators;
- at a given configuration $\boldsymbol{q}^k$, the set of primitives from which to choose is actually a subset of U that depends on the configuration itself, and in particular on which CoM primitive has produced $\boldsymbol{q}^k$. For example, only another cruise step or a stopping step are admissible;

[5]Note that the `free_CoM` can be used also in fulfilling also other portions of the task, not only to cope the final part.

no static step can be selected after a dynamic cruise step. Similarly, the only dynamic step that can follow a static step is a starting step; and so on.

The policy with which a primitive has to be selected in the primitive set U can be purely random (in order to explore with uniform probability the set) or based on appropriate heuristics, possibly designed for a specific task. Since we want to create a motion planner as general as possible, we choose the random solution. However, it is trivial to specify an heuristic within our framework.

To summarize, the CoM movement selector gives as output

1. a duration T_k for the movement and a time interval $[t_k, t_{k+1}]$, with $t_{k+1} = t_k + T_k$. It comes from the primitive itself that embeds a duration (except for `free_CoM`);
2. for all primitives but `free_CoM`, a reference trajectory $\boldsymbol{z}^*_{\mathrm{CoM}}$ for the position of the CoM (the position component of $\boldsymbol{q}_{\mathrm{CoM}}$) in $[t_k, t_{k+1}]$, where $t_{k+1} = t_k + T_k$;
3. for all primitives, a reference trajectory $\boldsymbol{z}^*_{\mathrm{swg}}$ in $[t_k, t_{k+1}]$ for the swing foot (position and orientation).

Obviously, if `free_CoM` has been selected as primitive, the swing foot reference trajectory is simply $\boldsymbol{z}^*_{\mathrm{swg}}(t) = \boldsymbol{z}^*_{\mathrm{swg}}(t_k), \forall t \in [t_k, t_{k+1}]$.

Although the $\boldsymbol{q}^*_{\mathrm{CoM}}$ is a pose vector, we decided to constrain only its position, $\boldsymbol{z}^*_{\mathrm{CoM}}$. In fact, some humanoid movements are more naturally described by the position rather than the pose of the CoM. We prefer to leave the planner free to decide on how to move the orientation of the CoM reference frame, using these additional degrees of freedom to cope with other constraints (e.g., the assigned task). This is of course possible due to the fact that most WPGs generate CoM trajectories just in position and we are focusing only on walking primitives. Furthermore, we implemented both versions: the one where the CoM pose[6] is constrained and the one where just its position is constrained. As just explained, we found out that the latter outperforms the former, giving more freedom to the motion planner to fulfil other planning requirements.

5.2.2 Joint motion generation

This section is mainly equal to Section 4.2.2. Here we summarizes the main concepts of that section, in order to create a self-contained chapter. The reader is then referred to Section 4.2.2 for further details.

Once a CoM movement primitive with duration T_k has been selected, joint velocities can be instantaneously generated. These velocities should enable the robot to realize the assigned task $\boldsymbol{y}^*$ in $[t_k, t_{k+1}]$, with $t_{k+1} = t_k + T_k$. Moreover, it has to fulfil the trajectory of the for the CoM $\boldsymbol{z}^*_{\mathrm{CoM}}$ and the swing foot $\boldsymbol{z}^*_{\mathrm{swg}}$, output of the CoM movement selector, within the same interval.

[6] We recall that, in this case, the orientation is the one of the torso, expressed in unit quaternion.

Let $\boldsymbol{y}_a = (\boldsymbol{y}^T \quad \boldsymbol{z}_{\text{swg}}^T \quad \boldsymbol{z}_{\text{CoM}}^T)^T$ be the augmented task vector, in case `free_CoM` has not been selected as primitive. In such a case, the augmented task vector becomes $\boldsymbol{y}_a = (\boldsymbol{y}^T \quad \boldsymbol{z}_{\text{swg}}^T)^T$, since the CoM is not constrained when using such a primitive. Define $\boldsymbol{J}_a$ as the Jacobian matrix of $\boldsymbol{y}_a$ w.r.t. the joint variables $\boldsymbol{q}_{\text{jnt}}$ and $\boldsymbol{e} = \boldsymbol{y}_a^* - \boldsymbol{y}_a$ as the augmented task error, with $\boldsymbol{y}_a^*(t)$ the reference value of the augmented task in $[t_k, t_{k+1}]$. Joint velocity commands are generated as

$$\boldsymbol{v}_{\text{jnt}} = \boldsymbol{J}_a^{\dagger}(\boldsymbol{q}_{\text{jnt}})\,(\dot{\boldsymbol{y}}_a^* + \boldsymbol{K}\boldsymbol{e}) + (\boldsymbol{I} - \boldsymbol{J}_a^{\dagger}(\boldsymbol{q}_{\text{jnt}})\boldsymbol{J}_a(\boldsymbol{q}_{\text{jnt}}))\boldsymbol{w}, \tag{5.5}$$

where $\boldsymbol{J}_a^{\dagger}$ is the pseudoinverse of $\boldsymbol{J}_a$, $\boldsymbol{K}$ is a positive definite gain matrix and $\boldsymbol{w}$ is an n-vector that may be chosen arbitrarily without perturbing the execution of the augmented task. Indeed, since $\boldsymbol{I} - \boldsymbol{J}_a^{\dagger}\boldsymbol{J}_a$ is the orthogonal projection matrix in the null space of $\boldsymbol{J}_a$. If one substitutes eq. (5.5) in eq. (5.3) and then computes the error dynamics, the result is $\dot{\boldsymbol{e}} = -\boldsymbol{K}\,\boldsymbol{e}$, i.e., exponential convergence of the augmented task to its reference trajectory.

Regarding $\boldsymbol{w}$, in order to explore the space of possible solutions, we select it as

$$\boldsymbol{w} = \boldsymbol{w}_{\text{rnd}}\ , \tag{5.6}$$

where $\boldsymbol{w}_{\text{rnd}}$ is a bounded-norm random n-vector. In case `free_CoM` has not been selected as primitive, we use a slightly different choice of $\boldsymbol{w}$:

$$\boldsymbol{w} = -\eta \cdot \nabla_{\boldsymbol{q}_{\text{jnt}}} H(\boldsymbol{q}_{\text{jnt}}) + \boldsymbol{w}_{\text{rnd}}, \quad \eta > 0, \tag{5.7}$$

where $H(\boldsymbol{q}_{\text{jnt}})$ is the squared distance between the centroid of the support polygon and the ground projection of the CoM. The first right term in eq. (5.7) tends to move the CoM in the direction of the center of the support polygon, in order to privilege the generation of robot configurations that are statically stable. The second right term in eq. (5.7) has the aim of avoiding deterministic actions. Note that, if $\boldsymbol{w}$ is deterministic, eq. (5.5) is deterministic as well. In other words, given $\boldsymbol{y}_a$ and $\dot{\boldsymbol{y}}_a$, eq. (5.5) produces every time the same output. If this causes a collision, this is never avoided. On the contrary, if one imposes different values for $\boldsymbol{w}$ (even randomly), different solutions are found, increasing the probability to find at least one of them that is not colliding. Recall that any value assumed by $\boldsymbol{w}$ does not interfere with the fulfilment of the tasks in $\boldsymbol{y}_a$ since $\boldsymbol{w}$ is projected in the null space of $\boldsymbol{J}_a$.

The trajectories generated by (5.5–5.6), or (5.5–5.7), are continuously checked for collisions as well as for position and velocity joint limits. In case `free_CoM` has been selected, static equilibrium is also explicitly checked, since the CoM is not explicitly constrained and it might leave the support polygon region, in principle. If any of these conditions is violated, the current execution of the motion generator is interrupted. On the contrary, a feasible collision-free joint motion $\boldsymbol{q}_{\text{jnt}}(t)$, $t \in [t_k, t_{k+1}]$ has been produced, ready for a possible usage from the motion planner.

Finally, we integrate eq. (5.5) using a numeric solver, whose integration step might be taken as small as possible in order to achieve an arbitrary accuracy of trajectory tracking. We tried

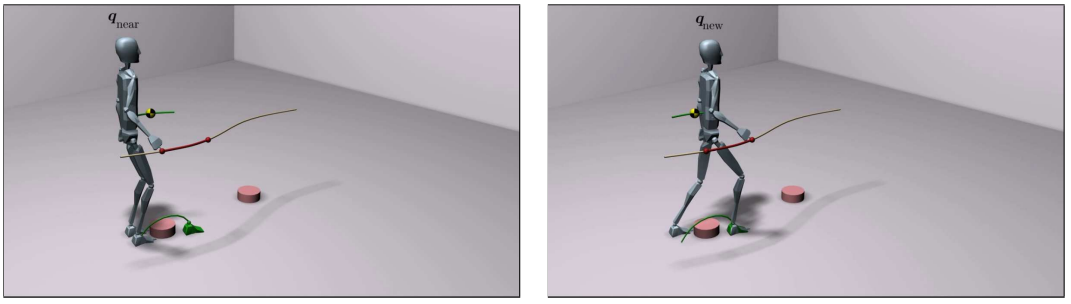


Figure 5.4 Example of the joint motion generation procedure. (left) Output of the CoM movement selector, i.e., a trajectory for the swing foot and for the CoM. (right) Joint motion can be generated such that the robot fulfils a portion of the assigned task, together with the above-mentioned trajectories.

different integrators, selecting the 4th order Runge-Kutta as best choice. One reason for which we integrate eq. (5.5) is that our robotic platform accepts only joint position inputs, rather than velocity commands. Moreover, joint limits should be checked and integration of eq. (5.5) is needed also for this purpose.

5.3 Planner overview

Our planner builds a tree $\mathcal{T}$ in configuration space with the root at the initial configuration $\boldsymbol{q}(t_i)$. Nodes are configurations of the humanoid associated to a time instant, while arcs represent feasible, collision-free whole-body motions that realize a portion of the task. As explained in the previous section, each of these motions has been computed using a CoM movement primitive as a 'seed'. The planner is somehow similar to the one presented in Section 4.3 with a few key modifications.

The motion planner makes usage of a task compatibility function, defined in the configuration space. This function $\gamma(\boldsymbol{q}, \bar{\boldsymbol{y}})$ measures the compatibility of a configuration $\boldsymbol{q}$ with a certain point $\bar{\boldsymbol{y}}$ of the assigned task trajectory $\boldsymbol{y}^*$. For example, for a manipulation task, $\gamma(\boldsymbol{q}, \bar{\boldsymbol{y}})$ can be defined as the inverse of the Euclidean distance between the ground projection of $\bar{\boldsymbol{y}}$ and the midpoint between the feet when the robot is in $\boldsymbol{q}$. The rationale here is that configurations where this distance is large are more prone to failure, when invoking the motion generation described in Section 5.2. This is due to the joint values, that result to be close to the limits of their available ranges. This kind of compatibility function is also appropriate for a navigation task. Since manipulation and navigation tasks are the kinds of tasks we want to face, this function is chosen in the planning experiments of Section 5.4. Tasks of a different nature (e.g., visual) would require the definition of appropriate compatibility functions.

Some snapshots of a generic iteration of the proposed planner are depicted in Figure 5.5. First, a random $3D$ point $\boldsymbol{y}^*_{\text{rand}}$ on the assigned task trajectory is sampled and its projection on the ground is computed, as in Figure 5.5a. Then, the most suitable configuration $\boldsymbol{q}_{\text{near}}$ is randomly extracted using a probability that is proportional to $\gamma(\boldsymbol{q}, \bar{\boldsymbol{y}})$. To this purpose, the compatibility

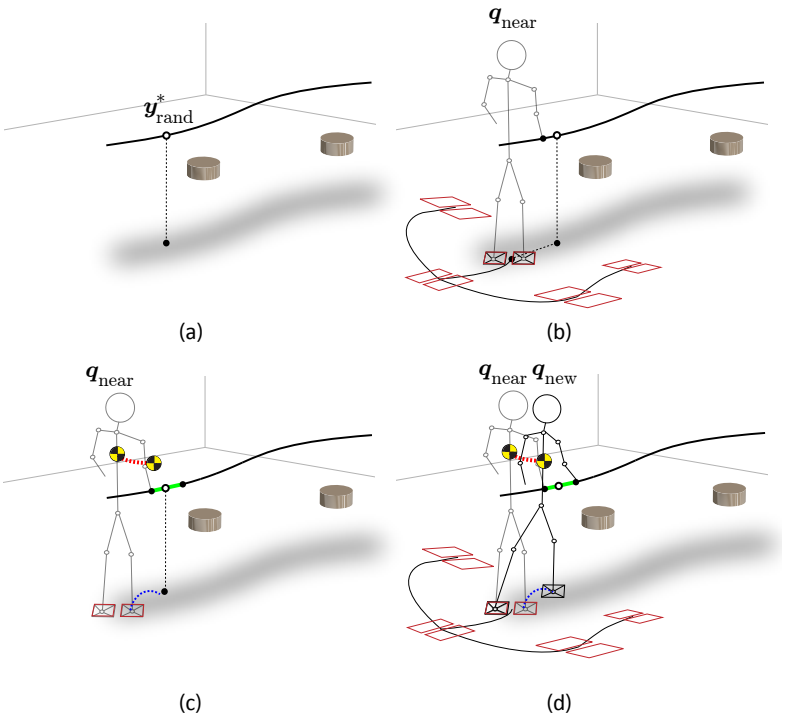


Figure 5.5 A generic iteration of the proposed planner. (a) A random sample $\boldsymbol{y}^*_{\text{rand}}$ is extracted from the assigned path and its ground projection is computed. (b) The task compatibility function $\gamma(\boldsymbol{q}, \bar{\boldsymbol{y}})$ is computed w.r.t. each configuration $\boldsymbol{q}$ in the tree $\mathcal{T}$. A configuration $\boldsymbol{q}_{\text{near}}$ is extracted randomly using a probability proportional to the values assumed by the compatibility functions. (c) The motion generation is invoked and it selects a particular CoM movement primitive that defines the time interval for the current iteration. It also defines a trajectory for the CoM and the swing foot (respectively dashed red and blue) as well as the portion of the assigned task to be executed (green). (d) The joint motion generator produces a whole-body trajectory that complies with the task. If this trajectory is feasible and collision-free, its final configuration $\boldsymbol{q}_{\text{new}}$ is added to the tree.

function $\gamma(\boldsymbol{q}, \bar{\boldsymbol{y}})$ is computed w.r.t. each configuration $\boldsymbol{q}$ in the tree $\mathcal{T}$. In details, the following probability is assigned to each configuration node $\boldsymbol{q}$ in the tree $\mathcal{T}$

$$p(\boldsymbol{q}^k) = \frac{\gamma(\boldsymbol{q}^k, \bar{\boldsymbol{y}})}{\sum_{i=1}^{N} \gamma(\boldsymbol{q}^i, \bar{\boldsymbol{y}})} ,$$

with $\boldsymbol{q}^k$ the k-th node in $\mathcal{T}$. The node $\boldsymbol{q}^k$ is extracted as the most suitable configuration $\boldsymbol{q}_{\text{near}}$ with probability $p(\boldsymbol{q}^k)$. Obviously, $\boldsymbol{q}_{\text{near}}$ corresponds to $\boldsymbol{q}(t_i)$ at the first iteration, since it is the only node in $\mathcal{T}$. The reason for introducing such a mechanism instead of selecting just the nearest configuration is the completeness. Assume to have two configurations in $\mathcal{T}$ associated with the same time instant t_k and assume that their values for the task compatibility function are different. If the nearest criteria is used, the configuration with the larger value of this function will never be selected (given a task point $\bar{\boldsymbol{y}}$) and that node will be never be expanded, compromising the completeness of the algorithm. This process is depicted in Figure 5.5b.

Once $\boldsymbol{q}_{\text{near}}$ has been identified with its associated time instant t_k, the motion generation is invoked. First, the CoM movement selector chooses a primitive in the available set, as described

Algorithm 1: Planner

root the tree $\mathcal{T}$ at $\boldsymbol{q}(t_i)$;
repeat
  $i \leftarrow i + 1$;
  select a random sample $\boldsymbol{y}^*_{\text{rand}}$ on the task trajectory;
  select a random node $\boldsymbol{q}_{\text{near}}$ from $\mathcal{T}$ with probability proportional to $\gamma(\cdot, \boldsymbol{y}^*_{\text{rand}})$;
  get the time instant t_k associated with $\boldsymbol{q}_{\text{near}}$;
  $[\boldsymbol{q}_{\text{new}}, \overline{\boldsymbol{q}_{\text{near}}\boldsymbol{q}_{\text{new}}}, t_{k+1}] \leftarrow$ GenerateMotion($\boldsymbol{q}_{\text{near}}, t_k$);
  if $\boldsymbol{q}_{\text{new}} \neq \emptyset$ **then**
    add node $\boldsymbol{q}_{\text{new}}$ and arc $\overline{\boldsymbol{q}_{\text{near}}\boldsymbol{q}_{\text{new}}}$ to $\mathcal{T}$;
  end
until $t_{k+1} = t_f$ **or** $i =$ MAX_IT ;

Procedure GenerateMotion$(\boldsymbol{q}_{\text{near}}, t_k)$

select a random CoM primitive $\boldsymbol{u}^k_{\text{CoM}}$ from the currently available subset of U given by eq. (5.4);
get the associated duration T_k, CoM trajectory $\boldsymbol{z}^*_{\text{CoM}}$ and swing foot trajectory $\boldsymbol{z}^*_{\text{swg}}$;
extract the portion of task trajectory $\boldsymbol{y}^*$ in $[t_k, t_k + T_k]$;
build the extended task $\boldsymbol{y}_a = (\boldsymbol{y}^T \;\; \boldsymbol{z}^T_{\text{swg}} \;\; \boldsymbol{z}^T_{\text{CoM}})^T$;
repeat
  generate motion by integrating joint velocities (5.5);
  if collision **or** joint position/velocity limit violation **then**
    return $[\emptyset, \emptyset, \emptyset]$
  end
until $t = t_k + T_k$;
return $[\boldsymbol{q}_{\text{new}}, \overline{\boldsymbol{q}_{\text{near}}\boldsymbol{q}_{\text{new}}}, t_k + T_k]$

Figure 5.6 Pseudocode of the proposed planner

in Section 5.2.1. As explained in the same section, this subset depends on $\boldsymbol{q}_{\text{near}}$. The chosen primitive comes with a duration T_k that defines the time interval in this iteration $[t_k, t_{k+1}]$, with $t_{k+1} = t_k + T_k$. Moreover, the CoM movement selector provides a trajectory for the CoM ($\boldsymbol{z}^*_{\text{CoM}}$) and for the swing foot ($\boldsymbol{z}^*_{\text{swg}}$). The portion of the task to be fulfilled is uniquely identified since the time interval $[t_k, t_{k+1}]$ has been already selected. This is depicted in Figure 5.5c.

As last step, the joint motion generator, described in Section 5.2.2, is invoked in order to produce a whole-body trajectory that complies with the portion of the assigned task and, at the same time, fulfils the trajectories of the swing foot and of the CoM (if `free_CoM` has not been selected as CoM movement primitive), output of the CoM movement selector. If the trajectory is collision-free and feasible, its final configuration $\boldsymbol{q}_{\text{new}}$ is added to the tree; otherwise, a new iteration is started. When $t_{k+1} = t_f$, the planner ends and a solution is found by backtracking in the tree $\mathcal{T}$.

In order to perform a comparison with the motion planner described in Section 4.3, note the following important points

- the duration of the current iteration T_k is chosen by the planner in Section 4.3 (recall that we also assumed a constant duration T for simplicity). Then, the duration comes from the planner that decides which portion of the assigned task has to be fulfilled. In the new planner, T_k comes from the CoM movement primitive and, then, the duration comes from the primitive;

- tasks having a starting error or assigned as simple destinations in the task space are not taken into account in the motion planner described in Section 4.3, while the novel version complies naturally with them (see Section 5.4);

- the framework in Section 4.3 includes just static steps, while the one presented in this chapter handles both static and dynamic steps, as described in Section 5.2.1. The extension of the former for including dynamic steps is not so trivial as it may appear;

- composite tasks (i.e., tasks obtained by concatenating different tasks, even of different nature) are not handled in Section 4.3, while they are discussed for the novel motion planner (see Section 5.4).

The pseudocode of the proposed planner is provided in Figure 5.6.

Since it will be handled in Section 5.4, let explicitly analyse the case where the assigned task is a simple destination in the task space, i.e., $\boldsymbol{y}^*(t) = \boldsymbol{y}^*(t_f), \forall t \in [t_i, t_f]$. The main difference is that the random point on the task $\boldsymbol{y}^*_{\text{rand}}$ is always equal to $\boldsymbol{y}^*(t_f)$. Moreover, the heuristic used for the selection of $\boldsymbol{q}_{\text{near}}$ favours configurations that are closer to the task point during the expansion of the tree. Moreover, the convergence on that point is guaranteed from eq. (5.5), by setting $\dot{\boldsymbol{y}}^* = \boldsymbol{0}$. The motion generator will produce a linear motion in the task space with exponential rate of convergence.

5.4 Planning experiments

The proposed planner has been implemented in V-REP for NAO, a small humanoid by Aldebaran Robotics, and runs on an Intel Core 2 Quad at 2.66 GHz. A description of the NAO robot is given in Section 4.4.1 while an overview of the simulation platform is given in Section 4.4.2.

The set of CoM movement primitives is defined as in eq. (5.4). The set of static steps $U^{\text{S}}_{\text{CoM}}$ has been precomputed using a Static Walking Pattern Generator (or, as example, the method described at the end of Section 4.2.1). In details, we included different step lengths in the range $[0.03, 0.12]$ m for forward/backward steps and $[0.01, 0.03]$ m for lateral steps. We also provided static steps with different steps (in the range $[0.02, 0.06]$ m), in order to let the robot to step over low obstacles. All static steps have a duration of around 2 s. The stepping motions in $U^{\text{D}}_{\text{CoM}}$ have been precomputed by a ZMP-based Walking Pattern Generator. In details, it includes a starting step of length 0.038 m and duration 1.6 s, a cruise step of length 0.04 m and duration 0.425 s, and a stopping step of length 0.038 m and duration 1.325 s, all in the forward direction.

In all dynamic steps, the maximum height for the swing foot is 0.02 m. These trajectories for the CoM are depicted in Figure 5.3. The total number of CoM movement primitives in U is 16. As mentioned in Section 5.2.1, we decided to use a uniform probability for selecting a primitive in U. Again, recall that only a subset of U is available at each step of the motion planning phase, depending on the state of $\boldsymbol{q}_{\text{near}}$. As example, if $\boldsymbol{q}_{\text{near}}$ was the result of starting a dynamic motion, the planner may only choose to stop the dynamic gait or to perform a dynamic cruise; no static step is allowed.

Regarding the joint motion generation, we set $\boldsymbol{K} = \text{diag}\{2, 2, 1\}$ and $\eta = 1.6$ in eqs. (5.5-5.6-5.7). Moreover, $\boldsymbol{w}_{\text{rnd}}$ is chosen randomly with a norm in the range $[0, 0.4]$ rad/sec. Numerical integration of joint velocities is performed with a 4th order Runge-Kutta algorithm with a step size of 0.025 s. As explained in Section 4.4, we preferred to use this integrator instead of a simple one (e.g., Euler) for its accuracy in fulfilling the assigned task even if it is slower.

We present two planning experiments. In the first, the robot should pick an object (a ball) placed on a low stool. Then, the robot has to reach a point across a corridor. This scenario is discussed in Section 5.4.1. In the second planning scenario, the robot should reach a point in a room. In doing it, it has to go through an automatic door whose guide rail represents a ground obstacle. This is discussed in Section 5.4.2. To further validate our results, we have also performed (for each proposed experiment) a dynamic playback of the planned motion in which full physical simulation (including joint control) is enabled: this means that the joint motions in the computed plan are feasible and can be effectively tracked by the NAO low-level joint controllers. In this playback, a physics engine runs in the background, enabling to simulate multibody dynamics and interaction with the environment (foot contact, object grasping and releasing). In few words, we tried to perform a simulation as close as possible to the real robot case[7].

5.4.1 Grasping and walking

As briefly discussed above, the robot in this scenario must pick an object (a ball) that is placed on a low stool, outside the robot workspace. Once the object has been grasped, the robot should reach a point in the workspace. This can be translated as a composite task, consisting in two goals that should be reached in sequence. First, a manipulation task is assigned for the right hand of the humanoid robot in order to grasp the ball. Its definition is simply a desired position for the right hand, then it is a regulation task. The second goal is to place the midpoint between the feet to a desired position. Then, it is a navigation task. Its definition is again a regulation task. The navigation task is automatically activated when the manipulation task is completed. In view of the nature of our composite task, the task compatibility function γ is defined as described in Section 5.3.

[7]For further details, please visit `http://www.dis.uniroma1.it/~labrob/research/HumPrtvPlan.html`, where there is also a video of the proposed approach.

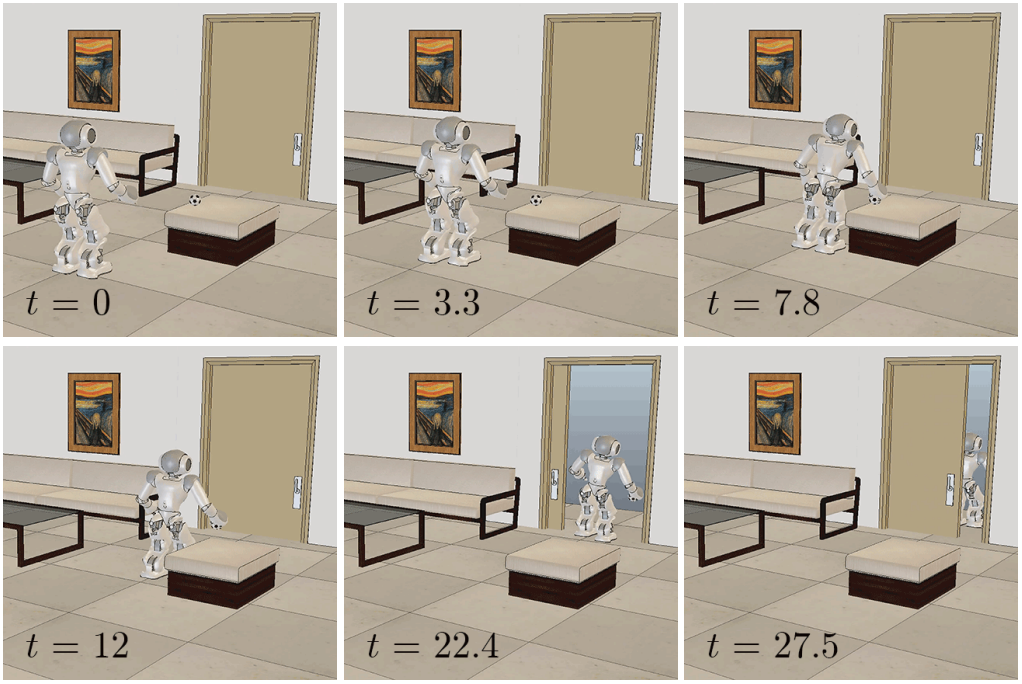


Figure 5.7 Grasping and walking: snapshots from a solution.

It is important to underline how simple would be for a user to specify this task. In fact, he/she just has to specify the two positions for the two specific points (in this case the right hand and the midpoint between the feet) and the motion planner will automatically come up with the joint motions that fulfil the assigned task.

In order to avoid unnatural motions, we decided not to constrain the hand during the early stages of the manipulation phase. This is performed by deleting the task component $\boldsymbol{y}$ from the augmented task vector $\boldsymbol{y}_a$ in eq. (5.5). When the right hand is within a certain ball around the desired position (in this scenario, the ball is centred into the object to be grasped), $\boldsymbol{y}$ is pushed back in $\boldsymbol{y}_a$ in eq. (5.5). It is important to underline that, even removing $\boldsymbol{y}$ from $\boldsymbol{y}_a$, the task plays a fundamental role in the expansion of the tree. In fact, expansion of the tree towards the manipulation goal point is still guaranteed in view of the metric γ used by our planner to select $\boldsymbol{q}_{\text{near}}$. In particular, the procedure for computing $\boldsymbol{q}_{\text{near}}$ remains the same also for a regulation task, where the only difference is that $\boldsymbol{y}^*_{\text{rand}} = \boldsymbol{y}^*(t_f)$ since it is the only point of the assigned task. Once the hand enters the ball, the task is activated; as a consequence, the robot performs a natural reaching motion only when is sufficiently close to the object. We motivate the activation of the hand task just in the vicinity of the manipulation goal by saying that if the hand was always constrained, the robot arm will stretch[8] resulting in an unnatural approaching behaviour.

Some snapshots from a solution are depicted in Figure 5.7. At the beginning ($t = 0$ s), the task is simply composed by the goal position (of the ball to be grasped) for the right hand of the humanoid. Note how the planner is able to create an approaching phase composed by

[8]Recall that, in the case of regulation task, like the grasping point, the convergence is exponential in time and linear in the task space.

data	grasp and walk	step over obstacle
planning time (s)	6.4	8.3
tree size (# nodes)	144.2	72.1
motion duration (s)	27.5	18.0

Table 5.1 Planner performance at a glance.

11 dynamic steps. These steps are needed since the ball is out of the robot workspace. Then, $t = 7.8$ s, it decides to switch to the `free_CoM` primitive in order to grasp the object in double support. Once the grasping has been completed, the navigation task is activated and a dynamic walking gait is restored (at $t = 12$ s) in order to cross the automatic door and reach its final destination at $t = 27.5$ s. It is important to underline that the selection of which primitive to use at each stage is totally left to the planner. In other words, this sensible solution was automatically produced by our planner by taking advantage of the catalogue of movements represented by the set of primitives. For the sake of clarity, it is worth noting that the grasping action is purely demonstrative.

Note that, using just a naive task description (two points), our framework is able to cope with composite tasks of different nature and complexity. Moreover, the planner is able to automatically switch between the primitives depending on the task it has to accomplish.

Table 5.1 collects some data (averaged over 20 runs since the proposed planner is probabilistic) related to the planner performance in both experiments. It reports the time needed by the planner to generate a solution (planning time), the dimension of the final tree (tree size) and the time needed for executing the motion (motion duration).

5.4.2 Stepping over an obstacle

In the second scenario, the assigned task is composed by a navigation task. In particular, a goal position is assigned for the midpoint of the feet of the humanoid robot. In order to reach it, the robot has to go through an automatic door, whose guide rail represents a ground obstacle.

Figure 5.8 depicts some snapshots taken from a solution obtained with our planner. As shown in these snapshots, the robot approaches the automatic door with a dynamic gait, stopping it at $t = 6$ s. Once there, it takes two static steps, having appropriate heights, in order to overcome the guide rail that acts as an obstacle (between $t = 6$ s and $t = 11.5$ s). Finally, it resumes a dynamic gait in order to complete the task at $t = 18$ s. The planner is using exactly the same set of primitives of the first scenario: here, the switch from a dynamic to a static gait is triggered by the characteristics of the environment. In fact, the robot cannot complete the task by using a dynamic gait since the height of the swing foot is not sufficient to overcome the obstacle (the guide rail). Note that the switching between primitives in the first scenario was a consequence of the composite nature of the task (the robot stopped to pick up the ball). On the other side, the switching here is caused by the environment, proving the versatility of the

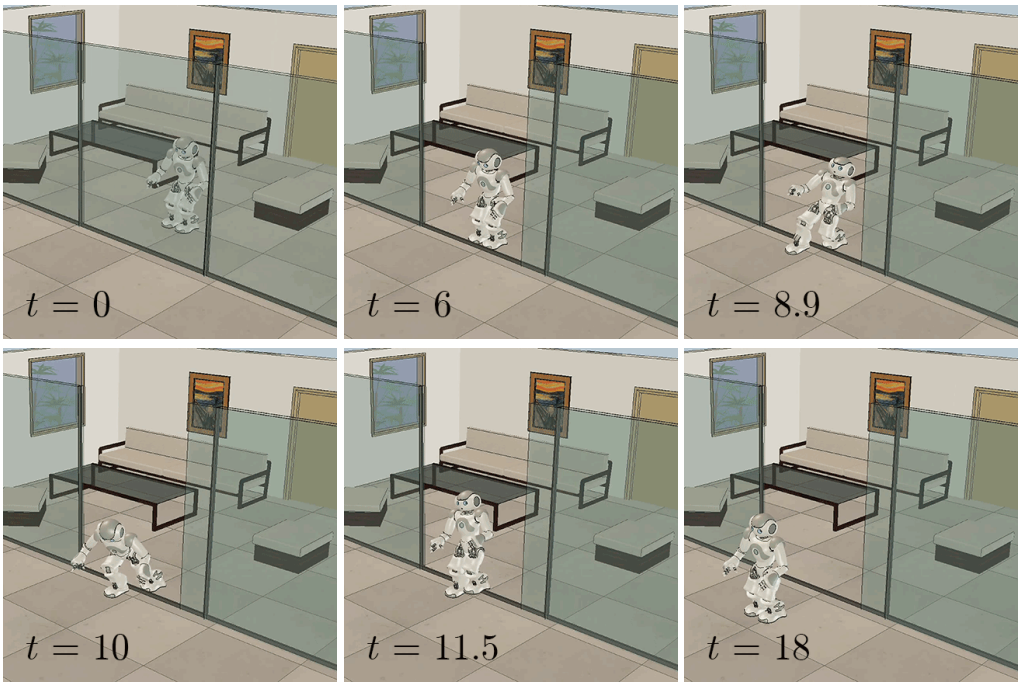


Figure 5.8 Stepping over an obstacle: snapshots from a solution.

proposed planner. Once again, the planner takes full advantage of the richness of the set of CoM movement primitives.

Table 5.1 reports some planning data (planning time, tree size and motion duration) for the proposed scenario (the data refers to the average of 20 runs, since the proposed motion planner is probabilistic).

Solutions for the first experiment have a longer duration and this is reflected in a larger exploration tree. On the other hand, the time needed to find a solution is longer for the second scenario. This is due to the difficulties that the motion planner encounters when overcoming the door guide rail. In fact, many configurations in that area were rejected due to collisions (especially between the feet and the ground obstacle).

5.5 Conclusions

In this chapter, we have presented an approach for planning motions of a humanoid robot that has to fulfil an assigned task in an environment cluttered by obstacles. The motion planner is based on the concept of CoM movement primitive, defined as precomputed trajectories of the CoM (and possibly other points of the robot, e.g., the swing foot in case of stepping primitives). It builds a tree in the configuration space by concatenating feasible, collision-free whole-body motions that realize the selected CoM movement primitives and, at the same time, the assigned task. We have tested the planner in different scenarios, proving that it takes full advantage from the CoM primitive set. In particular, it is able to automatically switch between the primitives when required from the task (e.g., the switching between the dynamic gait and the `free_CoM` primitive, aimed

to grasp the ball in Section 5.4.1) or the environment (e.g., the switching between a dynamic to a static gait in order to overcome the guide rail in Section 5.4.2).

The planner generalizes and outperforms the one presented in the previous chapter since it can indifferently handle tasks specified as trajectories or as simple points, as for the composite task considered in Section 5.4. Moreover, it naturally includes dynamic gait, while they are not considered in the previous formulation. Finally, composite tasks (i.e., tasks obtained as the concatenation of subtasks, even of different nature) are easily handled within the framework presented in this chapter. As example, we have showed that manipulation of a target object and a navigation task can be performed using just two points as definition of the assigned task (in Section 5.4.1).

Although we have essentially focused on walking primitives, we emphasize that the proposed framework is general and can only benefit from a richer set of primitives, such as jumping, crouching, and so on. Moreover, task-based heuristics for guiding the choice of the primitive can be formulated in order to further reduce the planning time.

The planner presented in this chapter has been published in [19][9].

[9]For further details, please visit `http://www.dis.uniroma1.it/~labrob/research/HumPrtvPlan.html`, where there is also a video of the proposed approach.

Chapter 6

Task-oriented whole-body motion planning for humanoids along deformable paths

The problem we want to face in this chapter is exactly the same as the one presented in Chapter 4 and Chapter 5, i.e., finding a collision-free motion for a humanoid robot whose assigned a task that it has to fulfil. However, the main limitation of that approaches relies in the definition of the task, supposed to be assigned a priori and unmodifiable. This means that, in the case in which the task is hard to be fulfilled by the robot (e.g., it passes very close to an obstacle), the motion planner has huge difficulties in finding a solution to the TCMP problem (taking a huge planning time). What a user expects from an ideal motion planner is that it is automatically able to deform the task, if needed.

Moreover, we motivate the task deformation in two points. First, it can significantly simplify the planning problem, letting the motion planner to easily fulfil the (deformed) task. Second, a user typically does not want to explicitly assign a trajectory for a task, but he/she wants to specify high-level actions (such as "move an object from here to there" or "take the ball and bring it into the other room", as in the case described in Section 5.4.1). In the example of moving an object from one location to another, what a user wants to specify is just the starting and the goal position of the object to move, without giving the entire trajectory that it has to follow. The goal of this chapter is to build a framework that deals with this. The overall idea is that a user just assigns a rough initial task (e.g., a simple straight line joining the start and end pose of the object). The motion planner tries first to fulfil this trajectory. In case of difficulties, it automatically deforms the task in such a way a new call to the motion planner reveals in an easier motion planning problem to be solved. It is important to underline that, even in the case in which an explicit task trajectory is not assigned, a TCMP problem has to be solved anyway, since the task is implicitly defined as a straight line with an exponential time history, as discussed in Section 5.4. Then, the framework will be presented for the trajectory case, without loss of generality.

To summarize, we do want to propose another motion planner in this chapter. On the contrary, we want to show how the framework described in Chapter 5 can be modified in order to allow the

automatic deformation of the task, if needed. This new framework should include a deformation detection, i.e., a module that detects when a deformation is needed and a deformation mechanism that illustrates how the path should be deformed.

In the literature, there are different techniques for deforming a task path. The work in [37] modifies it by minimizing its length and maximizing its clearance. It is a post-processing approach, i.e., it works on a solution already found by a motion planner. For the path length the authors suggest three approaches: path pruning, shortcuts and partial shortcut. Without entering into details, these techniques are known in the literature as shortcutting algorithms [36, 99]. The idea is to sample two random configurations on the path, create an "alternative" path (it is done by means of a local path planner. In the early papers, it was a simple straight line planner), check if it is collision free and replace the correspondent segment of the original path with the new path. Regarding the clearance, two algorithms are described: W-RETRACTION and C-RETRACTION. The first one is designed for rigid translational robots, so it is useless within our planning problem. The second one consists in picking a random direction and, for each node, move the joints along this direction. In other words, it tries to move the entire path along a direction in the configuration space, with a given step size, checking if it improves the clearance. Doing that for some iterations, the clearance of the path is actually improved. Again, this approach seems interesting but it lacks of computational time reliability, due to the collision checker calls. The two metrics (path length and clearance) are then fused in order to produce high-quality paths. The main concern is about the time needed by this algorithm (for some experiments, it is 10 times bigger to the initial motion planning time). On the other hand, our idea is to deform the path without performing any form of post-processing action. Moreover, our deformation mechanism should be as fast as possible.

The work in [121] maintains multiple trajectories at the same time and deforms them using a genetic algorithm, in real-time. Optimization is used in [127], where the authors uses functional gradient techniques to iteratively improve the quality of an initial trajectory. Finally, the works in [7, 124] try to combine the motion planning with the reactive behavior, enabling the task of a mobile manipulator to be deformed in real-time.

Up to the author knowledge, there are very few works on deformable paths applied to humanoid robots. In fact, most of the papers mentioned above are aimed for manipulators or mobile robots. A rare exception is in [40], where a shortcutting algorithm is presented, aimed to smooth jerky trajectories for many-DOFs robot manipulators subject to collision constraints, velocity bounds, and acceleration bounds. The heuristic repeatedly picks two points on the trajectory and attempts to replace the intermediate trajectory with a shorter, collision-free segment. Here, the metrics used is the time (to be minimized). The algorithm consists in first converting the existing path (it is assumed to be a piecewise-constant acceleration curve) into a trajectory that stops at every milestone (through a bang-coast-bang trajectory). Then, repeatedly, the starting and ending time for the current iteration are randomly sampled and the current position and velocity are extracted from the current path. The time needed in this iteration is first computed as the maximum

time needed by each joint (supposed independent each other) to satisfy the starting and ending position and velocity. In other words, first the time needed by the slowest joint is computed. To this purpose, four primitives are considered and evaluated. The primitive that minimizes the time is chosen. Once this time is known, a minimum-acceleration interpolant is computed for each joint, again choosing between four primitives, picking the one that needs less acceleration. Finally the trajectory is checked by a collision checking algorithm and, if it is collision-free, it replaces the portion of the original trajectory. However, the approach proposed here is different: we want an approach that is able to deform the path in such a way the task-constrained motion planning problem would be easier to solve.

This chapter is organized as follows. In Section 6.1, we briefly recall the humanoid motion model and formulate our planning problem that slightly modifies the one presented in Section 4.1.1. An overview of the framework is provided in Section 6.2. In Section 6.3, we perform some minor modifications on the motion planner presented in Section 5.3, while the deformation mechanism is presented in Section 6.4. Motion planning experiments for the NAO humanoid robot are presented in Sections 6.5 and 6.6. The overall chapter is finally discussed in Section 6.7.

6.1 Problem formulation

As usual, we first introduce a humanoid motion model.

6.1.1 Humanoid motion model

The humanoid motion model is exactly the same as in Section 5.1.1. The reader is then referred to Section 5.1.1 for further details. Here we just repeat a small summary, in order to create a self-contained chapter.

A configuration $\boldsymbol{q}$ of a free-flying humanoid with n joints can be identified by specifying the joint angles $\boldsymbol{q}_{\text{jnt}} \in \mathcal{C}_{\text{jnt}}$ and the pose (position plus orientation) $\boldsymbol{q}_{\text{CoM}} \in SE(3)$ of a reference frame attached to the robot CoM

$$\boldsymbol{q} = \begin{pmatrix} \boldsymbol{q}_{\text{CoM}} \\ \boldsymbol{q}_{\text{jnt}} \end{pmatrix}.$$

The configuration space $SE(3) \times \mathcal{C}_{\text{jnt}}$ has then dimension $n + 6$.

The above mechanism for motion generation approach can be compactly represented by the following hybrid (partly algebraic, partly continuous-time) model

$$\boldsymbol{q}_{\text{CoM}}(t) = \boldsymbol{q}^k_{\text{CoM}} \oplus \boldsymbol{A}(\boldsymbol{q}^k_{\text{CoM}})\boldsymbol{u}^k_{\text{CoM}}(t) \tag{6.1}$$

$$\dot{\boldsymbol{q}}_{\text{jnt}}(t) = \boldsymbol{v}_{\text{jnt}}(t) \tag{6.2}$$

that describes the robot evolution over the interval $[t_k, t_{k+1}]$ in which the CoM is performing a certain primitive movement of duration $T_k = t_{k+1} - t_k$. In the previous equations, $\boldsymbol{q}^k_{\text{CoM}} = \boldsymbol{q}_{\text{CoM}}(t_k)$ is the pose of the CoM at time t_k in world frame, $\boldsymbol{u}^k_{\text{CoM}}(t)$ is the pose displacement

that is the output of the CoM selected primitive, $\boldsymbol{v}_{\text{jnt}}$ is the velocity vector for the humanoid joints and $\oplus$ is a operator whose aim is to sum two poses. Finally, $\boldsymbol{A}(\boldsymbol{q}^k_{\text{CoM}})$ is a transformation matrix from the CoM frame at t_k to the world frame. Please refer to Section 5.1.1 for additional details about the humanoid motion module.

6.1.2 Task-contrained planning

Since a motion model is available, we can turn our attention to the task. Its formulation is slightly different w.r.t. the one presented in Section 5.1.2.

Assume that a task is defined as a trajectory (path plus time history) for the position (and possibly the orientation) of a specific point (body) of the humanoid. As example, a manipulation task may be assigned as for one hand of the humanoid robot. For notation purposes, we will use for brevity $\boldsymbol{y}^{[i]} = \boldsymbol{y}^{[i]}(s)$ and $s^{[i]} = s^{[i]}(t)$ for indicating a generic task path and time history, respectively. We indicate the task trajectory with $\boldsymbol{y}^{[i]}(s^{[0]}(t))$.

Denoting by $\boldsymbol{y}$ the task coordinates vector, this is related to generalized coordinates via the forward kinematic map

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{q}) = \boldsymbol{f}(\boldsymbol{q}_{\text{CoM}}, \boldsymbol{q}_{\text{jnt}}).$$

Assume that an initial reference task trajectory $\boldsymbol{y}^{[0]}(t)$, $t \in [t_i, t_f]$, is assigned[1] as a geometric path $\boldsymbol{y}^{[0]}(s), s \in [s_i, s_f]$, plus a time history $s^{[0]}(t), t \in [t_i, t_f]$, with $s_i = s(t_i)$, $s_f = s(t_f)$. Furthermore, the path $\boldsymbol{y}^{[0]}$ is assumed to be a *deformable* curve in task space. This means that the path endpoints ($\boldsymbol{y}^{[0]}(s_i)$ and $\boldsymbol{y}^{[0]}(s_f)$) are fixed, but the actual shape of the curve may be changed if necessary by acting on certain parameters $\boldsymbol{\sigma}$. As example, one might use – as we will do in the rest of this thesis – the *B-splines* as deformable curves, and the action parameters will be *control points*. Note that other choices are possible. Elastic strips [7] are another example of deformable curves that can be used to this scope. The reason for which we are making such assumption is that our planner will be able to change the shape of the task path (and consequently, the time history), if this is deemed necessary to solve the considered problem.

We emphasize that our framework naturally extends to the case in which the task is a desired set-point $\boldsymbol{y}(t_f)$ of a specific point of the robot (e.g., move the hand in a desired position) in task space. In this context, the reference task path $\boldsymbol{y}^{[0]}$ may be trivially chosen as a straight line (or any other curve) joining the starting position (computed as $\boldsymbol{y}(t_i) = \boldsymbol{f}(\boldsymbol{q}(t_i))$) and the set-point task position $\boldsymbol{y}(t_f)$. The time history may be simply chosen as $s^{[0]}(t) = t, \forall t \in [t_i, t_f]$.

We have then all the ingredients for formally define our planning problem and its solutions. The considered planning problem consists in finding a feasible whole-body motion for a humanoid robot over $[t_i, t_f]$ that realizes the assigned task trajectory, possibly deformed as explained below, while avoiding collisions with workspace obstacles, whose geometry is known in advance. Then, a solution to our problem consists of

[1] Since we are addressing a planning problem, it will be assumed that task value at the initial humanoid configuration matches the starting point of the reference trajectory, i.e., $\boldsymbol{y}^{[0]}(t_i) = \boldsymbol{f}(\boldsymbol{q}_{\text{CoM}}(t_i), \boldsymbol{q}_{\text{jnt}}(t_i))$.

1. a final reference task trajectory $\boldsymbol{y}^*(t), t \in [t_i, t_f]$, composed by a geometric path $\boldsymbol{y}^*(s), s \in [s_i, s_f]$, obtained by (repeated) deformation of $\boldsymbol{y}^{[0]}(s)$, and an appropriate time history $s^*(t), t \in [t_i, t_f]$;

2. a whole-body motion of the humanoid over $[t_i, t_f]$ that satisfies the following requirements:

   - the final reference task trajectory is realized; that is, for $t \in [t_i, t_f]$ it is

   $$\boldsymbol{y}(t) = \boldsymbol{f}(\boldsymbol{q}(t)) = \boldsymbol{y}^*(s^*(t)) = \boldsymbol{y}^*(t) \ ;$$

   - self-collisions and collisions with workspace obstacles are avoided;
   - joints and joint velocities are within their bounds, i.e., $\boldsymbol{q}_{\mathrm{jnt,m}} < \boldsymbol{q}_{\mathrm{jnt}} < \boldsymbol{q}_{\mathrm{jnt,M}}$ and $\boldsymbol{v}_{\mathrm{jnt,m}} < \boldsymbol{v}_{\mathrm{jnt}} < \boldsymbol{v}_{\mathrm{jnt,M}}$, respectively.

In the end, however, the solution may be described purely in terms of joint trajectories that achieve the planned whole-body motion.

Note that the previous formulation handles the following specific cases

- *Set-point task*: as mentioned before, when the task reduces to a desired set-point $\boldsymbol{y}(t_f)$ (e.g., bring the humanoid hand to a certain placement), the initial task path $\boldsymbol{y}^{[0]}(s)$ may be chosen as any deformable curve (e.g., a line) joining the starting task position $\boldsymbol{y}(t_i) = \boldsymbol{f}(\boldsymbol{q}(t_i))$ with the final task position $\boldsymbol{y}(t_f)$. The time history may be simply set as $s^{[0]}(t) = t, \forall t \in [t_i, t_f]$;

- *Partially deformable task*: in some problems, it may be desiderable to allow deformation only on a subset of task components. For example, consider a manipulation problem in which the robot has to carry a glass containing some liquid. The glass should be kept vertical, so as not to spill the liquid. A deformation can therefore be applied only to the position components of the task (i.e., to the Cartesian trajectory of the glass), whereas the orientation components should not be affected.

Just for comparison, the major different w.r.t. the task definition in Section 5.1.2 relies in the explicit split of the task in path and time history. In fact, we recall that the task was directly formulated as a trajectory in Section 5.1.2. The formulation we use in this chapter is useful in view of the deformation mechanism (Section 6.4) that provides different procedures for deforming the path and the time history of a task.

6.2 Planner overview

The overall idea of the motion planner is rather simple and summarized in Figure 6.1. It works in an iterative way. First, it attempts to find a solution for the initial reference task trajectory $\boldsymbol{y}^{[0]}(t) = \boldsymbol{y}^{[0]}(s^{[0]}(t)), t \in [t_i, t_f]$ by means of a task-constrained motion planner (described in

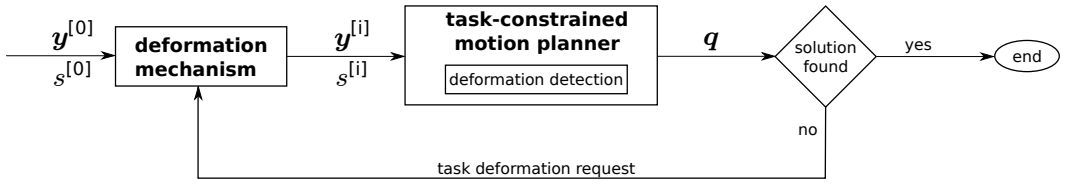


Figure 6.1 A block scheme that gives an overall idea of the planner. The task-constrained motion planner is invoked on the initial reference task trajectory ($i = 0$). If a solution is found, the algorithm ends; on the other side, a deformed task path $\boldsymbol{y}^{[i]}$ and a time history $s^{[i]}$ are computed by the deformation mechanism and the process is repeated.

Section 6.3). If this proves to be too difficult (in the sense formally described in Section 6.3), the planner returns a partial solution whose *limit task point* $\tilde{\boldsymbol{y}}$ (defined as the furthermost task point realized by configurations contained in the tree $\mathcal{T}$) is different from $\boldsymbol{y}(t_f)$. In this case, the deformation mechanism (characterized in Section 6.4) is in charge of deforming the reference task path (and the time history) by means of appropriate heuristic functions which take place at the limit task point. The output is a deformed reference task path $\boldsymbol{y}^{[i]}(s), s \in [s_i, s_f]$ and a time history $s^{[i]}(t)), t \in [t_i, t_f]$, denoted at the generic i-th iteration. Then, the task-constrained motion planner is invoked again on the deformed reference task. This deformation-planning cycle is repeated until a task-constrained solution is found. The pseudocode of the proposed planner is depicted in Algorithm 3.

In Section 6.3 and Section 6.4 we describe the two fundamental components of the planner: the task-constrained motion planner and the deformation mechanism.

6.3 Task-constrained motion planner

The task-constrained motion planner takes as input, at the i-th iteration, a reference task trajectory $\boldsymbol{y}^d(t) = \boldsymbol{y}^{[i]}(s^{[i]}(t)),\ t \in [t_i, t_f]$. We refer to $\boldsymbol{y}^d(t)$ as $\boldsymbol{y}^d$ for brevity. Its goal is to find a feasible collision-free that fulfils the reference task trajectory $\boldsymbol{y}^d$ while avoiding self-collisions and collisions with workspace obstacles (in addition to joint and velocity joint limits, as described at the end of Section 6.1.2).

At this point, the problem to be solved is exactly the same as in Section 5.3. In fact, from the point of view of the task-constrained motion planner, the task is assigned and unmodifiable, as for the one described in Section 5.3. We then use the planner presented in Section 5.3 also in this case, since its results were really promising. We refer the reader to Section 5.3 for the description of this module.

The pseudocode of the task-constrained motion planner is illustrated in Algorithm 4. It should be mentioned that, with a slight abuse of notation, we denote with $\tilde{\boldsymbol{y}}.x$ the quantity x associated with the limit task sample $\tilde{\boldsymbol{y}}$ in Algorithm 4 and Figure 5. The task-constrained motion planner works in an iterative way, building a tree $\mathcal{T}$ rooted at $\boldsymbol{q}(t_i)$. The nodes are configurations of the robot associated with a time instant, while arcs represent whole-body motions that join adjacent

Algorithm 3: Planner

sol_found $\leftarrow$ **false**; $i \leftarrow 0$;
get the initial task path $\boldsymbol{y}^{[0]}$ and time history $s^{[0]}$;
repeat
  build the current task trajectory $\boldsymbol{y}^{[i]} \leftarrow \boldsymbol{y}^{[i]}(s^{[i]})$;
  $[\mathcal{T}, \tilde{\boldsymbol{y}}] \leftarrow$ ConstrainedMotionPlanner($\boldsymbol{y}^{[i]}$);
  if $\tilde{\boldsymbol{y}} = \boldsymbol{y}^{[i]}(t_f)$ **then**
    sol_found $\leftarrow$ **true**;
    $\boldsymbol{y}^* \leftarrow \boldsymbol{y}^{[i]}$; $s^* \leftarrow s^{[i]}$;
  else
    $[\boldsymbol{y}^{[i+1]}, s^{[i+1]}] \leftarrow$ TaskDeformation($\boldsymbol{y}^{[i]}$, $s^{[i]}$, $\mathcal{T}$, $\tilde{\boldsymbol{y}}$);
  end
  $i \leftarrow i + 1$;
until sol_found = **true** **or** i = MAX_DEFORM ;

Algorithm 4: ConstrainedMotionPlanner ($\boldsymbol{y}^d$)

root the tree $\mathcal{T}$ at $\boldsymbol{q}(t_i)$;
$\tilde{\boldsymbol{q}} \leftarrow \boldsymbol{q}(t_i)$; $\tilde{\boldsymbol{y}} \leftarrow \boldsymbol{y}^d(t_i)$; $j \leftarrow 0$;
$\tilde{\boldsymbol{y}}$.exp_fail $\leftarrow 0$; $\tilde{\boldsymbol{y}}$.coll_fail $\leftarrow 0$; $\tilde{\boldsymbol{y}}$.jnt_fail $\leftarrow 0$;
repeat
  $j \leftarrow j + 1$;
  select a random sample $\boldsymbol{y}^d_{\text{rand}}$ from the task trajectory;
  select a random node $\boldsymbol{q}_{\text{near}}$ from $\mathcal{T}$ with probability proportional to $\gamma(\cdot, \boldsymbol{y}^d_{\text{rand}})$;
  get the time instant t_k associated with $\boldsymbol{q}_{\text{near}}$;
  $[\boldsymbol{q}_{\text{new}}, \overline{\boldsymbol{q}_{\text{near}}\boldsymbol{q}_{\text{new}}}, t_{k+1}, \tilde{\boldsymbol{y}}] \leftarrow$ MotionGeneration($\boldsymbol{q}_{\text{near}}, t_k, \tilde{\boldsymbol{y}}$);
  if $\boldsymbol{q}_{\text{new}} \neq \emptyset$ **then**
    add node $\boldsymbol{q}_{\text{new}}$ and arc $\overline{\boldsymbol{q}_{\text{near}}\boldsymbol{q}_{\text{new}}}$ to $\mathcal{T}$;
  end
until $t_{k+1} = t_f$ **or** $\tilde{\boldsymbol{y}}$.exp_fail = MAX_FAIL **or** j = MAX_IT ;
return $[\mathcal{T}, \tilde{\boldsymbol{y}}]$;

nodes and have been verified to be collision-free and feasible. First, a random sample $\boldsymbol{y}^d_{\text{rand}}$ is extracted from $\boldsymbol{y}^d$. Then, an high-compatibility node $\boldsymbol{q}_{\text{near}}$ is extracted among the nodes in the current tree $\mathcal{T}$, with a probability proportional to $\gamma(\cdot, \bar{\boldsymbol{y}})$ (the function $\gamma(\boldsymbol{q}, \bar{\boldsymbol{y}})$ defines the compatibility of the robot configuration $\boldsymbol{q}$ with respect to a certain sample $\bar{\boldsymbol{y}}$). The motion generation is invoked, in order to create a feasible collision-free whole-body motion for the current iteration.

The pseudocode of the motion generation procedure is depicted in Figure 6.2. First, a CoM movement primitive is chosen within the following set

$$U = \left\{ U^{\text{S}}_{\text{CoM}} \cup U^{\text{D}}_{\text{CoM}} \cup \texttt{free\_CoM} \right\} \ , \tag{6.3}$$

with $U^{\text{S}}_{\text{CoM}}$ and $U^{\text{D}}_{\text{CoM}}$ subsets of static and dynamic stepping movements, respectively; `free_CoM` is a special primitive having a stretchable duration. This step was denoted as CoM movement selector and it is described in Section 5.2.1. The chosen primitive comes with a duration T_k and the time interval for the current iteration is uniquely identified as $[t_k, t_{k+1}]$, with $t_{k+1} = t_k + T_k$. The same module also provides a desired trajectory for the CoM ($\boldsymbol{z}^d_{\text{CoM}}$) and for the swing foot ($\boldsymbol{z}^d_{\text{swg}}$) within the same time interval.

Procedure MotionGeneration($\boldsymbol{q}_{\text{near}}$, t_k, $\tilde{\boldsymbol{y}}$)

pick from (6.3) a random CoM primitive $\boldsymbol{u}^k_{\text{CoM}}$ of duration T_k;
compute the associated CoM trajectory $\boldsymbol{z}^d_{\text{CoM}}$ and swing foot trajectory $\boldsymbol{z}^d_{\text{swg}}$;
extract the portion of task trajectory $\boldsymbol{y}^d$ in $[t_k, t_k + T_k]$;
build the augmented task $\boldsymbol{y}_a = (\boldsymbol{y}, \boldsymbol{z}_{\text{CoM}}, \boldsymbol{z}_{\text{swg}})$;
repeat
    generate motion by integrating joint velocities (4.6);
    if collision **then**
        if $\boldsymbol{y}^d(t_k) = \tilde{\boldsymbol{y}}$ **then**
            // *(expanding from a limit configuration)*;
            $\tilde{\boldsymbol{y}}$.exp_fail $\leftarrow$ $\tilde{\boldsymbol{y}}$.exp_fail + 1;
            $\tilde{\boldsymbol{y}}$.coll_fail $\leftarrow$ $\tilde{\boldsymbol{y}}$.coll_fail + 1;
        end
        return $[\emptyset, \emptyset, \emptyset, \tilde{\boldsymbol{y}}]$;
    else if joint limit/velocity bound violation **then**
        if joint limit violation **then**
            if $\boldsymbol{y}^d(t_k) = \tilde{\boldsymbol{y}}$ **then**
                // *(expanding from a limit configuration)*;
                $\tilde{\boldsymbol{y}}$.exp_fail $\leftarrow$ $\tilde{\boldsymbol{y}}$.exp_fail + 1;
                $\tilde{\boldsymbol{y}}$.jnt_fail $\leftarrow$ $\tilde{\boldsymbol{y}}$.jnt_fail + 1;
            end
        end
        return $[\emptyset, \emptyset, \emptyset, \tilde{\boldsymbol{y}}]$;
    end
until $t = t_k + T_k$;
if new limit task point reached **then**
    $\tilde{\boldsymbol{y}} \leftarrow \boldsymbol{y}^d(t_k + T_k)$ // (*update the limit task point*);
    $\tilde{\boldsymbol{y}}$.exp_fail $\leftarrow$ 0; $\tilde{\boldsymbol{y}}$.coll_fail $\leftarrow$ 0; $\tilde{\boldsymbol{y}}$.jnt_fail $\leftarrow$ 0;
end
return $[\boldsymbol{q}_{\text{new}}, \overline{\boldsymbol{q}_{\text{near}}\boldsymbol{q}_{\text{new}}}, t_k + T_k, \tilde{\boldsymbol{y}}]$

Figure 6.2 The pseudocode of the motion generation procedure.

At this point, joint motions are generated so as to realize the portion of the task trajectory $\boldsymbol{y}^d$ between t_k and t_{k+1}, together with $\boldsymbol{z}^d_{\text{CoM}}$ and $\boldsymbol{z}^d_{\text{swg}}$. This module was denoted as joint motion generation and it is described in Section 5.2.2. In particular, let $\boldsymbol{y}_a = (\boldsymbol{y}^T \ \boldsymbol{z}^T_{\text{CoM}} \ \boldsymbol{z}^T_{\text{swg}})^T$ be the augmented task vector, except for `free_CoM`, for which we choose $\boldsymbol{y}_a = (\boldsymbol{y}^T \ \boldsymbol{z}^T_{\text{swg}})^T$. Denote by $\boldsymbol{J}_a$ the Jacobian matrix of $\boldsymbol{y}_a$ w.r.t. $\boldsymbol{q}_{\text{jnt}}$, and by $\boldsymbol{e} = \boldsymbol{y}^d_a - \boldsymbol{y}_a$ the augmented task error, where $\boldsymbol{y}^d_a(t)$ is the reference value of the augmented task in $[t_k, t_{k+1}]$. Joint velocity commands are then computed as

$$\boldsymbol{v}_{\text{jnt}} = \boldsymbol{J}^\dagger_a(\boldsymbol{q}_{\text{jnt}}) \left(\dot{\boldsymbol{y}}^d_a + \boldsymbol{K}\boldsymbol{e}\right) + (\boldsymbol{I} - \boldsymbol{J}^\dagger_a(\boldsymbol{q}_{\text{jnt}})\boldsymbol{J}_a(\boldsymbol{q}_{\text{jnt}}))\boldsymbol{w}, \tag{6.4}$$

where $\boldsymbol{J}^\dagger_a$ is the pseudoinverse of $\boldsymbol{J}_a$, $\boldsymbol{K}$ is a positive definite gain matrix, and $\boldsymbol{w}$ is a bounded norm n-vector projected in the null space of $\boldsymbol{J}_a$ through the orthogonal projection matrix $\boldsymbol{I} - \boldsymbol{J}^\dagger_a\boldsymbol{J}_a$. Use of eq. (6.4) guaranteed $\dot{\boldsymbol{e}} = -\boldsymbol{K}\,\boldsymbol{e}$, i.e., exponential convergence to the desired augmented task trajectory. The trajectories generated by eq. (6.4) are continuously checked for collisions as well as for position and velocity joint limits. The static equilibrium is also explicitly

checked for the `free_CoM` movement primitive. If any of these conditions is violated, the motion generation is interrupted and no node is added to the tree; otherwise, integration proceeds until t_{k+1} is reached. In the latter case, we have obtained a feasible, collision-free joint motion $\boldsymbol{q}_{\text{jnt}}(t)$, $t \in [t_k, t_{k+1}]$, that complies with a portion of the task. Its final configuration $\boldsymbol{q}_{\text{new}}$ is added as a node in the tree $\mathcal{T}$. Moreover, the motion joining $\boldsymbol{q}_{\text{near}}$ and its final configuration $\boldsymbol{q}_{\text{new}}$ is added to the tree $\mathcal{T}$ as an arc.

Repeated calls to the above-mentioned procedure result in a solution to the motion planning problem for the reference task trajectory $\boldsymbol{y}^d$. However, there are cases where it is better to abort the search of a solution for $\boldsymbol{y}^d$ and change the task. As example, the reference task path might be really close to an obstacle or it is not compatible with the environment (see Section 6.4). To this aim, at the end of each iteration of the task-constrained motion planner, the algorithm checks if one of the following situation occurs

- a maximum number of iterations has been reached (MAX_IT in Algorithm 4);
- the number of failed expansions from the *limit task point* $\tilde{\boldsymbol{y}}$ (defined as the furthermost task point met by the tree) exceeds a predefined threshold (MAX_FAIL in Algorithm 4).

In such a case, the task-constrained motion planner is aborted and the task should be deformed. These are the conditions that we denoted in Figure 6.1 as deformation detection, i.e., the conditions for which a deformation would help in finding a solution for the motion planning problem.

The rationale behind the first deformation condition is trivial. In fact, even assuming that a solution exists, a planner may fail in finding a solution because its time budget (or equivalently a predefined number of iterations) is exceeded.

The second condition needs more attention. Let formally define the limit task point $\tilde{\boldsymbol{y}}$. It is the furthermost task sample met by the task-constrained motion planner. Note that this point changes over time. When the task-constrained motion planner is invoked, it is initialized as $\tilde{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{q}(t_i))$ and its associated time is set to $\tilde{t} = t_i$ (since $\boldsymbol{q}(t_i)$ is the only configuration in the tree). Then, whenever the task-constrained motion planner generates a feasible and collision-free whole-body motion in $[t_k, t_{k+1}]$, its endpoint becomes the limit task point if $t_{k+1} > \tilde{t}$. In formula, if the motion is feasible in $[t_k, t_{k+1}]$ (arriving in $\boldsymbol{q}_{\text{new}}$ at t_{k+1}) and $t_{k+1} > \tilde{t}$, then the limit task point is updated as $\tilde{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{q}_{\text{new}})$, while its associated time is set to $\tilde{t} = t_{k+1}$. When the limit task point is updated, the number of failing expansions is reset to zero. On the other side, if the task-constrained motion planner fails in expanding a node associated with the limit task sample, the number of fails is increased of one. It is important to mention that there are several *limit configurations* in the tree associated with the limit task point. In fact, a configuration $\tilde{\boldsymbol{q}}$ in the tree is defined as a limit configuration if $\tilde{\boldsymbol{y}} = \boldsymbol{f}(\tilde{\boldsymbol{q}})$, with $\tilde{\boldsymbol{y}}$ the limit task point. The role of these configurations within our framework will be clear in the next section.

There is a reason for which we decide to deform the reference task when a predefined number of expansions from the limit task point occurs. In fact, it is reasonable to deduce that the planner

Algorithm 6: TaskDeformation($\boldsymbol{y}^{[i]}$, $s^{[i]}$, $\mathcal{T}$, $\tilde{\boldsymbol{y}}$)

if $\tilde{\boldsymbol{y}}$.jnt_fail $\geq$ $\tilde{\boldsymbol{y}}$.coll_fail **then**
  // (*robot-based heuristic*);
  retrieve a limit configuration $\tilde{\boldsymbol{q}}$ from the tree $\mathcal{T}$;
  compute the line joining the limit task point $\tilde{\boldsymbol{y}}$ to the CoM at $\tilde{\boldsymbol{q}}_{\mathrm{CoM}}$;
  deform $\boldsymbol{y}^{[i]}$ by adding to $\boldsymbol{\sigma}^{[i]}$ a new control point along this line, obtaining $\boldsymbol{y}^{[i+1]}$;
else
  // (*obstacle-based heuristic*);
  compute the line joining the limit task point $\tilde{\boldsymbol{y}}$ to the closest obstacle point;
  deform $\boldsymbol{y}^{[i]}$ by adding to $\boldsymbol{\sigma}^{[i]}$ a new control point along this line, obtaining $\boldsymbol{y}^{[i+1]}$;
end
compute $s^{[i+1]}$ proportional to the path length of $\boldsymbol{y}^{[i+1]}$;
return $[\boldsymbol{y}^{[i+1]}, s^{[i+1]}]$

Figure 6.3 Pseudocode of the deformation mechanism.

has difficulties in fulfilling the task from the limit task point on. Deforming the task around the limit task sample might solve this problem, since the deformed path will not contain the limit task sample (where the motion planner encountered problems) anymore. Details on the task deformation are given in Section 6.4.

If none of the above-mentioned deformation conditions occurs, the task-constrained motion planner has found a solution for the motion planning problem. The current reference task path and time history are assigned as the final reference task path $\boldsymbol{y}^* = \boldsymbol{y}^{[i]}$ and time history $s^* = s^{[i]}$. Moreover, the joint motions that fulfil the above-defined task are found by backtracking in the tree $\mathcal{T}$, as in Section 5.3.

For reasons that will be clear in Section 6.4, we count the number of failings from the limit task point due to collisions and joint limits, separately, as it can be seen in Algorithm 4.

6.4 Deformation mechanism

In this section, we describe the deformation mechanism we use to deform a reference task path. The input of this module (denoted as deformation mechanism in Figure 6.1) is the latest task path $\boldsymbol{y}^{[i]}(s), s \in [s_i, s_f]$ and the time history $s^{[i]}(t), t \in [t_i, t_f]$. Moreover, the limit task point $\tilde{\boldsymbol{y}}$ and the associated exploration tree $\mathcal{T}$ complete the input for this module. The pseudocode of the deformation mechanism is given in Algorithm 6.

Recall that the current task path[2] $\boldsymbol{y}^{[i]}$ is parametrized by a set of control points, collected in a vector $\boldsymbol{\sigma}^{[i]}$; the endpoints are however fixed. The idea is to deform $\boldsymbol{y}^{[i]}(s)$ by inserting a new control point in $\boldsymbol{\sigma}^{[i]}$. The time history $s^{[i]}(t)$, $t \in [t_i, t_f]$ will then be adapted to the new path. Two heuristics are used to choose where to place the new control point.

We define two heuristic functions for choosing where to insert the control points. Note that, in case there are no obstacles in the environment, our task-constrained motion planner will be

[2]With a slight abuse of notation, in this section (and in Algorithm 6) $\boldsymbol{y}^{[i]}$ stands for the path $\boldsymbol{y}^{[i]}(s)$, $s \in [s_i, s_f]$, rather than the full trajectory.

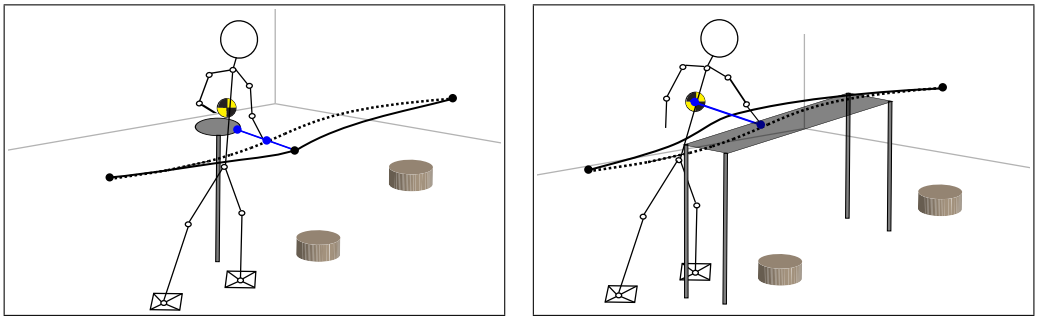


Figure 6.4 Example of the heuristic functions. Solid black lines refer to the initial task paths; dotted black lines refer to the deformed paths. (left) Obstacle-based heuristic: its aim is to push the path away from the closest obstacle (here the table). (right) Robot-based heuristic: its aim is to get the path closer to the robot CoM.

able to find a solution for the reference task, thanks to its randomness. The aim of this section is to present heuristics that are able to deform the task in presence of obstacles.

We refer to the first heuristic function as *obstacle-based heuristic*. Its purpose is simply to push the path away from the closest obstacle. Consider Figure 6.4. To this aim, a new control point is inserted on the line joining a representative point of the closest obstacle (e.g., its closest point) and $\tilde{\boldsymbol{y}}$, with a predefined distance w.r.t. the latter (as in the left side of Figure 6.4). The height of the control point is set as the z-component of $\tilde{\boldsymbol{y}}$[3]. The result is that the entire task path is pushed away from the obstacle, as shown in the left side of Figure 6.4.

Despite its simplicity, this heuristic is not general. In fact, there exist cases in which this function is not the best strategy to be employed. As example, consider the scenario in the right side of Figure 6.4. Here, the reference task path (defined for one hand of the humanoid robot) passes over an obstacle (a table). If one applies the obstacle-based heuristic the result is that the path is deformed along the vertical axis, making it even more difficult to accomplished. A more suitable deformation in this case is along the lateral direction, as in the right side of Figure 6.4.

For this reason, we introduce the *robot-based heuristic*. Its purpose is to get the path (locally) closer to the robot. More in details, it reduces the distance between the limit task point $\tilde{\boldsymbol{y}}$ and the robot CoM at the limit configuration (position component of $\tilde{\boldsymbol{q}}_{\mathrm{CoM}}$). The intuition behind this mechanism is that constrained planning is more difficult if the task path is far from the humanoid CoM, because (1) motion generation must realize an extended task which includes both the original task variables and the CoM position (2) outstretched postures typically push the joints towards the limit of their available range. To bring the task path closer to the humanoid, a *limit configuration* is associated to the limit task point $\tilde{\boldsymbol{y}}$ by extracting from $\mathcal{T}$ a node $\tilde{\boldsymbol{q}}$ such that $\tilde{\boldsymbol{y}} = \boldsymbol{f}(\tilde{\boldsymbol{q}})$, and a new control point is inserted on the line joining $\tilde{\boldsymbol{y}}$ with the humanoid CoM at $\tilde{\boldsymbol{q}}$, again at a predefined distance from $\tilde{\boldsymbol{y}}$ (see Figure 6.4). We noticed that this heuristic is particularly helpful for tasks whose execution requires the robot to stretch (as in the right of

[3]We emphasize that this choice for the z-component of the control point is arbitrary. Other choices are possible and we are currently evaluating them.

Figure 6.4), bringing some of its joints very close to their limits. The result of applying the robot-based heuristic for such a case is that the robot has no need to stretch anymore to accomplish the task. Then, it is able to fulfil the task in a more human-like way, without 'stressing" its joints. From a more practical point of view, the line joining $\tilde{\boldsymbol{y}}$ and the robot CoM at the limit configuration (position component of $\tilde{\boldsymbol{q}}_{\mathrm{CoM}}$, the CoM part of $\tilde{\boldsymbol{q}}$) is computed. A new control point is inserted along this line, with a predefined distance w.r.t. the latter. This procedure can be seen in the right of Figure 6.4. As for the other heuristic, the height of this control point is set as the z-component of $\tilde{\boldsymbol{y}}$.

Assume that the task-constrained motion planner has identified that the reference task has to be deformed (i.e., a predefined number of expansions from $\tilde{\boldsymbol{y}}$ reveals in failures). A policy for selecting which heuristics to use has to be introduced. We bound it to the two main reasons for which an expansion may fail: joint limit violations and collisions. We choose the following policy: if the number of failed expansions due to the former is greater than the one due to the latter, then the robot-based heuristic is selected. On the other side, the obstacle-based heuristic is picked. We motivate this policy by saying that, in the first case, the robot is likely trying to fulfil the task by stretching itself and then violating some of its joint limits. The choice of using the robot-based heuristic makes sense since its effect is to get the path closer to the robot so that it is more compliant with the task. As said before, the result is that the robot has no need to stress its joints to reach the task and, then, the joint limits violation problem is alleviated. On the other side, if the main reason of failures is the collisions, the obstacle-based heuristic is chosen.

This is depicted in the pseudocode of Figure 6.2, where the number of failed expansions due to joint limits and collisions are counted separately. Note that a failure is counted only if a node pointing to the limit task point $\tilde{\boldsymbol{y}}$ is expanded. The policy is then depicted in Algorithm 6.

At this point, it should be clear how to deform the reference task path $\boldsymbol{y}^{[i]}$ and obtain $\boldsymbol{y}^{[i+1]}$ (i.e., by inserting a new control point based on the chosen heuristic function). Regarding the time history $s^{[i+1]}$, it is adapted to the deformed task path $\boldsymbol{y}^{[i+1]}$ via uniform scaling within a total duration proportional to the new path length.

The reason for which we have defined such a deformation mechanism is that it is really fast to be computed (any heuristic function is chosen, the computations to be performed are really easy), then it is very good from a computational point of view. Here, we do not provide any form of optimal deformation strategy and other heuristic functions can be taken into account. Our goal was to introduce heuristics that easily deform the task path and are really computationally fast. This because the task-constrained motion planner takes time to find a solution; then, the smallest is the time to deform the path, the smallest is the time to find a solution for the planning problem.

6.5 Planning experiments

The proposed planner has been implemented in V-REP (an overview of this simulator is given in Section 4.4.2) on an Intel Core 2 Quad at 2.66 GHz. We have chosen NAO (a description of the

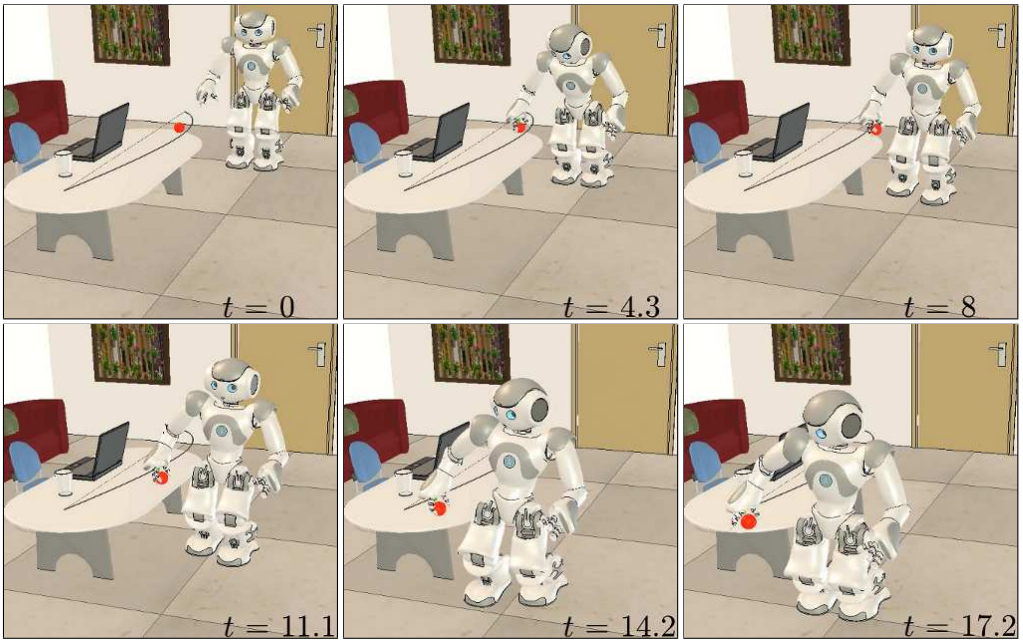


Figure 6.5 Planning experiment 1: snapshots from a final solution. The dotted black line represents the initial reference task path while the solid black line is the deformed task path.

robot is provided in Section 4.4.1) by Aldebaran Robotics as humanoid robot for our experiments.

We impose the same parameter set as in the experiments shown in Section 5.4. Just as reminder, we set $\boldsymbol{K} = \mathrm{diag}\{2, 2, 1\}$ for generating the joint motions in eq. (6.4), while $\boldsymbol{w}$ was randomly chosen, with a bound norm limit at 0.4 rad/sec. The CoM movement primitive set is defined as in eq. 6.3. Its static subset $U^{\mathrm{S}}_{\mathrm{CoM}}$ is composed by different primitives in forward and backward directions in the range $[0.03, 0.08]$ m and in lateral directions in the range $[0.01, 0.03]$ m. Each primitive in $U^{\mathrm{S}}_{\mathrm{CoM}}$ has a duration of 2 s. The dynamic subset $U^{\mathrm{D}}_{\mathrm{CoM}}$ contains a starting step with length of 0.038 m and duration of 1.6 s, a cruise step with length 0.04 m and duration 0.425 s and a stopping step with length 0.038 m and duration 1.325, all in forward direction. The total number of primitives is 16. We have used B-splines as representation for the task path.

Here we present two planning experiments. In Section 6.5.1, the robot must move an object from one side to the other of a table, while avoiding collisions with the table itself. In the scenario depicted in Section 6.5.2, the robot must place an object on a low stool while avoiding collisions with the obstacle of the environment.

6.5.1 Experiment 1

As already mentioned, the robot has to pick an object (a ball) and move it from one to the other side of a long table. This is a manipulation task, expressed as a constraint for the right hand of the humanoid robot. Snapshots from a final solution are shown in Figure 6.5. As first reference task path, we simply assign a simple composition of two straight lines: the first from the initial right hand position to the point where the ball is placed and the second from that point to the position where we want to place the ball (on the other side of the long table). The starting task is denoted

data	exp 1	exp 2
init planning time (s)	26.1	19.3
deformation time (s)	1.33	2.32
final planning time (s)	15.9	10.2
final tree size (# nodes)	68.1	61.9
motion duration (s)	17.2	13.3

Table 6.1 Planner performance at a glance.

in Figure 6.5 as a dotted black line. The time history is simply set to $s^{[0]}(t) = t, \forall t \in [t_i, t_f]$, with $t_i = 0$ s and $t_f = 17.2$ s. The choice of the initial task path is motivated also from a user point of view, since it is really simple and a user is able to assign such a task.

The task-constrained motion planner is invoked on the initial reference task.

The task-constrained motion planner is not able to find a solution for the initial task path: after covering less than one third of it, further expansion fails repeatedly due to joint limit violations as the robot extends its upper body and arm in order to stay on the path while avoiding collisions between its legs and the table. Since the main reason of failures is the joint limit violation, the robot-based heuristic is used. Note how our planner selects the appropriate heuristics. In fact, the deformation would be along the vertical axis, by using the obstacle-based heuristic, causing even an harder task to be fulfilled to the motion planner. The deformed reference task path is depicted in Figure 6.5 with a solid black line. As one may expect, the deformed problem is significantly easier to be solved since the task is closer to the robot (and more distant to the table). Moreover, the "stretched" configurations (and consequently the joint limit violations) are not generated anymore, since the hand is constrained along a curve significantly less influenced by the table. The task-constrained motion planner is invoked again on the deformed reference task and a solution is found. This solution includes different types of CoM movement primitives. In fact, a dynamic gait copes with the initial part of the task while a `free_CoM` places the ball on its final position. Finally, Table 6.1 collects some data for this experiment. It reports the time needed by the task-constrained motion planner to trigger a task deformation (init planning time), the time needed for deforming the path (deformation time) and the time needed to find the final solution on the deformed task (final planning time). In the same table, we also report the number of nodes in the tree of the final solution (final tree size) and the overall duration of motion (motion duration). Since our planner is randomized, the data refer to the average over 10 runs. Note that the time needed for fulfilling just a portion of the initial task (init planning time) is bigger than the time needed to fulfil the entire deformed task (final planning time). This confirms that a task deformation can significantly help the planner in finding a solution to the TCMP problem.

This experiment validates the robot-based heuristic and shows the versatility obtained by the introduction of the deformation mechanism, since the task-constrained motion planner was not enough for solving this TCMP problem (or, at least, it would take a relevant amount of planning

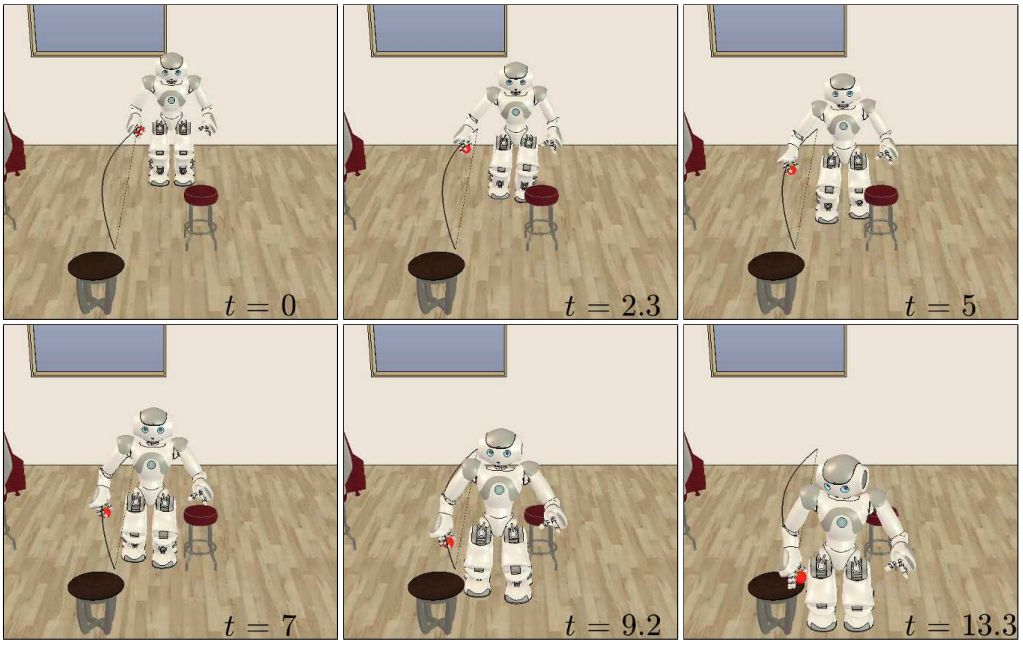


Figure 6.6 Planning experiment 2: snapshots from a final solution. The dotted black line represents the initial reference task path while the solid black line is the deformed task path.

time to find a solution), even if the initial reference task is not in collision with any obstacle of the environment.

6.5.2 Experiment 2

In the second experiment, the robot must place an object (again, a ball) on a low stool. The robot has already grasped the ball. The task is again a manipulation task assigned for the right hand of the humanoid robot. The initial reference task path is again assigned as a straight line joining the initial right hand position and a point on the low stool where the object has to be released while the time history is $s^{[0]}(t) = t, \forall t \in [t_i, t_f]$, with $t_i = 0$ s and $t_f = 13.7$ s. Snapshots from a final solution are given in Figure 6.6.

The main difference w.r.t. the previous scenario is the low stool placed in the middle of the room. At first sight, this stool seems to be not a problem for the planner. However, the task-constrained motion planner is not able to solve the TCMP problem, mainly due to the collisions between the stool in the middle and the left leg of the robot. Then collisions are the main reason for invoking the deformation mechanism, that uses the obstacle-based heuristic to deform the reference task path. Note how the deformation mechanism selects the appropriate heuristic. In this case, the robot-based heuristic is not helpful. On the contrary, it would get the path closer to the robot (then closer to the obstacle), causing even more difficulties in finding a solution. The deformed path is shown in Figure 6.6 as a solid black line. Then, the task-constrained motion planner is invoked again on the deformed path and a solution is found, since the (deformed) task steers the robot away from the stool.

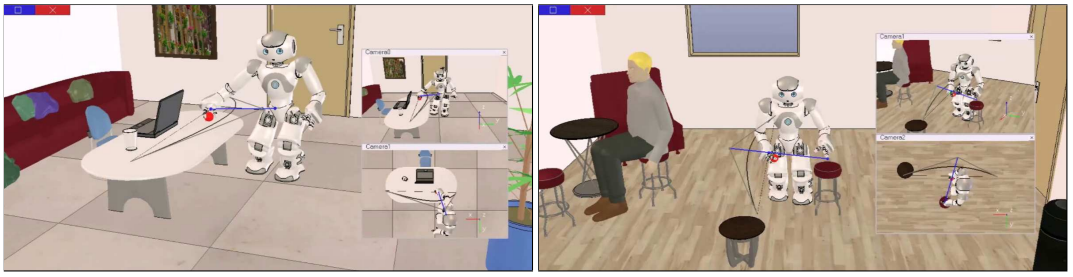


Figure 6.7 The deformation mechanism in the two proposed experiments. Dotted black lines refer to initial reference tasks while the solid black lines represent the deformed tasks. The configurations shown are the limit task configurations. (left) The robot-based heuristic computes the line joining the limit task point and the robot CoM (computed at the limit task configuration) and inserts a new control point along that line. (right) The obstacle-based heuristic computes the line joining the limit task sample and the obstacle closest point and places a new control point along that line.

As for the other scenario, the final solution is composed by different CoM movement primitives. A dynamic gait is selected for the first part of the assigned task, while a `free_CoM` places the ball over the low stool. This experiment validates the obstacle-based heuristic and illustrates the advantage of introducing the deformation mechanism. In fact, the task-constrained motion planner was not able to find a solution by itself, even if the initial reference task is valid, i.e., it does not collide with any obstacle in the environment.

Data for this experiment are collected in Table 6.1. This table reports the initial planning time before detecting that a deformation is needed (init planning time), the time needed for deforming the reference task path (deformation time) and the planning time for the deformed path (final planning time). It also reports the tree size for the final solution (final tree size) and the overall duration of motion (motion duration). As for the previous experiment, note that the time needed for fulfilling just a portion of the initial reference task (init planning time) is bigger than the time needed to fulfil the entire deformed task (final planning time). This confirms the advantage of introducing the deformation mechanism.

Figure 6.7 illustrates the deformations applied in the two proposed experiments.

6.6 An extension to the on-line planning

The approaches described so far work in an offline fashion, i.e., a snapshot of the environment is taken once and the motions are compute to deal with that environment. If the environment is static (i.e., it does not change), these motions perfectly solve the problem.

In this section, we present a modification of the proposed framework towards the on-line planning scenario, i.e., assuming that the environment may change. This a key feature in case, e.g., if an obstacle is not detected (due to an occlusion) and then not taken into account in the planning phase. Or, even more importantly, if there are moving obstacles in the environment, e.g., humans. An ideal planner should be able to deal with this scenario, possibly in real-time. As first step toward this direction, we propose a planner that is able to detect changes in the environment

and then react by stopping the current motions and replanning new motions for the robot. These motions have still to ensure the accomplishment of the task (possibly deformed), dealing with the dynamic environment. It is worth mentioning that we do not want to propose a real-time motion planner in this section, due to the computational payload of the framework presented in this chapter. We just propose a modification of the framework that deals with dynamic environments.

Since the robot has to perceive changes of the environment, it is equipped with a exteroceptive sensor. We have chosen a depth camera mounted on the robot head to this aim. We assume that a nominal plan (which was computed for the initial environment) is available and some unexpected obstacles appear. The idea is to use the approach presented so far to compute motions from the current robot state. Clearly, we need a mechanism for detecting the occurrence of unexpected obstacles and their relevance to the current plan.

6.6.1 Detecting possible dangerous situations

Assume to have a nominal plan, i.e., the TCMP problem was solved for the initial planning problem and a joint motion is available (e.g., using the planner in Section 6.2). During the nominal plan, we recorded also the distance between the robot and the closest obstacle (that may vary over time), at each time step. This distance can be easily computed with the depth camera mounted on the robot head, that gives as output a distance for each pixel of its image. A simple feature extraction algorithm is able to provide the information needed. We use this information $d^*(t), \forall t \in [t_i, t_f]$ to check if a change in the environment occurs.

Once the nominal plan and $d^*(t), \forall t \in [t_i, t_f]$ are available, the robot starts to execute this plan. At the same time, the same algorithm used for computing d^* runs in background. Its aim is to compute the current distance d to the closest obstacle. A dangerous situation occurs if d and the corresponding distance d^* on the nominal plan verify the following two conditions

- $\|d^* - d\| \leq d_1$, with d_1 a predefined threshold;
- $\|d\| \geq d_2$, with d_2 a predefined threshold.

The rationale behind the first condition is to check if a significant change w.r.t. the nominal plan happens. In fact, if the distance to the closest object to the robot changes, it means that the environment has been changed in a significant way. The aim of the second condition is to avoid unnecessary planning actions. In fact, in the case the closest obstacle is far from the current robot position, no change is needed from the nominal plan. The combination of these two conditions provides a good policy to detect if there is a relevant change (and then a potential dangerous situation) in the environment.

When a dangerous situation is detected, the reference task has to be deformed. The idea is that, though the deformation, the robot is pushed away from the closest obstacle (that causes the dangerous situation) while continuing to fulfil the (deformed) task. The deformation mechanism, together with the heuristic to be used, are novelties with respect to the offline case and will be described in the next section.

6.6.2 Deformation mechanism and heuristic

Once a dangerous situation has been detected, the reference task should be deformed. The idea is to push the robot away from the closest obstacle that caused the deformation mechanism activation. The deformation has the aim to steer the robot away from the closest obstacle. This kind of deformation is different w.r.t. the one presented in Section 6.4. Here, the deformation to be applied is local. On the contrary, it is global in the mechanism in Section 6.4. In fact, when a new control point is inserted, the whole shape of the path changes. Here, we cannot allow that because the robot has already executed a portion of the task when the deformation mechanism is invoked and it will cause discontinuity on the reference task. The deformation is then applied locally, in the sense that only the portion of the reference task between the task point where the robot recognizes the dangerous situation and the end of the task is allowed to be deformed. The complementary portion of the task should remain unchanged. This is equivalent to consider a new curve, having as $\boldsymbol{\sigma}$ only a subset of the control points of the nominal curve. Finally, the task-constrained motion planner is invoked on the (locally deformed) reference task and, in case of other dangerous situations, the process is repeated.

Having defined the deformation, we can turn our attention to the heuristic to be used. Since we stated that the aim is to push the task away to the closest obstacle, we use the obstacle-based heuristic, with a minor modification. The line between the closest point of the obstacle (now provided by the exteroceptive sensor) and the task point having the minimum distance w.r.t. this point is computed. A control point is inserted along this line, with a predefined offset w.r.t. the latter. Once the task is locally deformed, the task-constrained motion planner presented in Section 6.2 is invoked on the deformed task. Note that the planner is invoked just for the remaining portion of the reference task (the deformed part).

As summary, the on-line modification consists in the following steps

- compute a nominal plan and the distance to the closest obstacle w.r.t. the robot $d^*(t), \forall t \in [t_i, t_f]$;
- start to execute the nominal plan;
- if a dangerous situation is detected (Section 6.6.1), locally deform the reference task (Section 6.6.2);
- invoke the task-constrained motion planner presented in Section 6.2 on the deformed task;
- repeat the procedure (except the first two points) if another dangerous situation is detected.

6.6.3 Planning experiments

We validate the on-line extension for the scenario depicted in Figure 6.8. We have used the Kinect[4] as exteroceptive sensor.

[4]https://www.microsoft.com/en-us/kinectforwindows/

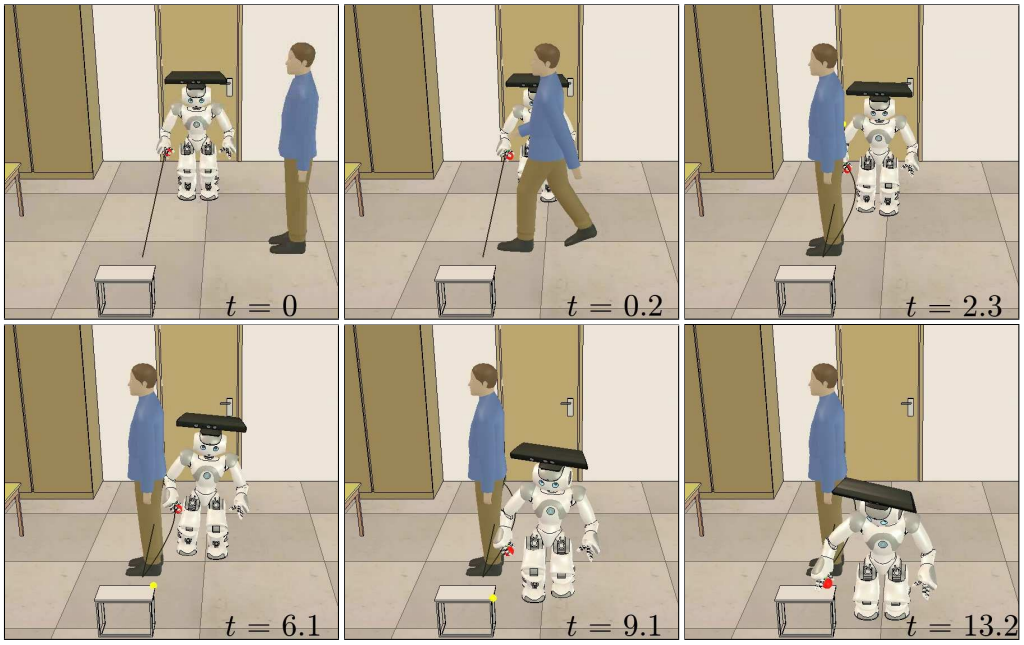


Figure 6.8 Extension to on-line scenario: snapshots from a solution. The dotted black line represents the initial reference task path while the solid black line is the deformed task path.

The robot must place an object (a ball) on the low table of Figure 6.8, then a task is assigned for the right hand of the humanoid robot. The initial reference path is a simple straight line joining the initial right hand position and a point over the table. The time history is set as $s = t, \forall t \in [t_i, t_f]$, with $t_i = 0$ s and $t_f = 13.2$ s. As it is evident from the first snapshot of Figure 6.8, the human does not interfere with the task at the beginning and a nominal plan is found, e.g. by using the task-constrained motion planner presented in Section 6.2.

The robot starts to execute the nominal plan and the human moves (second snapshot of Figure 6.8), completing its motion at $t = 2.3$ s (third snapshot of Figure 6.8). At this point, our planner recognizes a dangerous situation, since the human is on the way of the path that the robot should fulfil. In fact, the human is perceived as the closest obstacle, whose distance is significantly smaller w.r.t. to the one in the nominal plan, then detecting a dangerous situation. The robot stops and invokes the deformation mechanism that deforms the remaining portion of the task (fourth snapshot in Figure 6.8). The task-constrained motion planner is invoked on the deformed path and it computes feasible joint motions that fulfils the task (last two snapshots).

6.7 Conclusions

In this section, we have presented a motion planner able to deform the task, if needed, that is hinged on the task-constrained motion planner described in Chapter 5. For this reason, it can be seen as an extension of the framework presented in Chapter 5.

The motion planner presented in this chapter is automatically able to detect when the reference path path has to be deformed and we have proposed two heuristic functions to deform the path. We also have provided a policy to select among the heuristics.

The algorithm has been validated in V-REP for the humanoid robot NAO. We also have presented an extension to the on-line scenario, showing how our framework can be modified to cope with dynamic environments.

In the next future, we want to search for cases not handled from the proposed policy and extend the framework to the on-line scenario, possibly in real-time. The real-time planning is indeed the topic of the next chapter. Another interesting development would be to allow deformation (rather than simple adaptation) on the time history as well. For example, this may allow to solve difficult planning problems where the main reason for failed expansions is violation of joint velocity bounds (rather that collision or violation of joint limits) by slowing down the motion.

The planner presented in this chapter has been accepted in [21] and it will be presented at the *2016 IEEE Int. Conference on Robotics and Automation, Stockholm, Sweden, 16–21 May 2016.*

II

REAL-TIME MOTION PLANNING FOR EVASIVE MOTIONS

Chapter 7

Real-time planning and execution of evasive motions for a humanoid robot

The approaches we have developed so far are deliberative and they require the knowledge a priori of the environment. There are situations, such as when a robot has to avoid an incoming obstacle, in which the robot does not have the time to plan its motion in an off-line fashion but it has to plan and execute a motion in real-time. This problem can be classified as a part of the safety coexistence between humans and robots. Safety is a fundamental feature for letting robots out of factories and to achieve safe behaviors, both for humans and robots. There are many works that handles the collision avoidance problem in the presence of moving obstacles, especially for manipulators [24, 35, 67] and for mobile robots [60, 61, 125]. However, there are very few approaches that face the safety problem for humanoid robots. This is due to the difficulties introduced by a humanoid. As example, a humanoid robot is not a free-flying system and its motion must be planned appropriately, taking into account the equilibrium constraint.

In this chapter, we do not want to provide a framework for dealing with the entire safety problem for humanoid robots but we focus only on its lowest level. Motivated by an on-going research project COMANOID[1] which targets the deployment of humanoid robots in aeronautic assembly operations, we address in this chapter the basic problem of performing real-time evasion maneuvers for a humanoid robot when a moving obstacle (such as a human) enters its vicinity. In other words, we want to equip the humanoid robot with reactive capabilities, in order to provide a safe coexistence between humans and humanoids. Solving this problem is important also for reacting to unexpected situations that may happen in the environment (supposed to be highly dynamic). Among the available fundamental tools we find basic layers allowing detection and avoidance of moving obstacles as well as the definition of safety measures in [72].

As mentioned before, a humanoid introduces novel challenges for the reactive problem, i.e., the problem of avoiding a moving obstacle that moves towards the robot. In fact, it is able to perform evasive motion by performing steps. Moreover, any movement must be planned so as to

[1] See www.comanoid.eu

maintain balance [55]. A closely related problem, which also involves stepping and balancing, is push recovery in humanoid robots. Several authors have investigated this issue; e.g., see [65, 86].

We propose an algorithmic framework which in principle can accommodate (and in which we intend to test) various evasion strategies. The proposed method goes through several conceptual steps. First, we define a safety area for the humanoid robot. When a moving obstacle enters the safety area, its approaching direction with respect to the robot is computed. Based on this information, a suitable evasion maneuver represented by footsteps is generated using a controlled unicycle as a reference model. From the footstep sequence, we compute an appropriate trajectory for the CoM of the humanoid, which is finally used to generate joint motion commands that track such trajectory.

Another key point within the reactive problem (but in general in the safety context) is the reaction time, i.e., the time from the detection of the incoming obstacle in the safety area and the first reactive motion. This time should be kept as small as possible in order to provide real-time capabilities to the robot. In recent years, some approaches proposed algorithms for on-line generation of humanoid motions [46, 88]. Unlike these approaches, we rely on the existence of analytical (closed-form) expressions relating a desired Zero Moment Point (ZMP) trajectory to the associated bounded CoM trajectory, as illustrated in [74, 75]. Closed form solution, as we will see through the chapter, are essential for a real-time implementation of the proposed framework.

In the literature, there are also other approaches that faces the problem of the real-time motion planning. The approach in [94] treats the motion planning in dynamic environments as an optimization problem in the configuration space. It interleaves planning and execution of the robot in an adaptive manner to balance between the planning horizon and responsiveness to obstacle. In other works, this approach consists in running two parallel threads, one aimed to the execution of the motion and one devoted to the planning of the motion. Roughly speaking, while the robot is executing an action, the planning thread computes the motion that will be executed at the next iteration. At each cycle, the planning thread solves an optimization problem composed by two cost functions: one is related to the static obstacles while the other is related to dynamic obstacles. To do that, a simple obstacle prediction is performed, in order to know where the obstacles will be and then predict a collision-free motion. The main idea is then to run a motion planner (with a limited time budget) while the robot is already executing an action so that once the current motion terminates, the motion computed from the motion planner are executed. At this point, another instance of the motion planner is invoked and the interleaved procedure is repeated. Due to limited time budget, the planner may not be able to compute an optimal solution of the optimization function and the resulting trajectory may be, and usually is, sub-optimal. Obviously, the interleaving strategy is subject to the constraint that the current trajectory being executed cannot be modified, in order to avoid discontinuities in the trajectory the robot has to follow. One possible concern of this paper regards the computational time, since just spherical obstacles are considered during the simulations. Spherical obstacles are really simple objects

and their predictions are computational negligible. Moreover, planning result is guaranteed to be safe only during a short time period (due to the rough trajectory prediction performed by the authors). Finally, the sub-optimal solution of the motion planner may not be collision-free or may violate some other constraints during the next execution interval. The authors alleviates this problem by increasing the weights associated with the obstacles. However, it is important to underline that if the optimization result is valid but not optimal, the planner can also improve it incrementally during following time intervals. Moreover, the results shown in this paper are really good in terms of performance. Unlike this paper, we want to show that a real-time solution may be achieved using closed-form solutions.

The authors in [121] proposed a framework that, similarly to [94], faces the real-time motion planning as an optimization problem. The idea here is to reject any form of precomputed motion plan and create, at each cycle, new trajectories that the robot might follow. Multiple trajectories (called population in their terminology) are maintained at the same time, in order to be able to react also to drastic changes of the environment. A fitness function is associated to each trajectory, in order to evaluate its feasibility and optimality (the optimization criteria is defined as a combination of time, energy and manipulability associated to a trajectory). In other words, they propose a framework that alternates planning and execution of trajectories, that the robot has to follow. One of the advantages in this approach is that the robot does not have to wait for a trajectory having a high fitting value. It will execute the trajectory with the highest fitting function and, then, in future times, it will automatically switch to a trajectory having better fitting values. In few words, the algorithm starts by creating a population (both randomly and/or based on previous plans). To deal with moving obstacles, the planner predicts their future motions. Since the planner needs just the future position for the next few samples, the obstacle model does not need to be accurate for long times. In fact, a simple linear interpolator of two or more samples is used for this purpose. At each cycle, the planner first predicts where the obstacles will be in the next iteration. Then a operator is selected in order to modify the actual trajectories. The selection is performed within a set of primitives, creating a new set of trajectories. Then, they are evaluated (w.r.t. the optimization criteria) and inserted in the population. Just the trajectory with the highest fitting function is maintained for the next iteration, in order to maintain the diversity preservation. Even if the authors of [121] did not mention this explicitly, the algorithm sounds like a genetic algorithm, for the concept of population, fitness and modification operators (that stands for crossover or mutation operator). In addition, also the idea of preserving the diversity is common in a genetic algorithm. The real good aspect of this work relies in the computational time, that is in order of few milliseconds. However, this approach is different w.r.t. the one we propose in this chapter for two main reasons. First, the work in [121] is proposed for free-flying system. Second, there is no guarantee that the robot will avoid the obstacle.

The most complete and formal approach regarding the real-time motion planning is in [93]. It

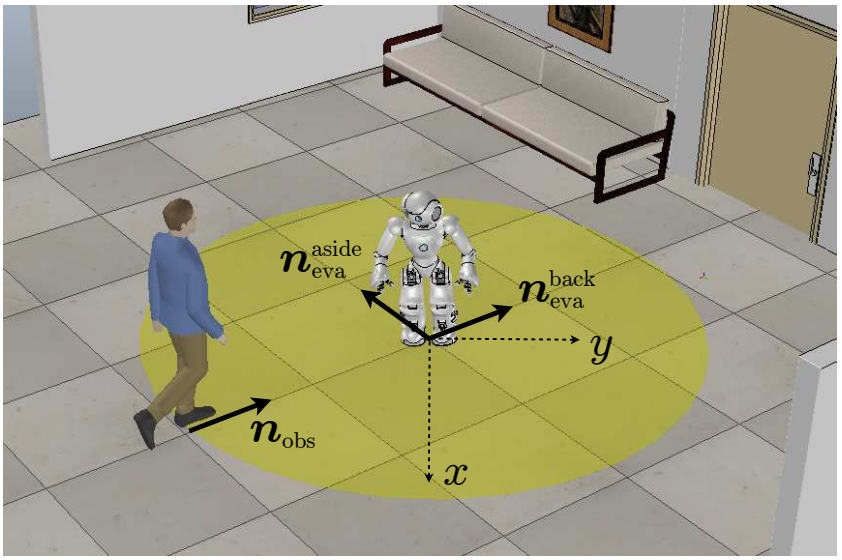


Figure 7.1 The situation of interest. A moving obstacle enters the safety area (yellow circle) of a humanoid and heads towards it. The humanoid must plan and execute a fast evasive motion. The figure also shows the moving frame associated to the humanoid.

is the first asymptotically optimal sampling-based motion planning algorithm for real-time navigation in dynamic environments. The authors propose a framework for real-time planning/replanning whose goal is to minimize the path length. The framework is proved to be asymptotically optimal in static environments. A graph is built in the configuration space, growing from the goal configuration. Again, the assumptions are the knowledge of the poses of static obstacles and the ability of predicting the poses of dynamic obstacles. The procedure is similar to the RRT*algorithm, since the algorithm is ensured to be asymptotically optimal thanks to the rewire procedure of RRT*. This framework is developed for mobile robots and the results are shown for a $2D$ scenario but its extension to the humanoid case is not trivial in my opinion. However, it is really remarkable the mathematical derivation of the approach, very elegant and formal through the paper.

This chapter is organized as follows. In Section 7.1 we formally define the reactive problem we want to solve. Section 7.2 outlines our solution approach, which mainly hinges on two conceptual blocks. The first is described in Section 7.3 and it generates an appropriate evasive maneuver in the form of a timed sequence of footsteps. The second is analysed in Section 7.4 and its aim is to compute a bounded CoM trajectory associated to these footsteps. Simulations and experiments that validate the proposed framework are shown in Section 7.5. Finally, an adaptation of the basic method for use in a replanning framework is discussed in Section 7.6.

7.1 Problem formulation

The problem that we want to solve in this chapter is completely summarized in Figure 7.1. A humanoid robot is standing in a workspace. At some point, a moving obstacle (e.g., a human)

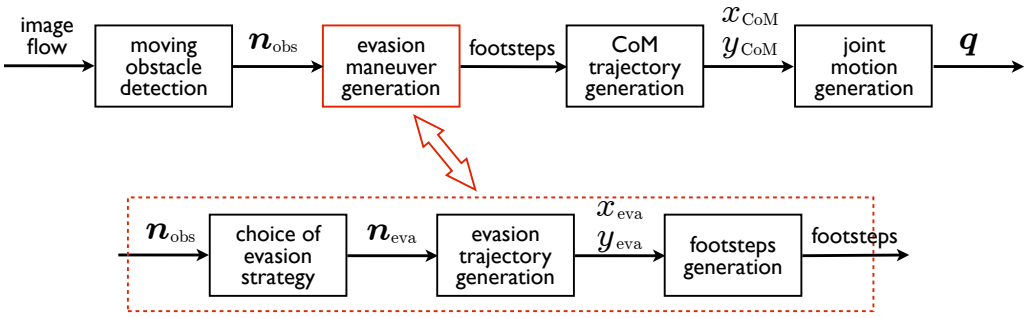


Figure 7.2 A block scheme of the proposed approach for planning and executing evasive motions.

enters its safety area and moves toward its direction. The robot must plan and execute as soon as possible an evasive maneuver in order to avoid the collision with the incoming obstacle.

We make the following assumptions for this problem. First, the robot is not executing any particular task or, in case, it is ready to abort it and focus on the evasive motion only. To this aim, safety should have higher priority with respect to any task the robot is executing. Otherwise, the result is a collision with the obstacle. Second, the speed of the incoming obstacle is slow enough to allow an evasive maneuver. This is obvious since, even assuming a very small reaction time, the robot needs time to execute an evasive motion (that also depends on the robot actuators). Third, the obstacle is pointing the robot and its direction does not change. This is the worst case scenario since the problem would be easier in any other direction case. Finally, the robot can freely move inside the safety area for performing the evasive motion.

Even if we intend to remove these assumptions in the future (for example, the hypothesis of constant obstacle direction is relaxed in Section 7.6), the above situation may already be of interest in practical situations. As example, straight lines are used by humans for moving to one point to another (in order to minimize his/her energy consumption). It is of interest, as for the on-going research project[2] mentioned above, to consider humans as the moving obstacles in view of safe coexistence between humans and humanoid robots. Then, the assumption about the constant direction of the incoming obstacle is not so far from a realistic application. Obviously, it is not the general case but it is a really good and realistic scenario.

7.2 Proposed approach

Our approach follows several successive steps

1. the entrance of the incoming obstacle is detected and its direction $\boldsymbol{n}_{\text{obs}}$ with respect to the robot is computed;

2. an evasive maneuver is generated, in order to react to the incoming obstacle. This module gives as output the footsteps the robot has to follow, by means of the following procedure

[2] See www.comanoid.eu

- one of the two proposed evasion strategy is chosen (see below);
- a trajectory for the evasive motion is determined;
- the footsteps are placed around the trajectory found at the previous step;

3. a suitable trajectory for the CoM of the humanoid robot is computed. This trajectory is obtained by the ZMP that can be easily computed if one knows the footsteps;
4. joint motions are generated so as to track such CoM trajectory, and used to actually move the humanoid, which we assume to be position-controlled.

A block scheme that describes these steps is represented in Figure 7.2. In this chapter, we will focus on the second and third blocks of Figure 7.2. For the first block, we assume that the robot is equipped with an exteroceptive sensor that perceives the environment. To this purpose, we have used a depth-camera mounted on the robot head. A simple extraction of the closest point can be used, in order to detect if an obstacle enters the safety area. In a similar way, we shall not dwell on the structure of the last block; we assume that joint motion is straightforwardly generated from the planned CoM trajectory by Jacobian-based kinematic control, as the one presented in Section 5.2.2. The only difference between the joint motion generation we need here and the one in Section 5.2.2 relies in the task constraint ($\boldsymbol{y}$), that should be not included in the augmented task vector (now it will be composed only by the CoM trajectory and by the swing foot trajectory). The reader is then referred to Section 5.2.2 for further details about the joint motion generation. Then, Section 7.3 is devoted to the description of the second block of Figure 7.2 (evasion maneuver generation) while Section 7.4 analyses the third block of Figure 7.2 (CoM trajectory generation).

We empathize that all the computations in each block should be performed as fast as possible. The reason is obvious: the smaller the reaction time between stimulus (detection of the moving obstacle) and action (beginning of the evasive motion), the more likely is the robot to perform successful collision avoidance. In view of this requirement, we shall look for closed-form solutions and expressions at all stages.

7.3 Evasion maneuver generation

The Evasion Maneuver Generation block receives as input a unit vector $\boldsymbol{n}_{\mathrm{obs}}$ representing the direction of an incoming obstacle and produces as output a sequence of footsteps that implements an appropriate evasion maneuver. This is achieved by three conceptual steps, that are shown in Figure 7.2

1. an evasion direction is chosen $\boldsymbol{n}_{\mathrm{eva}}$;
2. a planar trajectory that quickly aligns with this direction is generated as a reference for the humanoid;
3. footsteps are appropriately placed around this trajectory.

Each step is the subject of the following section.

7.3.1 Choice of evasion strategy

We propose two evasion strategies

- *move back*: the humanoid aligns with the direction of the moving obstacle and moves backwards;
- *move aside*: the humanoid aligns with the direction orthogonal to that of the moving obstacle and moves backwards.

First, it should be noted that both strategies dictate that the humanoid moves backwards. For the *move back* strategy, this is obvious since if the robot moves forward it moves in the direction of the obstacle. For the *move aside*, this requirement is related to the possibility of keeping the obstacle in view, as moving backwards allows to maintain the obstacle in the half-plane in front of the robot. In this way, it might monitor if there are changes in the direction of the obstacle and eventually react.

The aim of the *move back* strategy is to maximize the distance between the robot and the moving obstacle. This approach has been considered because it is, in human beings, the most instinctive reaction. This is related to the concept of approach aversion presented, e.g., in [48]. However, since the robots remains on the direction of the obstacle, this strategy is not sufficient to avoid a collision if the obstacle moves faster than the robot.

On the other side, the *move aside* embodies a different strategy. Its aim is to move, as fast as possible, away from the course of the obstacle. If executed sufficiently fast, this strategy may allow to avoid obstacles that are faster than the robot. Note that the distance to the obstacle may also decrease, at the first stages, and then increase again.

In practice, the *move back* strategy is realized by setting $\boldsymbol{n}_{\mathrm{eva}}^{\mathrm{back}} = \boldsymbol{n}_{\mathrm{obs}}$. On the contrary, the *move aside* strategy is obtained by setting $\boldsymbol{n}_{\mathrm{eva}}^{\mathrm{aside}} = \boldsymbol{n}_{\mathrm{obs}}^{\perp}$, where $\boldsymbol{n}_{\mathrm{obs}}^{\perp}$ is the normal unit vector to $\boldsymbol{n}_{\mathrm{obs}}$ in the half-plane behind the robot. Both strategy directions are shown in Figure 7.1.

7.3.2 Evasion trajectory generation

Once $\boldsymbol{n}_{\mathrm{eva}}$ has been chosen, the next steps consists in computing a planar trajectory that implements this strategy. We use a controlled unicycle model as reference model for this section. This model was described in Section 1.2.1 and the reader is referred to that section for further details. Indeed, this choice is enforced from experimental studies on human locomotion [85, 117], that have identified a gait model in which the orientation of the body is for most of the time tangent to the path. In other words, human trajectories closely resemble those typical of nonholonomic wheeled mobile robots, such as the unicycle. This kind of viewpoint was already effectively assumed in [33].

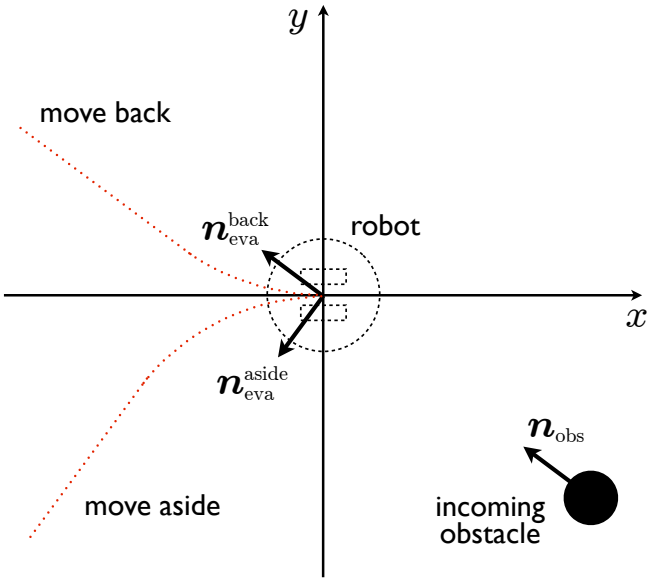


Figure 7.3 Generation of the evasion trajectory by means of a controlled unicycle as reference model.

Consider Figure 7.3. The reference unicycle is initially placed at the origin of the humanoid frame, with the same heading, and obeys the model in eq. (1.2) that we report here for completeness

$$\dot{x} = v \sin\theta \tag{7.1}$$

$$\dot{y} = v \cos\theta \tag{7.2}$$

$$\dot{\theta} = \omega \,, \tag{7.3}$$

where x, y are the unicycle Cartesian coordinates, θ is its orientation w.r.t. the x axis, and v, ω are the driving and steering velocity inputs. Let $\boldsymbol{n}_{\text{eva}}$ be the unit vector of the generic evasion direction, and denote with $\angle \boldsymbol{n}_{\text{eva}} \in [0, 2\pi)$ its phase angle. A unicycle traveling *backwards* in the direction of $\boldsymbol{n}_{\text{eva}}$ would have orientation $\theta_{\text{eva}} = \angle \boldsymbol{n}_{\text{eva}} - \pi$.

We propose a control law that aligns the unicycle with the desired orientation θ_{eva} while traveling at a constant velocity $\bar{v}$, typically chosen to realize the evasion maneuver as fast as possible. In formula, the proposed control law is

$$v = \bar{v} \tag{7.4}$$

$$\omega = k \ \text{sign}(\theta_{\text{eva}} - \theta), \tag{7.5}$$

where $\bar{v} < 0$ (since it is a backward movement) is a constant negative driving velocity and k is a positive constant. The resulting trajectory is an arc of circle that starts at the origin of the plane (as in Figure 7.3), having a radius equal to $|\bar{v}|/k$. The desired orientation θ_{eva} is achieved at the finite time instant $t_s = |\theta_{\text{eva}}|/k$. After t_s, the trajectory becomes a line, as shown in Figure 7.3.

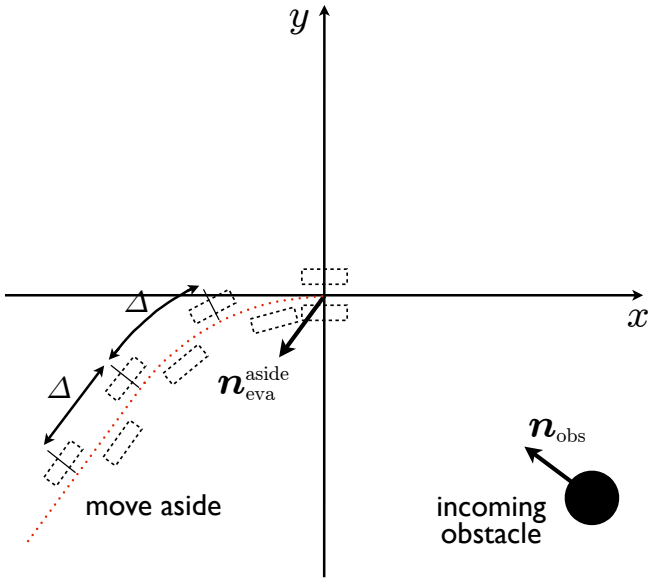


Figure 7.4 Footstep placement around the evasion unicycle trajectory. In the figure, a step aside maneuver is considered for illustration.

The fundamental aspect is that the integration of the model equations (7.1–7.3) under the control law (7.4–7.5) readily provides a closed form for such trajectory, i.e.,

$$x(t) = \bar{v}\,\frac{\sin kt}{k} \tag{7.6}$$

$$y(t) = \mathrm{sign}(\theta_{\mathrm{eva}})\,\bar{v}\,\frac{1-\cos kt}{k} \tag{7.7}$$

$$\theta(t) = \mathrm{sign}(\theta_{\mathrm{eva}})\,kt \tag{7.8}$$

for $t \leq t_s$ and

$$x(t) = x(t_s) + \bar{v}(t-t_s)\cos\theta_{\mathrm{eva}} \tag{7.9}$$

$$y(t) = y(t_s) + \bar{v}(t-t_s)\sin\theta_{\mathrm{eva}} \tag{7.10}$$

$$\theta(t) = \theta_{\mathrm{eva}} \tag{7.11}$$

The availability of a closed form expression for the evasion trajectory is of fundamental importance within our framework for two main reasons. First, it perfectly suites in a real-time framework since it is really fast to compute the unicycle trajectory given the desired orientation θ_{eva}. Second, it is important also in view of the adoption of a replanning scheme that will be presented in Section 7.6, since it allows to compute in real-time the trajectory in case the desired θ_{eva} angle changes.

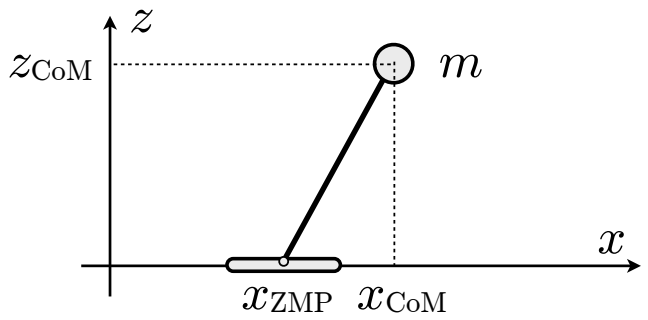


Figure 7.5 The Linear Inverted Pendulum (LIP) in the sagittal plane.

7.3.3 Footstep generation

Once the trajectory for the controlled unicycle is available, the last step for creating an evasive maneuver is to generate the footsteps for the humanoid robot. The basic idea is to sample the footsteps around the unicycle trajectory by using a constant stepsize Δ, as shown in Figure 7.4. This is obtained by evaluating the trajectory expressions (7.6–7.8) and (7.9–7.11) using a constant time interval $\Delta t = \Delta/|\bar{v}|$, and displacing the x, y, θ samples alternatively to the right and to the left of the trajectory. In details, it is obtained by a discretization of the expressions in eqs. (7.6–7.8) and eqs. (7.9–7.11) and then by computing

$$\begin{aligned} x_{r,i} &= x(t_i) + d\ sin\theta(t_i) \\ y_{r,i} &= y(t_i) - d\ cos\theta(t_i) \end{aligned}$$

where $x_{r,i}, y_{r,i}$ are the x, y coordinates where the (right) footstep has to be placed; $x(t_i), y(t_i), \theta(t_i)$ is the pose of the evasive trajectory (sampled from (7.6–7.8) and (7.9–7.11)) evaluated at time t_i (a generic time instant). Finally, d is half of the distance between the feet at rest and $t_i = i\ \Delta/\bar{v}$ is the time associated to the ith footstep, since we assumed a constant parametrization of the length and the speed. Similar formulas hold for the left foot case. The first step is an exception, because it is actually a half-step, i.e., its length is $\Delta/2$.

We assume that the inner foot is the first to move. In formula, if $\theta_{\mathrm{eva}} - \theta(t_i) > 0$, the the first foot that moves is the right, with $\theta(t_i)$ the starting orientation. Otherwise, it is the left foot that moves first.

7.4 CoM trajectory generation

The CoM trajectory generation block of Figure 7.2 is described in this section. Its input is the footsteps generate by the procedure described in Section 7.3 and gives as provides as output the CoM trajectory that the joint motion generation module has to fed.

As for the previous procedure, all the computations within this block should be performed as fast as possible. This neglects the usage of the full humanoid dynamics. We shall instead resort

to the Linear Inverted Pendulum (LIP, see Figure 7.5), a well-known approximate model that describes the motion of the humanoid CoM when its height is kept constant and no rotational effects are taken into account. The LIP model has already been used for real-time CoM trajectory generation, as in [55].

7.4.1 The Linear Inverted Pendulum (LIP) model

As mentioned before, the LIP model is a common approximation of the motion of the CoM of a humanoid robot. The lateral and sagittal motions of the CoM are completely decoupled and obey to identical linear differential equations. For this reason, we can analyse the sagittal plane motion and similar equations will hold for the lateral motion. For the sagittal plane, the dynamics of the LIP model are described in [9, 54] and they are equal to

$$\ddot{x}_{\mathrm{CoM}} = \frac{g}{z_{\mathrm{CoM}}}(x_{\mathrm{CoM}} - x_a) + \frac{1}{m z^*_{\mathrm{CoM}}}(\tau_a - \tau_h) + \frac{F(t)}{m} \ , \tag{7.12}$$

where x_{CoM} is the CoM position along the x-axis, g is the gravity acceleration constant, z^*_{CoM} is its constant height, x_a is point foot location, m is the total mass of the humanoid robot, τ_a is an ankle torque, τ_h is a hip torque acting on a reaction mass and $F(t)$ is disturbance force in the direction of the x-axis (that may vary over time). Denoting with $\eta = \sqrt{g/z_{CoM}}$, eq. (7.12) can be rewritten as

$$\ddot{x}_{\mathrm{CoM}}(t) = \eta^2 x_{\mathrm{CoM}}(t) - \eta^2 z(t) \ , \tag{7.13}$$

where $z(t)$ represents all external inputs to the dynamic equations of the CoM. The model in eq. (7.13) has been used for different gaits model. As example, [65] treats the following gait models

- point foot model. This model has a no torques applied and the pendulum is modelled as a massless telescoping leg with point foot location at x_a and a point mass m, kept at constant height z_{CoM}. The Center of Pressure (CoP) is fixed in x_a if no step is taken. For this reason, x_a is the control input and the model can be written as

$$\ddot{x}_{\mathrm{CoM}} = \eta^2 x_{\mathrm{CoM}} - \eta^2 x_a \ ; \tag{7.14}$$

- finite-size foot model. This model has a torque located at the ankle (τ_a) and the CoP location is related to the ankle torque through $x_{\mathrm{CoP}} = x_a - \tau_a/(mg)$. The model can be written as

$$\ddot{x}_{\mathrm{CoM}} = \eta^2 x_{\mathrm{CoM}} - \eta^2 x_{\mathrm{CoP}} \ ; \tag{7.15}$$

- reaction mass model. Here, the torso and arms represented by an actuated reaction mass that generates a momentum around the CoM and, as consequence, a torque around the x-axis. The rotational dynamics of the reaction mass are not included in the following

formula, since they are not important for the concept we want to assert here. Defining the Centroidal Moment Pivot (CMP) as

$$x_{\mathrm{CMP}} = x_{\mathrm{CoP}} + \frac{\tau_h}{mg} = x_a - \frac{\tau_a - \tau_h}{mg} \; ;$$

the model can be represented as

$$\ddot{x}_{\mathrm{CoM}} = \eta^2 x_{\mathrm{CoM}} - \eta^2 x_{\mathrm{CMP}} \; . \tag{7.16}$$

Since eqs. (7.14, 7.15, 7.16) are variations of eq. (7.13), we use the last as reference model.

7.4.2 Bounded CoM trajectory

This section follows the approach in [75], where a method for deriving a closed form solution for the CoM trajectory, in certain conditions, is presented.

Eq. (7.16) can be seen as a linear system with state space $\boldsymbol{x} = (x_{\mathrm{CoM}} \; \dot{x}_{\mathrm{CoM}})^T$, input $z(t)$ and

$$A_c = \begin{pmatrix} 0 & 1 \\ \eta^2 & 0 \end{pmatrix} \qquad B_c = \begin{pmatrix} 0 \\ -\eta^2 \end{pmatrix} \; . \tag{7.17}$$

The idea is to apply a change of coordinates proposed in [116]

$$\begin{pmatrix} x_u \\ x_s \end{pmatrix} = \begin{pmatrix} 1 & 1/\eta \\ 1 & -1/\eta \end{pmatrix} \begin{pmatrix} x_{\mathrm{CoM}} \\ \dot{x}_{\mathrm{CoM}} \end{pmatrix} \tag{7.18}$$

to the system represented in eq. (7.16), obtaining a decoupled system

$$\dot{x}_u = \eta \; x_u - \eta \; z \tag{7.19}$$

$$\dot{x}_s = -\eta \; x_s + \eta \; z \; , \tag{7.20}$$

where x_s (resp. x_u) represents the stable (resp. unstable) dynamics of the LIP model. The unstable system corresponds to the extrapolated CoM in [47], to the divergent component of motion in [31, 116] and to the capture point dynamics in [65].

The idea is now to select a trajectory for the CoM that remain bounded and avoid the divergent behavior associated to the unstable eigenvalue η. According to [75], this can be done either by a proper choice of $x_u(t_i)$ at the starting time instant t_i or by designing the input $z(t)$ so as to maintain $x_u(t)$ bounded.

Let focus on the unstable dynamics. Eq. (7.19) can be integrated for a generic input $z(t)$, obtaining

$$x_u(t) = e^{\eta(t-t_i)} x_u(t_i) - \eta \int_{t_i}^{t} e^{\eta(t-\tau)} z(\tau) d\tau \; . \tag{7.21}$$

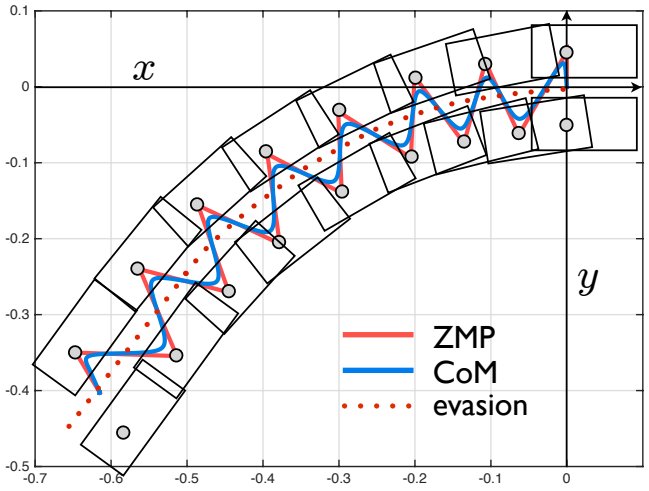


Figure 7.6 ZMP and CoM trajectory generation for an evasion maneuver: geometric path.

If the initial condition satisfies

$$x_u(t_i) = x_u^*(t_i) = \eta \int_{t_i}^{\infty} e^{-\eta\tau} z(t+\tau) d\tau \tag{7.22}$$

we obtain the following solution

$$x_u^*(t) = \eta \int_{t_i}^{\infty} e^{-\eta\tau} z(t+\tau) d\tau \ , \tag{7.23}$$

which is bounded under mild assumptions. We denote with $x_u^*(t)$ a trajectory for the system that satisfied eq. (7.22). Note that this solution depends on future value of z, then it is anticausal. This is a known fact (the CoM is anticausal w.r.t. the ZMP trajectory), e.g., in [54]. If one plugs the change of coordinates in eq. (7.18) into eq. (7.22), one obtains

$$x_u^*(t_i) = x_{\text{CoM}}(t_i) + \frac{1}{\eta} \dot{x}_{\text{CoM}}(t_i) \ , \tag{7.24}$$

that is known as the *boundedness constraint*.

Regarding the stable LIP dynamics, for any initial condition $x_s(t_i)$, the trajectory $x_s(t)$ will converge to a steady-state solution, if it exists. Finally, the CoM trajectory is computed since it is related to x_u^* and x_s^* through (using also eq. (7.18))

$$x_{\text{CoM}}^* = \frac{1}{2}(x_s^* + x_u^*) \qquad \dot{x}_{\text{CoM}}^* = \frac{\eta}{2}(x_u^* - x_s^*) \ . \tag{7.25}$$

The CoM trajectory is then obtained by plugging eqs. (7.21, 7.22, 7.23) into eq. (7.25)

$$x_{\text{CoM}}^*(t) = e^{-\eta t} x_{\text{CoM}}(t_i) + \frac{x_s(t) - e^{-\eta t} x_u(t_i) + x_u(t)}{2} \ , \tag{7.26}$$

and using as reference input $z(t) = x_{\text{ZMP}}^*$, i.e., the desired trajectory for the ZMP, that can

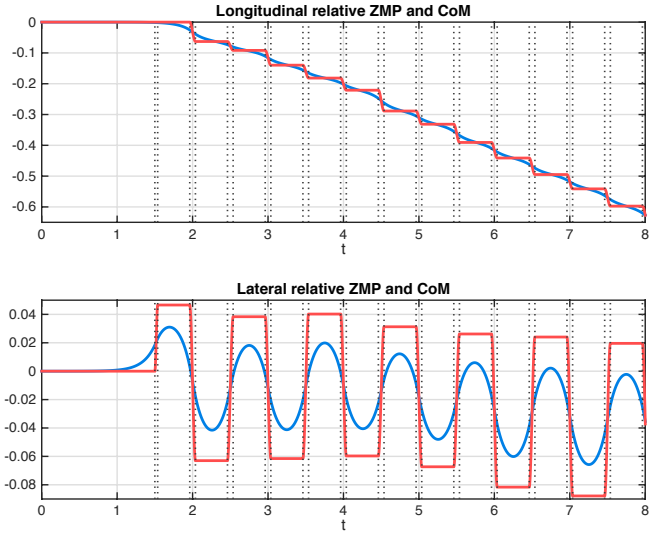


Figure 7.7 ZMP and CoM trajectory generation for an evasion maneuver: time evolution.

be easily computed given the timed footsteps, as shown in Figure 7.6. The reader is referred to [73, 74, 75] for further details about the CoM trajectory generation procedure described above.

Note that similar equations hold also for the lateral plane, defining y^*_{CoM}. Moreover, eq. (7.26) is valid for any ZMP trajectory. Typical choice of the ZMP trajectory (as ours) results in closed form integrable expression. As consequence, also eq. (7.26) is in closed form. In the context of evasive motions, being able to generate the CoM trajectory in a closed form expression and therefore in real-time is of a fundamental importance. Having a parametric analytical expression of the CoM trajectory for a desired ZMP not only allows fast computational schemes but also gives the necessary insight to solve replanning problems, as shown in Section 7.6. Some references about the ZMP can be found in [51, 52, 53, 54, 115, 122].

Once $\boldsymbol{z}^*_{\mathrm{CoM}}(t) = (x^*_{\mathrm{CoM}}(t)\ y^*_{\mathrm{CoM}}(t)\ z^*_{\mathrm{CoM}})^T$ has been computed, it can be used as a reference trajectory for joint motion generation. While it is $x^*_{\mathrm{CoM}}(t_i) = x_{\mathrm{CoM}}(t_i)$ by construction, the initial reference velocity

$$\dot{x}^*_{\mathrm{CoM}}(t_i) = \eta\,(x_{\mathrm{u}}(t_i) - x_{\mathrm{CoM}}(t_i)) \tag{7.27}$$

will not match the humanoid initial condition in general. This reflects to a transient error for the CoM and for the ZMP. In other words, if the humanoid starts with unmatched initial velocity $\dot{x}_{\mathrm{CoM}}(t_i) \neq \dot{x}^*_{\mathrm{CoM}}(t_i)$, a transient error in the CoM trajectory will be present and this will generate consequently an error in the ZMP which, if possible, should be avoided in order to minimize the risk of tilting. One way to avoid this error (and the associated risk) is to perform an *anticipative* motion aimed at achieving the correct initial velocity before actually starting to track $x^*_{\mathrm{CoM}}(t)$.

Typical result of the proposed CoM trajectory generation procedure is shown in Figure 7.6 and Figure 7.7. Here, the reference ZMP trajectory was built by choosing line segments to interpolate the evasion footsteps, then dividing each step into a double and single support phase, and

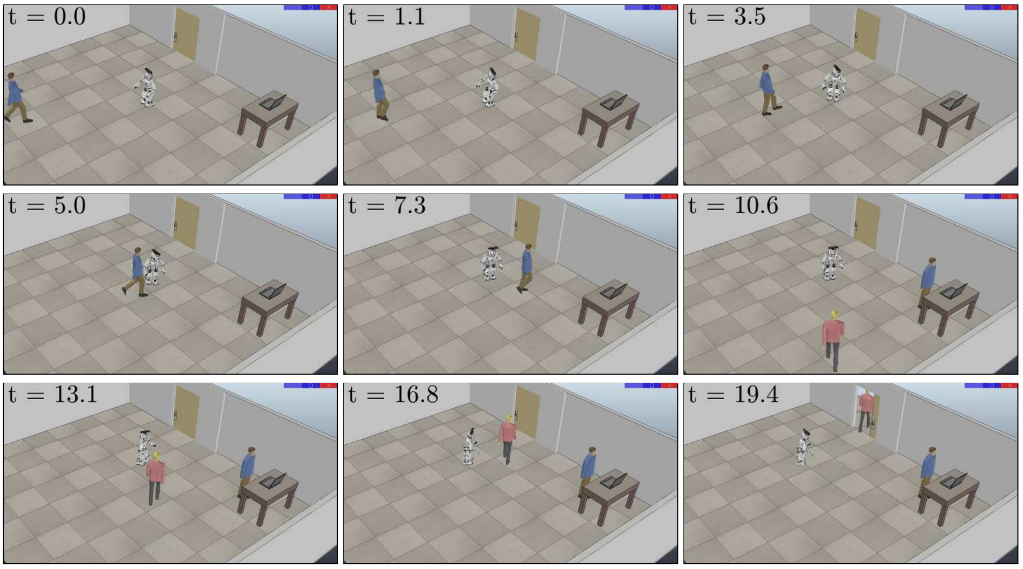


Figure 7.8 Evasive motions using the *move aside* strategy: snapshots from a simulation.

choosing the ZMP to be cubic in time over the first and constant over the second.

Finally, $\boldsymbol{z}^*_{\mathrm{CoM}}(t) = (x^*_{\mathrm{CoM}}(t)\ y^*_{\mathrm{CoM}}(t)\ z^*_{\mathrm{CoM}})^T$ is given as input to the joint motion generation, together with trajectories for the swing foot $\boldsymbol{z}_{\mathrm{swg}}(t)$ in the same time interval. These trajectories can be easily computed since the footsteps are known from Section 7.3.3[3].

The joint motion generation is very close to the scheme in Section 5.2.2. The only difference is in the augmented task vector defined as $\boldsymbol{y}_a = \left(\boldsymbol{z}^T_{\mathrm{swg}}\ \boldsymbol{z}^T_{\mathrm{CoM}}\right)^T$. In other words, there is no task for the specific point of the humanoid robot, as in Section 5.2.2. The reader is then referred to that section for additional details about the joint motion generation.

7.5 Simulations and experiments

We have validated the proposed framework for the NAO robot, introduced in Section 4.4.1. We have equipped the robot with a depth camera (Asus Xtion PRO Live[4]) mounted on its head. Using this sensor, it can detect obstacles that enter its safety area, defined as a circle of radius 1.5 m, and compute the unit vector $\boldsymbol{n}_{\mathrm{obs}}$ that represents the obstacle approach direction. In details, we computed the closest obstacle point $\boldsymbol{c}_p$ and derived the obstacle angle as $\theta_o = \mathrm{atan2}(\boldsymbol{c}_{\mathrm{p,y}}, \boldsymbol{c}_{\mathrm{p,x}})$. The approaching angle is then $\angle\boldsymbol{n}_{\mathrm{obs}} = \theta_o + \theta_h$, where θ_h is the yaw angle of the head (since the depth camera is rigidly attached to the robot head), available from the joints readings.

The proposed method was first tested in simulation, using V-REP (Section 4.4.2)[5]. Snapshots from a typical simulation are depicted in Figure 7.8. The humanoid is standing at the center of a room when a human (acting as a moving obstacle) walks in (first snapshot), directed

[3]The support and swing foot are uniquely determined by the footsteps. Then, appropriate trajectories can be easily computed since the poses of the swing foot at the start and end of each iteration are known.

[4]https://www.asus.com/us/Multimedia/Xtion_PRO_LIVE

[5]For further details, please visit `www.dis.uniroma1.it/~labrob/research/HumanoidEvasion.html`, where there is also a video of the proposed approach.

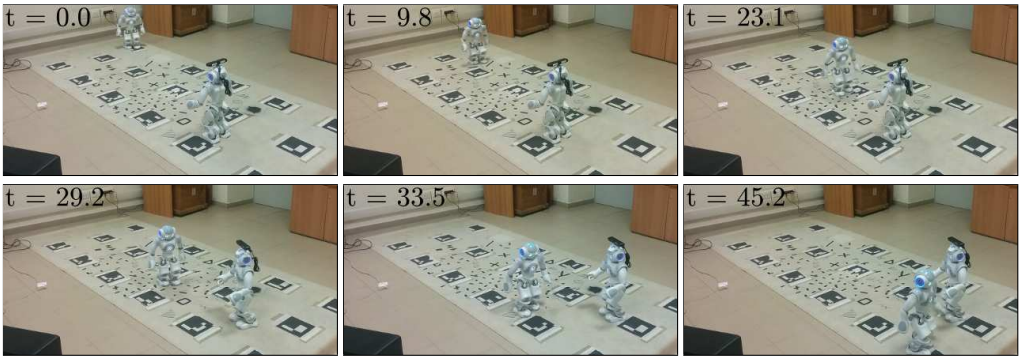


Figure 7.9 Evasive motion using the *move aside* strategy: snapshots from an experiment.

towards the desk on the right. The humanoid detects that the human enters its safety area (set to 1.75 m) in the second snapshot and it starts the evasion maneuver using the *move aside* strategy. Following the scheme depicted in Figure 7.2, an evasion trajectory is first generated using the controlled unicycle model, with $\bar{v} = 0.04$ m/s, $k = 0.2$ and $d = 0.1$ m, and footsteps are placed around this trajectory, using a stepsize $\Delta = 0.08$ m, in the range of NAO capabilities. Then, a ZMP trajectory interpolating the footsteps is computed, with duration of the double and single support respectively at 0.122 s and 0.425 s, and the corresponding bounded CoM trajectory is generated. Finally, joint commands are computed via a joint motion generation. Overall, the achieved response time (between the detection of the moving obstacle entering the safety area and availability of the first joint command) is around 21 ms, confirming that the use of closed-form expressions in all stages of the motion generation makes real-time evasion possible. The result is that the humanoid aligns with the direction orthogonal to the moving obstacle's one (third snapshot) following an arc of circle and then moves backward, by performing 8 backward steps (fifth snapshot). This allows the human to safely reach the desk. The simulation continues with another human crossing the room (sixth snapshot) towards the automatic door. The robot executes another successful *move aside* evasion maneuver (seventh to ninth snapshot).

Experimental validation of the proposed framework has been performed with two NAOs. The first robot is teleoperated and acts as a moving obstacle. The other (the one with the camera mounted on its head) runs the proposed evasive algorithm. Snapshots from a typical experiment are shown in Figure 7.9. The moving obstacle starts walking (second snapshot), entering the safety area of the other robot (third snapshot). As for the simulation, the robot starts an evasive maneuver using the *move aside* strategy. The result is that the robot first aligns its orientation with the direction orthogonal to the moving obstacle (fourth snapshot), by following an arc of circle. Then, it moves back, by performing 6 backward steps (fifth snapshot). In this way, a the evasive maneuver is successfully executed.

We should mention that the particular platform used for the simulation/experiment limits our framework. In fact, NAO has limiting gait capabilities (e.g., its size does not allow a realistic experiment with a real human having longer legs). So, we had to limit the velocity imposed to the moving obstacle; otherwise, the robot would have no opportunity to avoid it. Using a robot

having human-like gait capabilities, our framework can be even more effective and deal with real human actors.

7.6 Replanning

Assume that the moving obstacle changes its orientation and it is *malicious*, in the sense that it continuously changes its course so as to keep aiming at the humanoid while the latter performs the evasive motion. Depending on the obstacle's own motion model, this may result in different approach trajectories, more or less aggressive. In this case, the humanoid cannot continue to execute the evasive motion dictated by the first approaching direction of the obstacle but it has to update its motion. The mechanism for achieving this *replanning* is already embedded in the proposed scheme, and in particular in the unicycle feedback control law (7.4–7.5) that generates the evasion trajectory.

In this section, we do not want to propose a complete study of the evasion trajectories resulting from the interaction of the obstacle approach trajectory with the controlled unicycle model. On the other side, we want to show how our framework may deal with the replanning problem, as a preliminary step in that direction.

7.6.1 Replanning the evasion maneuver

The change of direction in the motion of the moving obstacle can be detected from the same exteroceptive sensor that detected its entrance into the humanoid safety area. Once a new value $\boldsymbol{n}'_{\mathrm{obs}}$ is available, it is immediately used to compute a new $\boldsymbol{n}'_{\mathrm{eva}}$, the associated evasion trajectory given by (7.6–7.8) and (7.9–7.11) with θ'_{eva} in place of θ_{eva}. The corresponding footsteps are then computed. It is essential to underline that these computations are extremely fast, thanks to the closed-form expression. Obviously, the updated trajectory and footsteps start from the current posture of the humanoid.

7.6.2 Replanning the CoM trajectory

Once the new set of footsteps is available, the ZMP and the CoM trajectories should be computed as fast as possible. We assume that the robot has completed (or has the time to complete) the double support phase and it is ready to perform a step in a single support phase, in order to follow the new sequence of footsteps. The difficulty is that the CoM trajectory associated to the current maneuver will be different from that corresponding to the new maneuver. In the light of Section 7.4, this reflects in a discontinuity in the CoM initial velocity at the time of switching and, consequently, in a transient error both on the CoM and the resulting ZMP. Moreover, we cannot perform an anticipative motion to match the initial velocity, as in Section 7.4, since the robot is already moving.

We propose to solve this issue by introducing some free parameters in the desired ZMP trajectory $x^*_{\mathrm{ZMP}}(t)$. The idea is to use these parameters to satisfy the boundedness constraint (7.24)

through $x_{\mathrm{u}}(t_i)$, which depends on $x^*_{\mathrm{ZMP}}(t)$. This leaves $x_{\mathrm{CoM}}(t_i)$ and $\dot{x}_{\mathrm{CoM}}(t_i)$ free. This approach can be seen as a simultaneous ZMP-CoM generation, as in [86, 89].

Assuming a point foot model (i.e., no double support phase) for simplicity, the ZMP is defined as a sequence of Heaviside functions $u_{\mathrm{step}}(t)$, with amplitudes α_k equal to the step lengths. For n steps, the reference ZMP trajectory $x^*_{\mathrm{ZMP}}(t)$ becomes

$$x^*_{\mathrm{ZMP}}(t) = \sum_{k}^{n} \alpha_k u_{\mathrm{step}}(t - t_k) \ , \tag{7.28}$$

that brings to

$$x_{\mathrm{u}}(t_i) = \sum_{k}^{n} \alpha_k e^{-\eta t_k} \ . \tag{7.29}$$

Considering the presence of free parameters in the first step only, the boundedness constraint can be written as

$$\alpha_1 e^{-\eta t_1} = -\sum_{k=2}^{n} \alpha_k e^{-\eta t_k} + x_{\mathrm{CoM}}(t_i) + \frac{1}{\eta}\dot{x}_{\mathrm{CoM}}(t_i) \ .$$

From the previous equation, it is possible to choose as design parameter either the step length α_1 or its duration t_1. The corresponding solutions are in the first case

$$\alpha_1 = e^{\eta t_1} \left(x_{\mathrm{CoM}}(t_i) + \frac{\dot{x}_{\mathrm{CoM}}(t_i)}{\eta} - \sum_{k=2}^{n} \alpha_k e^{-\eta t_k} \right) \ ,$$

and in the second

$$t_1 = -\frac{1}{\eta} \log \frac{1}{\alpha_1} \left(x_{\mathrm{CoM}}(t_i) + \frac{\dot{x}_{\mathrm{CoM}}(t_i)}{\eta} - \sum_{k=2}^{n} \alpha_k e^{-\eta t_k} \right) \ .$$

An example of the replanning algorithm is in Figure 7.10. The CoM trajectory in the top plot is obtained with an approach similar to the one shown in Section 7.4, i.e., by interpolating a sequence of footsteps from a ZMP trajectory with fixed timing and lengths. Assume that $t = 0$ is the time of switching and that the actual CoM velocity is -0.08 m/s, from a previous plan. The nominal CoM velocity (obtained from the boundedness constraint) is $\dot{x}^*_{\mathrm{CoM}}(0) = -0.108$ m/s. It is obvious that there is a mismatch between the two CoM velocities and that there would be a transient error in tracking the nominal CoM trajectory. As alternative, a matched CoM trajectory is computed by introducing a parameter in the ZMP trajectory and, in particular, in the first step, as shown in this section. As we mentioned above, two parameters can be maintained as in the nominal plan: the duration or the length of the first step. In the first case, a longer step length is obtained (as in the center plot of Figure 7.10. In the second case, this results in a shorter step is obtained (as in the bottom plot of Figure 7.10). These results are consistent with the observation that the actual CoM velocity is larger than needed.

This example makes clear that choosing as free parameter the first step length actually leads to modify the footstep sequence (at least for the first step) with respect to the updated sequence

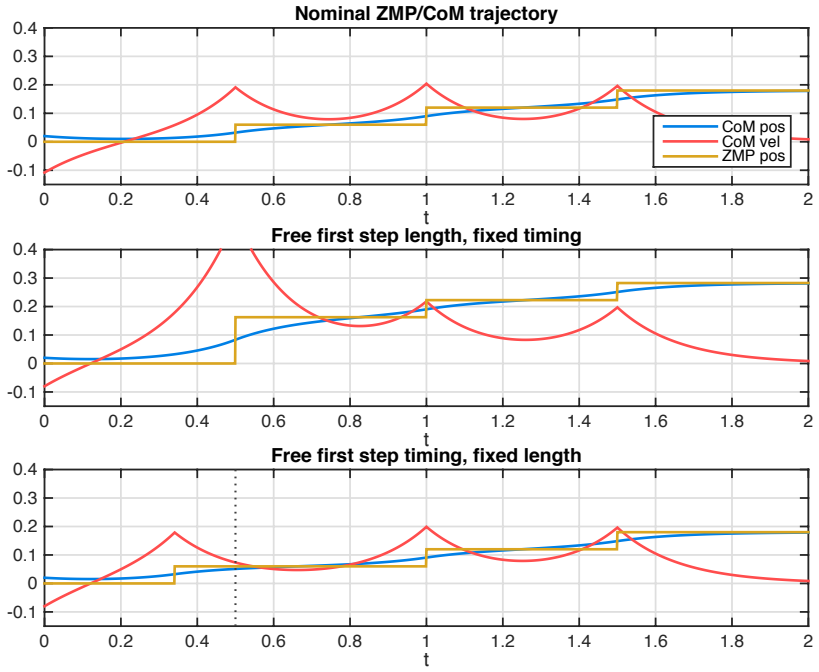


Figure 7.10 A replanning example. (top) The nominal CoM trajectory corresponding to an updated evasion maneuver is mismatched w.r.t. the CoM initial velocity. (center) A matched CoM trajectory computed by freeing the first step length. (bottom) A matched CoM trajectory computed by freeing the first step duration.

coming from the evasion maneuver generation block described in Section 7.3. Consider now the central plot in Figure 7.10. It is clear that the variable length approach may lead to high CoM velocities. For this reason, it appears that the variable duration approach is the most convenient choice for replanning the CoM trajectory without modifying the updated evasion maneuver.

7.7 Conclusions

In this chapter, we have described an approach for real-time planning and execution of evasive motions of a humanoid robot. The proposed method follows some conceptual steps, depicted in Figure 7.2. First, a moving obstacle is detected if it enters the safety area of the humanoid robot. The approaching direction of the obstacle is then determined. On the base of this information, an evasive trajectory is generated by means of footsteps. In particular, we propose two evasive strategies (move back and move aside) from which a reference evasion trajectory is computed for a unicycle reference model. The footsteps are then placed around this trajectory. Once the footsteps are known, an appropriate trajectory is computed for the CoM of the humanoid. Finally, joint motion commands are generated so as to track such trajectory. We showed that all these computations can be performed in real-time, thanks to the closed-form expressions of the two central blocks of Figure 7.2. The proposed approach has been successfully validated via simulations and experiments on a NAO humanoid. The possibility of adapting the basic method so as to be used in a replanning framework has also been discussed.

In the next future, we want to develop a *gazing* strategy for faster detection of obstacles entering the robot safety area and we want to design and test of additional evasion strategies in addition to the basic ones considered in this chapter. Finally, the replanning scheme should be analysed in more details also for the case of finite-sized feet.

The framework presented in this chapter has been accepted in [20][6] and it will be presented at the *2016 IEEE Int. Conference on Robotics and Automation, Stockholm, Sweden, 16–21 May 2016*.

[6]For further details, please visit `www.dis.uniroma1.it/~labrob/research/HumanoidEvasion.html`, where there is also a video of the proposed approach.

III

NONPREHENSILE REARRANGEMENT PLANNING

Chapter 8

Nonprehensile rearrangement planning using object-centric and robot-centric action spaces

This chapter focuses on the problem of manipulation tasks, requiring the robot to interact with the environment. This kind of tasks has received increasing emphasis by the robotic community over the years. The reason is simple: if the robot is able to interact with the environment, it can be used for performing (repetitive) tasks that might be dangerous for humans or for tasks in collaboration with humans. Commonly, robots (e.g., manipulators) rely solely on the ability of picking and placing objects through grasping actions. Humans use a much more diverse suite of actions to fulfil everyday tasks. As example, consider grabbing an inner object in a cluttered pantry. In this case, picking and placing each object in front the coveted one is not the best option. On the contrary, one may push aside items using the elbow, forearm and the back of the hand, while placing the coveted object in the palm of the hand and, finally, grasping it. In order to perform such an action, nonprehensile interactions such as pushing or pulling should be included even in the fulfilment of basic tasks. For this reason, we will develop techniques for generating whole-body nonprehensile interactions to solve the rearrangement planning problem by means of open-loop planners. In this problem, a robot should find a feasible trajectory in order to achieve a goal. This trajectory allows the robot to physically interact with multiple objects in the environment. As example, consider Figure 8.1. In this case, the robot has to move the green parallelepiped into the goal region depicted with a green circle by means of pushing actions. To this aim, the robot should interact with the objects in the environment, not only with the goal object. In fact, as it is evident from the right part of Figure 8.1, the purple bottle and one of the blue box are moved in order to fulfil the task.

We propose to solve the rearrangement planning problem at the planning stage. Our planners have to be able to simultaneously allow object interaction and manipulation.

This problem is challenging for a number of reasons. First, the integration of nonprehensile interactions is not a trivial problem. These kinds of actions were already been used for

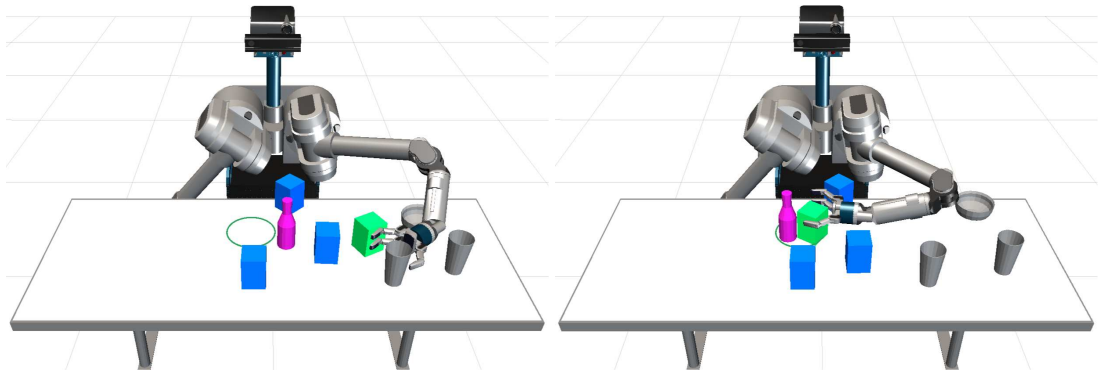


Figure 8.1 Example of the rearrangement planning problem. The robot should move the green box to the goal region denoted with a green circle. Doing that, it has to perform nonprehensile interactions with the green box but also with the other objects in the environment. (left) Initial planning problem. (right) A final solution found by our planner.

pre-grasp manipulation [13, 62], large object manipulation [29] and simultaneous object interaction [38, 63]. Moreover, they can be used for performing manipulation task for robots not designed for this scope. As example, a mobile robot may perform pushing actions and manipulate with the environment. The reader should then be convinced by the importance of introducing nonprehensile interactions. Unfortunately, it is not possible to reasoning about these interactions via geometric planners since an object does not move rigidly with the robot. On the contrary, an object subjected to a nonprehensile action evolves under non-holonomic constraint that represent the physics of the environment and the contact between the robot and the object. Some approaches face this problem by introducing some simplifications about the interaction (the objects move rigidly attached to the robot) or the objects (disc robot pushing disc objects) [1, 2]. On the other hand, we believe that the right way to proceed is to embed physical models directly inside the motion planners, as in [45, 63]. Doing that, we do not need any assumption about the robot nor the objects in the environment.

Another challenge relies in the dimension of the planning problem we want to solve. As mentioned before, an interaction modifies the planning environment. This means that we have to track all the poses of the objects that might move within the state of our motion planner. This leads to a state space that is linearly depended on the number of movable objects. In other words, the planning problem is defined in an high-dimensional space. The authors in [45, 63], on which this chapter is highly inspired, propose to alleviate this problem by projecting actions into a lower dimensional physics manifold. In fact, even if the contact is essential for this problem, there are large portions of the state space that does not involve a contact (e.g., the left part of Figure 8.1). It is shown that it is possible to project all actions to a lower dimensional manifold where contact is likely to occur. This speeds planning by focusing our search to regions of state space we know to be critical to goal achievement. Another way to reduce the high-dimensional problem is to consider only dynamic actions that result in statically stable states. Including a physics engine into the planning stage allows to model physical interaction between the robot and objects the environment. However, it forces to consider a state space composed both by the poses and the

velocities of the robot and the objects that might move in the environment (since a physics engine works at the second order, a state should include the zero and the first order). Unfortunately, this might have a negative impact on the planning time. The authors in [45, 63] observed that the absence of any external forces but the gravity causes a manipulated object to eventually come to rest due to friction. For this reason, the same authors showed that choosing dynamic actions that result in statically stable states (e.g., the environment comes to rest after each action), the planning problem can be analysed in a lower dimensional space that ignores the velocities.

From a research point of view, the rearrangement planning problem have been studied in several works. The work in [123] proves that this problem in NP-hard while in its first formulation it was denoted as Navigation Among Movable Obstacles (i.e., NAMO). Here, the task of the robot is to navigate from a start to a goal configuration and in the environment there are objects that it can move (i.e., movable obstacles). This problem was also further extended to the manipulation task case, initially focusing on pick and place tasks [87, 113]. The main difference with the work presented in this chapter relies in the fact that those works are limited to grasp actions. Some works [4, 27, 28] showed the advantages of equipping the robot with nonprehensile capabilities, allowing the robot to solve scenes where the object to be manipulated is too large or too heavy to be grasped.

Some authors tried to face the rearrangement problem as a classical planning problem. As example, the approach in [106] proposes two types of actions: (i) transit (the robot moves without being in contact with any object) and (ii) transfer (the robot changes the state of one or more objects). Another technique used especially for its speed are the Random Exploring Random Tree (RRT) [78]. Here, edges represent those actions while nodes represent state of the system. Only transfer actions are considered in [87] while the work in [2] extends this approach including nonprehensile actions. Another kind of approaches for dealing with the high-dimensionality of the problem is to decompose it in subproblems, as in [14]. The approach that will be described in this chapter is a state-space approach (RRT). However, instead of using motion primitives as in [3, 87] that forces a designer to develop effective and efficient actions, we believe that simple actions are able to solve the rearrangement planning problem.

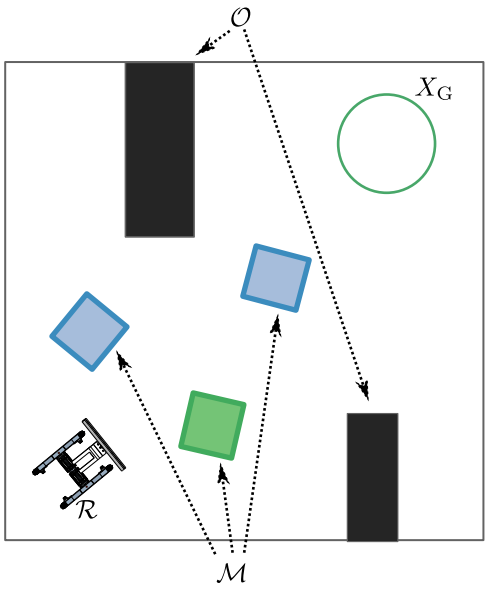
Another method for facing the problem is the backchaining. This method is inspired by the concept of goal pre-images [81, 84]. Within this framework, a graph is built where the nodes represent states and edges represent actions, as mentioned before. A pre-image of a state is the set of all the states (and also the actions) that converges to the given state. This idea can be applied also for solving the rearrangement planning problem, for which a planner starts from the goal. The idea is to search for actions that move a set of objects such that the final state corresponds to the desired state. This idea is the core of the work in [113] in case of transfer actions only. As mentioned before, the works in [27, 28] extended this framework including also nonprehensile actions such as pushing. At the end, the idea of backchaining is to reduce the problem to a list of object to move, limiting the ability to have simultaneous object interaction.

An alternative approach is to track the reachable free space, that is the workspace reachable from the robot when no obstacle is within its workspace. The idea is to place the robot and the goal in the component of the free space. This idea is particularly helpful when the number of movable objects is large, since searching in a space composed by the robot and the movable obstacles states becomes difficult as the number of movable obstacles increases. The works in [112, 118, 119] explore this idea in a hierarchical framework. A high-level planner is in charge of connecting disjoint regions while a low-level planner provides a path for the robot inside a single free space component.

The aim of this chapter is to extend the approaches in [45, 63] (on which it is highly inspired) including also object-centric and robot-centric action spaces. In particular, robot-centric actions move the robot without object relevant intent (as the transit action described above) and object-centric actions are aimed for interacting with objects in the environment (as the transfer action described above). These actions easily allow simultaneous objects contacts and whole arm interaction.

The robot-centric primitives were used in [45, 63]. This kind of actions has the advantage of allowing simultaneous contacts and that are independent w.r.t. the number of movable obstacles. On the other side, in case the planner uses only these actions, the resulting planner often suffers from long plan times due to the lack of goal directed motions available to the planner. This is the reason for which we introduce the object-centric actions. They are often exposed to the planner in the form of user-defined high-level primitives. They can be highly effective, allowing the planner to perform large advancements toward the goal. On the other side, using only this kind of actions (as in [2, 28, 50, 112, 113]) limits the types of solutions that can be obtained by the planner. In fact, they involve the contact between an object and a single part of the robot, i.e., its end-effector. This neglects the opportunity to have simultaneous contacts between the robot and two or more objects. Note that our idea is to use simple object-centric actions, in such a way we do have to embed accurate or very precise primitives in the planner.

We then formulate a hybrid planner that uses both action types. Our idea is based on the fact humans use a diverse set of actions when interacting with the world. While many of these actions are object-centric, focusing on interacting in a specific way with a single object, other interactions are purely coincidental. These interactions are the unplanned result of a motion with different intent. In other words, our insight is that both types of actions are critical to generating expressive solutions quickly. By integrating the two actions types, we can use the freedom of interaction fundamental to the robot-centric actions while still allowing for the goal oriented growth central to the object-centric methods. Consider the example of reaching for a milk jug in the back of a refrigerator. One might first carefully slide the juice jug out of the way, then simply reach for the milk, and trust that other objects that are touched will naturally be pushed out of the way, without requiring specific actions to move them. The idea is to create a planner that is able to solve this kind of problems.



$\mathcal{R}$	Robot
$\mathcal{C}^R$	Robot C-space
$\mathcal{M}$	Movable objects
$\mathcal{C}^i$	Movable i C-space
$\mathcal{O}$	Obstacles
$\boldsymbol{X}_{\text{free}}$	Free state space
$\mathcal{U}$	Control space
$\boldsymbol{X}_{\mathrm{G}}$	Goal region

Figure 8.2 The planning environment.

This chapter is organized as follows. Section 8.1 formally introduces the rearrangement planning problem while our approach for solving this problem is presented in Section 8.2. Finally, some planning experiments are proposed in Section 8.3 while Section 8.4 ends the chapter with some considerations about this chapter.

8.1 The rearrangement planning problem

In this section, we provide a formal description of the rearrangement planning problem. Consider Figure 8.2, where the problem is formulated for a mobile robot. In a bounded world $\mathcal{W}$, we assume to have a robot $\mathcal{R}$ and its configuration space $\mathcal{C}^R$. In $\mathcal{W}$, there are a set $\mathcal{M}$ of m objects that the robot is able to move (movable objects) and a set $\mathcal{O}$ composed by obstacles that the robot cannot move (static obstacles), i.e., a contact with one object in $\mathcal{O}$ is not allowed. Each movable object has its own configuration space, denoted with $\mathcal{C}^i$, as depicted in Figure 8.2.

Our state space is composed by the Cartesian product space of the robot and the movable obstacle states as $\boldsymbol{X} = \mathcal{C}^R \times \mathcal{C}^1 \times \cdots \times \mathcal{C}^m$. A state $\boldsymbol{x} \in \boldsymbol{X}$ is then defined as $\boldsymbol{x} = \left(\boldsymbol{q}, \boldsymbol{o}^1, \ldots, \boldsymbol{o}^m\right)$, with $\boldsymbol{q} \in \mathcal{C}^R$ a configuration assumed by the robot and $\boldsymbol{o}^i \in \mathcal{C}^i \; \forall i = 1, \ldots, m$ a configuration of the ith movable obstacle. We define the free space $\boldsymbol{X}_{\text{free}} \subseteq \boldsymbol{X}$ as the set of all states in $\boldsymbol{X}$ where the robot in not penetrating any movable or static obstacle. It is important to underline that the previous definition of $\boldsymbol{X}_{\text{free}}$ allows contacts between entities as limit case[1] and this is fundamental for manipulation tasks.

The goal of the rearrangement planning problem is to find a feasible robot trajectory $\boldsymbol{\xi} : \mathbb{R}^{\geq 0} \rightarrow \boldsymbol{X}_{\text{free}}$ starting from a given state $\boldsymbol{\xi}(t_i) \in \boldsymbol{X}_{\text{free}}$ and ending in a goal region $\boldsymbol{\xi}(T) \in \boldsymbol{X}_{\mathrm{G}} \subseteq \boldsymbol{X}_{\text{free}}$, at some time $T \geq 0$.

[1]Here, we use the term penetration to indicate two objects that not only are in contact but are colliding. Two objects that are in contact are not penetrating.

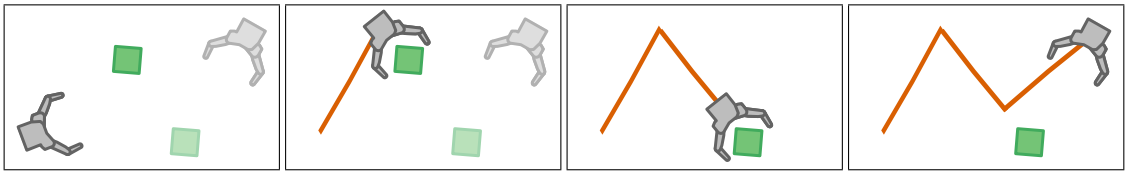


Figure 8.3 An example of solving the rearrangement planning problem with just one movable object. The dark is the initial state while the light is the end state. In the second figure, the robot moves into place to push the object while in the third figure it applies a pushing action in order to set the object in the goal pose. Finally, the robot moves in its goal pose.

Having formally introduced the rearrangement planning problem, we can turn our attention to the state dynamics. The state $\boldsymbol{x}$ evolves nonlinearly based on the physics of the manipulation, i.e. the motion of the objects is governed by the contact between the objects and the manipulator. This is expressed by the following non-holonomic constraint

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) \ , \tag{8.1}$$

where $\boldsymbol{u} \in \mathcal{U}$ is a control input that can be instantaneously applied to the robot $\mathcal{R}$. A path $\boldsymbol{\xi}$ is feasible if there exists a control $\boldsymbol{u} \in \mathcal{U}$ such that $\dot{\boldsymbol{\xi}}(t) = \boldsymbol{f}(\boldsymbol{\xi}(t), \boldsymbol{u})$, i.e., the constraint in eq. (8.1) is satisfied at all times while the robot is following $\boldsymbol{\xi}$.

It is important to underline that the constraint in eq. (8.1) is really hard to model. In fact, it has to propagate the state in case of contact between objects. This is the reason for which we will include within our planner a physics engine, such as ODE or BOX2D (see Section 8.2), to have an high accuracy propagation in the state space.

8.2 Planner overview

Our planner is based on the Random Exploring Random Tree (RRT) [78]. Traditional implementations of the algorithm solve the two-point boundary value problem (BVP) during tree extension. Since, as explained in Section 8.1, we have to plan in a state space composed by the robot state space and the movable objects state space, the two-point BVP problem is as hard as solving the full problem. Note also that a movable object is not directly controllable but its motion is the result of a pushing action performed by the robot. Therefore, solving the two-point BVP to connect $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \boldsymbol{X}_{\text{free}}$ requires finding a path for the robot that moves each object in $\mathcal{M}$ from its position in $\boldsymbol{x}_1$ to its position in $\boldsymbol{x}_2$. To convince the reader, consider the example where there is just one movable object, as in Figure 8.3. The start state is then defined as $\boldsymbol{x}_1 = (\boldsymbol{q}_1, \boldsymbol{o}_1)$ and the goal state as $\boldsymbol{x}_2 = (\boldsymbol{q}_2, \boldsymbol{o}_2)$. For solving this two-point BVP, the robot should first move to a location near $\boldsymbol{o}_1$, then push the object in its goal location $\boldsymbol{o}_2$ and finally move to its goal pose $\boldsymbol{q}_2$. All the actions performed by the robot must be collision-free and then feasible. It is evident from this example that the movable obstacle is able to move just from an action taken from the robot.

The remaining part of this section is organized as follows. Section 8.2.1 and Section 8.2.2 show useful tools needed for each version of the algorithm. The aim of Section 8.2.3 and Section 8.2.4 is to describe how previous works have handled this constraint. Finally, our planning approach is depicted in Section 8.2.5.

8.2.1 Configuration sampling

When using a probabilistic planner such as a RRT, it is essential to be able to sample a random state. In principle, one might sample from any distribution, as long as densely sampling from the space $\boldsymbol{X}_{\text{free}}$ is guaranteed. In practise, we uniformly sample the robot and all the movable objects from a uniform distribution. We use rejection sampling to ensure the sampled configuration is valid, i.e., discarding any sampled states that have object-object or manipulator-object penetration.

8.2.2 Distance metric

The distance is another key point for a probabilistic planner. The authors in [2] claim that the correct distance metric between two states $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ is the length of the shortest path traveled by the robot that moves each movable object from its configuration in $\boldsymbol{x}_1$ to its configuration in $\boldsymbol{x}_2$. However, computing such a distance (or even an approximation) is very difficult. For this reason, we have used a weighted Euclidean distance metric defined as

$$\texttt{Dist}(\boldsymbol{x}_1, \boldsymbol{x}_2) = w_q \|\boldsymbol{q}_1 - \boldsymbol{q}_2\| + \sum_{i=1}^{m} w_i \|\boldsymbol{o}_1^i - \boldsymbol{o}_2^i\| \ ,$$

where $\boldsymbol{x}_1 = (\boldsymbol{q}_1, \boldsymbol{o}_1^1, \dots, \boldsymbol{o}_1^m)^T$, $\boldsymbol{x}_2 = (\boldsymbol{q}_2, \boldsymbol{o}_2^1, \dots, \boldsymbol{o}_2^m)^T$, $w_q \in \mathbb{R}$ is the weight associated to the robot and $w_i \in \mathbb{R}, \forall i \in [1, \dots, m]$ is the weight associated to the ith movable object.

8.2.3 Robot-centric planner using random action sampling

As suggested in [77], a useful alternative to solving the two-point BVP is to use a discrete time approximation to eq. (8.1) to forward propagate all controls and select the best using a distance metric defined on the state space. In particular, define an action set $\mathcal{A} : \mathcal{U} \times \mathbb{R}^{\geq 0}$ where $\boldsymbol{a} = (\boldsymbol{u}, d) \in \mathcal{A}$, where $\boldsymbol{u}$ describes a control and d its associated duration for which one wants to apply the control. Then, a transition function, $\Gamma : \boldsymbol{X} \times \mathcal{A} \to \boldsymbol{X}$, is used to approximate the non-holonomic constraint.

Note that our control space $\mathcal{U}$ is continuous and this neglects the opportunity to enumerate all the action set. Instead, an approximation of this space is performed by taking k actions, forward propagating each under Γ and selecting the best from this discrete set.

Algorithm 7 describes a basic implementation of this algorithm. It builds a tree $\mathcal{T}$ in the configuration space until the constraint imposed by the goal region is reached. At each iteration, it samples a random configuration (Algorithm 7, line 3), as described in Section 8.2.1. Next,

Algorithm 7: Kinodynamic RRT with random action sampling and physics model propagation

```
1  T ← {nodes = {x_0 = x(t_i)}, edges = ∅};
2  while not ContainsGoal(T) do
3      x_rand ← SampleConfiguration();
4      x_near ← Nearest(T, x_rand);
5      for i = 1 ... k do
6          (u_i, d_i) ← SampleUniformAction();
7          (x_i, d_i) ← PhysicsPropagate(x_near, u_i, d_i);
8      end
9      i* = argmin_i Dist(x_i, x_rand) if Valid((x_near, x_i*), u_i*, d_i*)) then
10         T.nodes ∪ {x_i*};
11         T.edges ∪ {((x_near, x_i*), u_i*, d_i*)};
12     end
13 end
14 path ← ExtractPath(T);
```

the nearest configuration is retrieved (Algorithm 7, line 4) using the distance metric provided in Section 8.2.2. At this point, the best action for the robot is chosen using robot-centric actions (Algorithm 7, lines 5–8). In details, k samples are taken from a uniform distribution in the action space. For the generic ith action, a physical propagation is performed starting from $\boldsymbol{x}_{\text{near}}$ and applying the control $\boldsymbol{u}_i$ for a duration d_i. The result is a new state $\boldsymbol{x}_i$. Within this thesis, the physics propagation is seen as a black box that takes as input a state and an action and gives as output the new state. Additional details about the physics propagation can be found in [45, 63]. This is attractive because it allows complex interactions like multi-object pushing and whole arm manipulation to evolve naturally.

The best action $(\boldsymbol{u}_{i^*}, d_{i^*})$ is selected as the one whose ending state $\boldsymbol{x}_i$ (the state obtained from $\boldsymbol{x}_{\text{near}}$ applying a physics propagation using the action $(\boldsymbol{u}_{i^*}, d_{i^*})$) is the one that has the smallest distance, within the k actions, to the random state $\boldsymbol{x}_{\text{rand}}$ (first argument in Algorithm 7, line 9). The distance is again computed as in Section 8.2.2. If the path connecting $\boldsymbol{x}_{\text{near}}$ and $\boldsymbol{x}_{i^*}$ is valid, $\boldsymbol{x}_{i^*}$ is inserted in the tree $\mathcal{T}$ as a node while the path as an edge (Algorithm 7, lines 9–12). Finally, a path is extracted by backtracking (last line in Algorithm 7).

As it is evident from Algorithm 7, it relies on robot-centric actions. The drawback is its lack of focused tree growth. In particular, during each extension, the tree is not strongly pulled toward the sampled configuration. In fact, the tree is expanded by taking the best action from a set of random actions. In Figure 8.4 we show the distance between the end of an extension and the sampled state for several extensions of a tree grown to solve the scene from Figure 8.8. It is evident that the tree makes very small progress toward the target state using robot-centric actions, even using a large number of control samples (k in Algorithm 7). This is especially true if this data is compared to the progress performed by the object-centric actions, shown in Figure 8.4.

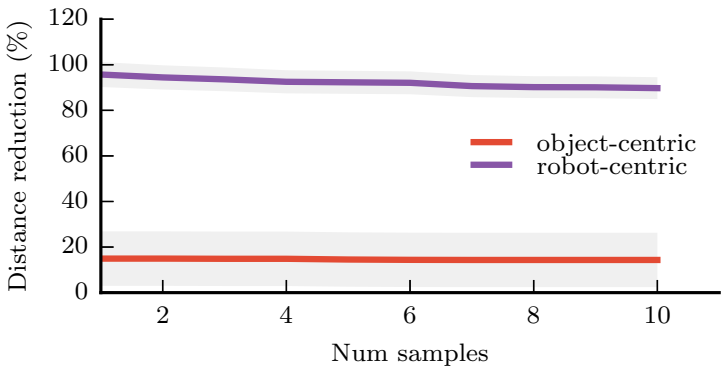


Figure 8.4 The distance between the achieved state and target state. As can be seen, sampling object-centric actions leads to significant improvement in reaching the target state.

Algorithm 8: Kinodynamic RRT using motion primitives

$\mathcal{T} \leftarrow \{\texttt{nodes} = \{\boldsymbol{x}_0 = \boldsymbol{x}(t_i)\}, \texttt{edges} = \emptyset\}$;
while *not* `ContainsGoal(`$\mathcal{T}$`)` **do**
  $\boldsymbol{x}_{\text{rand}} \leftarrow$ `SampleConfiguration();`
  $\boldsymbol{x}_{\text{near}} \leftarrow$ `Nearest(`$\mathcal{T}$`,` $\boldsymbol{x}_{\text{rand}}$`);`
  $(\boldsymbol{a}_1, \dots, \boldsymbol{a}_j) \leftarrow$ `GetPrimitiveSequence();`
  $\boldsymbol{x}_{\text{new}} \leftarrow$ `PrimitivePropagate(`$\boldsymbol{x}_{\text{near}}, (\boldsymbol{a}_1, \dots, \boldsymbol{a}_j)$`);`
  if `Valid(`$(\boldsymbol{x}_{\text{near}}, \boldsymbol{x}_{\text{new}}), (\boldsymbol{a}_1, \dots, \boldsymbol{a}_j)$`)` **then**
    $\mathcal{T}$`.nodes` $\cup \{\boldsymbol{x}_{\text{new}}\}$;
    $\mathcal{T}$`.edges` $\cup \{((\boldsymbol{x}_{\text{near}}, \boldsymbol{x}_{\text{new}}), (\boldsymbol{a}_1, \dots, \boldsymbol{a}_j))\}$;
  end
end
path $\leftarrow$ `ExtractPath(`$\mathcal{T}$`);`

The result is that the tree must randomly "stumble" on a goal rather than intentionally growing in that direction. This often leads to higher than desired plan times.

8.2.4 Object-centric planner using high-level actions

Another way for approaching the action problem is to use a set of object-centric primitives capable of solving the two-point BVP in a lower dimensional subspace. This approach was already used in [2, 50]. As example, consider the case of a push-object action. In this case, the primitive is composed by a set of actions that moves an object from its current to a desired configuration, as in Figure 8.3.

As explained above, object-centric actions have the advantage of allowing large extension of the tree, as depicted in Figure 8.4. Then, their introduction in the planner is interesting since it allows to reduce the planning time. Moreover, in [2], a clever sampling method is implored that ensures the primitives can be used to fully solve the two- point BVP. Thus, in the absence of obstacles, every extension will achieve the sample point. This is particularly useful when the sample is a goal state: it allows the tree to grow to the goal and, then, to find a solution for the rearrangement planning problem.

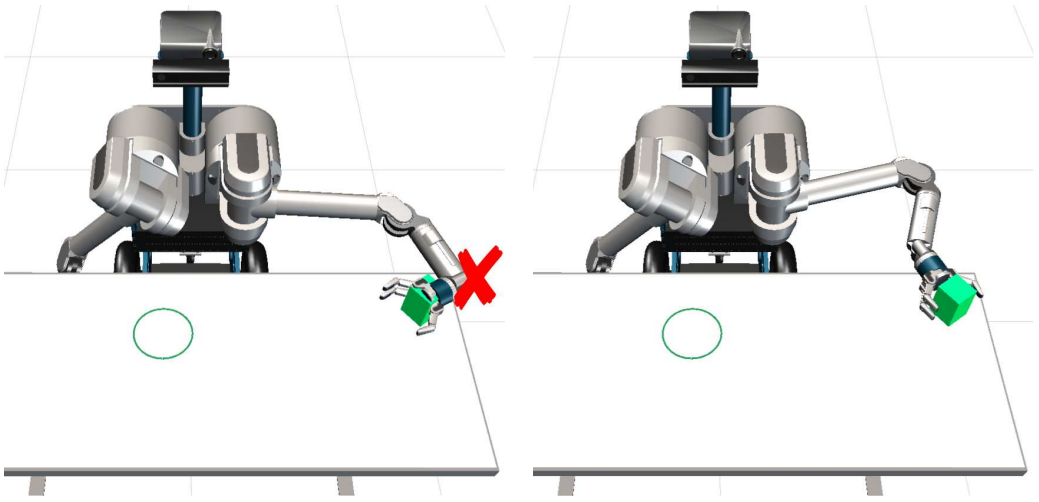


Figure 8.5 An example of object-centric primitives and their susceptibility to failure. (left) failed "push-object" primitive. The desired end-effector pose is not reachable when approaching the target object (green box). (right) An alternative primitive that is able to grasp and push the target object in the desired goal region (green circle).

On the other side, they have three main drawbacks. First, these actions limit the types of solutions generated by the planner. In particular, they usually involve contacts between the robot and one single selected object. This neglects the opportunity to have multiple contacts when applying an action (as it may happen using a robot-centric action). In fact, the `PrimitivePropagate` function in Algorithm 8 explicitly forbids contact with any other movable object but the selected one. Second, they might be difficult to design. Regarding the last problem, we just used really simple primitives, as described in Section 8.2.5. Last but not least, applying this kind of actions may result into a failure if the primitive cannot be successfully applied. As example, consider Figure 8.5. Here, an object-centric primitive can be described by two components. First, the robot arm moves close to the target movable object (the green box in Figure 8.5), with the palm of the hand in the direction of the desired pose (the goal region denoted with a green circle in Figure 8.5). Then, the robot pushes the target object along the direction of the desired push. Note that the object is really close to the edge of the reachable workspace of the robot. The result is that all the attempts in applying this primitive fail, as shown in the left of Figure 8.5. The reason is straightforward: when the robot tries to reach the point close to the target object, it reveals in a violation of the reachable workspace. Even more problematic, a solution to the scene cannot be found given the current action space. To generate a feasible solution, the programmer must define alternative primitives, as in the right of Figure 8.5. This example should convince the reader that object-centric primitives might be highly effective (in case of the alternative primitive in the right of Figure 8.5 it allows to reach the goal region) but they transfer to the programmer the responsibility to design an effective and general primitives, increasing the difficulty for the programmer. However, this problem is particularly true in case just object-centric actions are in the planner. As we will see in Section 8.2.5, our insight will be to provide to the planner both robot-centric and (simple) object-centric actions in such a way, even if a object-centric primitive

Algorithm 9: Kinodynamic RRT using hybrid action sampling

1 $\mathcal{T} \leftarrow \{\texttt{nodes} = \{\boldsymbol{x}_0 = \boldsymbol{x}(t_i)\}, \texttt{edges} = \emptyset\}$;
2 **while** *not* `ContainsGoal(`$\mathcal{T}$`)` **do**
3 $\quad \boldsymbol{x}_{\text{rand}} \leftarrow$ `SampleConfiguration();`
4 $\quad \boldsymbol{x}_{\text{near}} \leftarrow$ `Nearest(`$\mathcal{T}$`,` $\boldsymbol{x}_{\text{rand}}$`);`
5 $\quad$ **for** $i = 1 \dots k$ **do**
6 $\quad\quad r \leftarrow$ `Uniform01();`
7 $\quad\quad$ **if** $r < p_{rand}$ **then**
8 $\quad\quad\quad \boldsymbol{A}_i \leftarrow$ `SampleUniformAction();`
9 $\quad\quad$ **else**
10 $\quad\quad\quad \boldsymbol{A}_i \leftarrow$ `SamplePrimitiveSequence();`
11 $\quad\quad$ **end**
12 $\quad\quad (\boldsymbol{x}_i, \boldsymbol{A}_i) \leftarrow$ `PhysicsPropagate(`$\boldsymbol{x}_{\text{near}}, \boldsymbol{A}_i$`);`
13 $\quad$ **end**
14 $\quad i^* = \text{argmin}_i$ `Dist(`$\boldsymbol{x}_i, \boldsymbol{x}_{\text{rand}}$`);`
15 $\quad$ **if** `Valid(`$(\boldsymbol{x}_{\text{near}}, x_{i^*}), \boldsymbol{A}_{i^*}$`)` **then**
16 $\quad\quad \mathcal{T}$`.nodes` $\cup \{\boldsymbol{x}_{i^*}\}$;
17 $\quad\quad \mathcal{T}$`.edges` $\cup \{((\boldsymbol{x}_{\text{near}}, \boldsymbol{x}_{i^*}), \boldsymbol{A}_{i^*})\}$;
18 $\quad$ **end**
19 **end**
20 $path \leftarrow$ `ExtractPath(`$\mathcal{T}$`);`

fails in an iteration of the algorithm, it is not stuck in applying the same primitive over and over again.

A basic implementation of this approach is given in Algorithm 8. As the approach in Section 8.2.3, it builds a tree $\mathcal{T}$ in the configuration space until the constraint imposed by the goal region is reached. At each iteration, it samples a random configuration (Algorithm 8, line 3), as described in Section 8.2.1. Next, the nearest configuration is retrieved (Algorithm 8, line 4) using the distance metric provided in Section 8.2.2. Then, the sequence of action imposed by a object-centric primitive is selected. In practise, we select with a uniform probability which primitive to use within an iteration. This is denoted with the sequence of actions $\boldsymbol{A} = (\boldsymbol{a}_1, \dots, \boldsymbol{a}_j)$ in Algorithm 8. As example, $j = 3$ in the primitive shown in Figure 8.3. The selected primitive is then propagated (Algorithm 8, line 6) giving as output the new configuration $\boldsymbol{x}_{\text{new}}$. In case the path connecting $\boldsymbol{x}_{\text{near}}$ and $\boldsymbol{x}_{\text{new}}$ is valid, $\boldsymbol{x}_{\text{new}}$ is added as a node and the path is added as an edge of the tree $\mathcal{T}$ (Algorithm 8, lines 7–10). The above-mentioned procedure is repeated until the tree $\mathcal{T}$ does not contain a node in the goal region, i.e., the target object is in the goal region.

8.2.5 Hybrid planner

From Section 8.2.3 and Section 8.2.4, it should be clear that both robot-centric and object-centric actions have their advantages and disadvantages. Just for completeness, here we recall them. Robot-centric actions are useful because they do not depend on a specific object and they allow multi-contact between the robot and multiple movable objects. On the other side, they lack in

making progress toward a selected state. Object-centric actions allows large progresses but they may bring to a failure and they can be very difficult to design. Furthermore, they do not allow multi-contact when applying them.

For these reasons, we propose a method that allows for the freedom of interaction fundamental to the robot-centric methods while still allowing for the goal oriented growth central to the object-centric methods, i.e. we combine the two planner presented in Section 8.2.3 and Section 8.2.4.

Algorithm 9 depicts the proposed approach. A tree $\mathcal{T}$ is built in the configuration space (composed by the robot pose and the movable object poses) and the starting state $\boldsymbol{x}_0 = \boldsymbol{x}(t_i)$ is supposed to be given and inserted as the root of the tree. At each iteration, a random configuration is extracted (Algorithm 9, line 3) and the nearest configuration is computed on the basis of the distance metric defined in Section 8.2.2. Then, as in the method described in Section 8.2.3, the best of k possible actions is selected. However, each candidate extension i expresses a sequence of actions, $\boldsymbol{A}_i = (\boldsymbol{a}_1, \dots, \boldsymbol{a}_j)$, with j the number of actions in the primitive ($j = 3$ in Figure 8.3). This process is described in Algorithm 9, lines 5–14. To this aim, we select with probability p_{rand} a single action $\boldsymbol{a} = (\boldsymbol{u}, d)$ drawn uniformly at random from the space of feasible actions (robot-centric action as in Section 8.2.3). Otherwise, $\boldsymbol{A}_i$ contains a sequence of actions, $(\boldsymbol{a}_1 \dots \boldsymbol{a}_j)$, that are equivalent to the object-centric primitive described in Section 8.2.4 with noises applied to the primitive parameters. As example, the "push-object" primitive shown in Figure 8.5 is parameterized by the start point of the push, and the distance of the push and we use this primitive within our algorithm. Note that, even this fails in that situation, it is not a problem for our framework, since the algorithm is able to escape from the stuck situation by using a robot-centric primitive or another object-centric primitive. As counterpart, the simple design of such object-centric primitive allows to reduce the responsibility on the programmer.

In any case, the selected action is physically propagated using a physics engine. This propagation removes the problem of the object-centric primitives shown in Section 8.2.4. In fact, it is possible now to perform multi-contact between the robot and multiple objects by means of the physics engine. In other words, any unintended contact with other objects in the scene can be modeled now. Note that this unintended contact is not detrimental to overall goal achievement and it should be allowed. The result of the k physics propagations is a set of new states $\{\boldsymbol{x}_i\}, \forall i \in [1, \dots, k]$. Within this set, the state having the smallest distance to the random state $\boldsymbol{x}_{\text{rand}}$ is the one selected for validation ($\boldsymbol{x}_{i^*}$). Again, we use the distance function defined in Section 8.2.2. Finally, path between $\boldsymbol{x}_{\text{near}}$ and $\boldsymbol{x}_{i^*}$ is checked for collisions. In a positive case, $\boldsymbol{x}_{i^*}$ is added as a node and the path connecting $\boldsymbol{x}_{i^*}$ and $\boldsymbol{x}_{i^*}$ as an edge of the tree $\mathcal{T}$ (Algorithm 9, lines 15–18). Otherwise, no node is added to $\mathcal{T}$ and the process is repeated. When a node in $\mathcal{T}$ satisfied the goal region (e.g., the target object pose is within the goal region). Note that we have decided to validate just the state closest state (w.r.t. $\boldsymbol{x}_{\text{rand}}$) since the path connecting two states should be checked for collisions, that is a computationally expansive operation.

This method is attractive because it combines the strengths of the methods described in Section 8.2.3 and Section 8.2.4. Moreover, sampling random actions with some probability allows

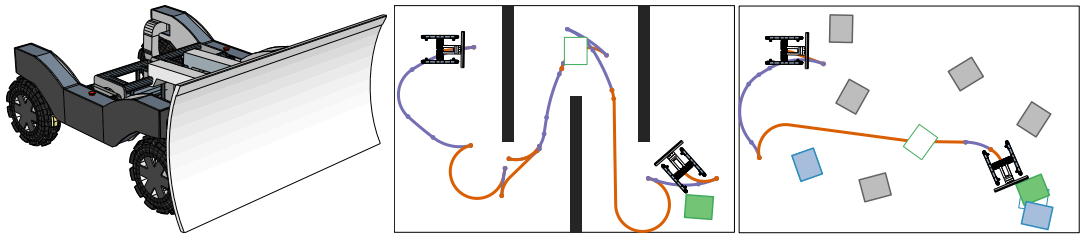


Figure 8.6 Rearrangement planning examples for the KRex robot. In each scenario, the robot must push the green box from its start pose to the goal region indicated by the green circle. Robot-centric (purple) actions are used to move the robot away from obstacles and into configurations where object-centric (orange) primitives can be applied. (left) A 3D CAD model of the KRex robot. (right) An example of two planning scenes solved using our hybrid planner.

the planner to generate actions that move an object when all primitives targeted at the object would fail (i.e., the example in Figure 8.5). Finally, using a physics engine at the planning stage allows to have multiple contacts between the robot and movable objects, even when propagating object-centric actions.

8.3 Planning experiments

We have implemented the planner described in Section 8.2.5 in the Open Motion Planning Library (OMPL) [114]. We have tested three versions of the planner that differ each other for the values imposed to p_{rand} (see Algorithm 9)

1. $p_{rand} = 0$. This is equivalent to always sample primitive sequences (object-centric actions). We denote this planner as *object centric* in all results;

2. $p_{rand} = 1$. This forces the planner to always sample a random action (robot-centric actions). We denote this planner as *robot-centric* in all results;

3. $p_{rand} = 0.5$. This allows the planner to choose primitives or random actions with equal probability. We denote this planner as *hybrid* in all results.

In this section, we want to prove the following two hypotheses

H1 on scenes easily solvable using *object-centric* actions, the *hybrid* planner performs equivalent to the *object-centric* planner in both success rate and plan time. Additionally, both the *object-centric* and *hybrid* planners outperform the *robot-centric* planner that samples only random actions;

H2 the *hybrid* planner achieves higher success rate and faster plan times than the *object-centric* or *robot-centric* planners on difficult scenes.

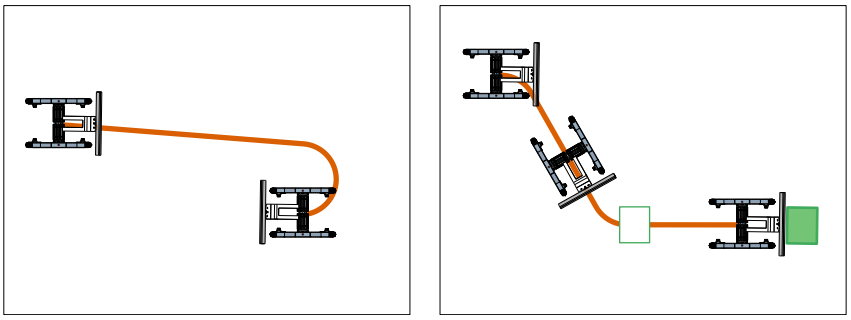


Figure 8.7 Two primitives defined for KRex (mobile manipulator). Dubins paths are used to generate paths between two poses in SE(2). (left) Robot-centric (transit) motion primitive. (right) Object-centric (push-object) motion primitive.

8.3.1 Mobile manipulator

We have tested our planner first on a mobile manipulator called KRex, depicted in Figure 8.6. We mounted an end-effector (plate) in the front of the robot, enabling the latter to push objects in the scene. The system acts as a steered car. For this robot, $\mathcal{C}^R = SE(2)$ and a control $\boldsymbol{u} = (\boldsymbol{v}, \boldsymbol{\delta}) \in \mathcal{U}$ describes the forward velocity and steering angle applied to the robot. We have used the Open Dynamics Engine (ODE)[2] as our physics model to forward propagate all actions.

In all the experiments, the task of the robot is to push a target object (depicted as a green box in Figure 8.6) into a goal region with radius 0.5 m (depicted as a green circle in Figure 8.6). We have bounded the robot to move in a world 15 m $\times 10$ m. This implies that any action that moves the robot or any object outside of this bounded region is considered invalid. For this problem, we use the following set of primitives

- robot-centric primitive. This primitive (a transit primitive) moves the robot from a start to a goal configuration in $SE(2)$ by finding the shortest length Dubins curves [30] connecting the two configurations (see Figure 8.7). Within our planner, the goal configuration pose is extracted randomly within the bounded world. Moreover, we randomly have selected a constraint on the forward velocity in the range $[-0.5, 0.5]$ m/s;

- object-centric primitive. This primitive (a push-object primitive) pushes an object along the straight line connecting a start and goal configuration for the object. The primitive returns a set of actions that first move the robot to a pose "behind" the object. In more details, this pose is chosen by computing a path (e.g., a line) between the starting robot pose and the starting object position and placing it with a distance randomly chosen in the range $[0.2, 0.3]$ m w.r.t. the latter. Finally, the primitive drives the robot straight along the ray, pushing the object (see Figure 8.7). The velocity of this primitive is chosen as for the robot-centric case.

The planner uses also random actions (another robot-centric action, see Algorithm 9, line 8). To this aim, we sample forward velocity from the range $[-0.5, 0.5]$ m/s and duration from the range

[2] See `http://www.ode.org`

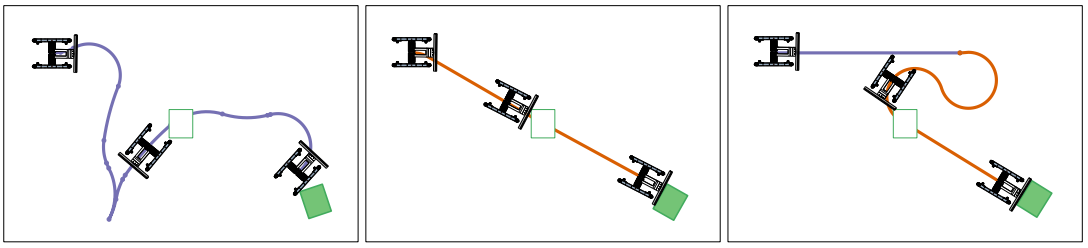


Figure 8.8 Example of solutions using the three planners for an easy scene. (left) Robot-centric planner. (center) Object-centric planner. (right) Hybrid planner.

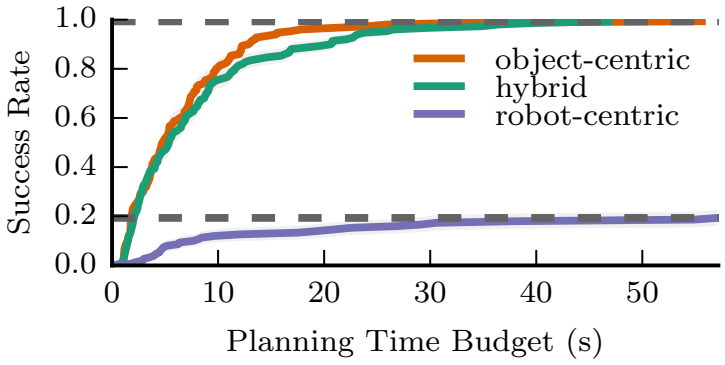


Figure 8.9 Success rate of the three planners for an easy scene using KRex (mobile manipulator). The planning time budget refers to the maximum time given to a planner to find a solution.

0.5s to 5s. Furthermore, we have defined the sampling range for steering angle using the same minimum and maximum angles as used for generating the Dubins paths.

8.3.1.1 Easy scenes

First, we have tested our planner on three scenes easily solvable using the object centric primitives defined above. Figure 8.8 shows an example scene and a single solution from each planner. We have run each planner 50 times on each scene, for a total of 150 trials per planner. For each trial, we record the total time to find a solution. Figure 8.9 reports the success rate for each planner, up to 60 s of total plan time budget. As it can be seen by the same figure, both the *hybrid* and the *object-centric* planners are able to solve the scenes in the allotted plan time, while the *robot-centric* planner often fails in finding a solution. This is due to the small progress toward the target state, randomly extracted at each iteration of the algorithm. This reflects into a planning time often larger to the allotted planning time (causing a failure).

A one-way ANOVA with Tukey HSD post-hoc analysis reveals there is no significant difference in mean plan time between the *hybrid* and *object-centric* planners ($p = 0.297$). There is a significant difference between *hybrid* and *robot-centric* ($p < 0.001$) and *object-centric* and *robot centric* ($p < 0.0001$). This supports our hypothesis (H1): the *hybrid* planner outperforms the *robot-centric* planner, and performs as well as the *object-centric* planner on simple scenes.

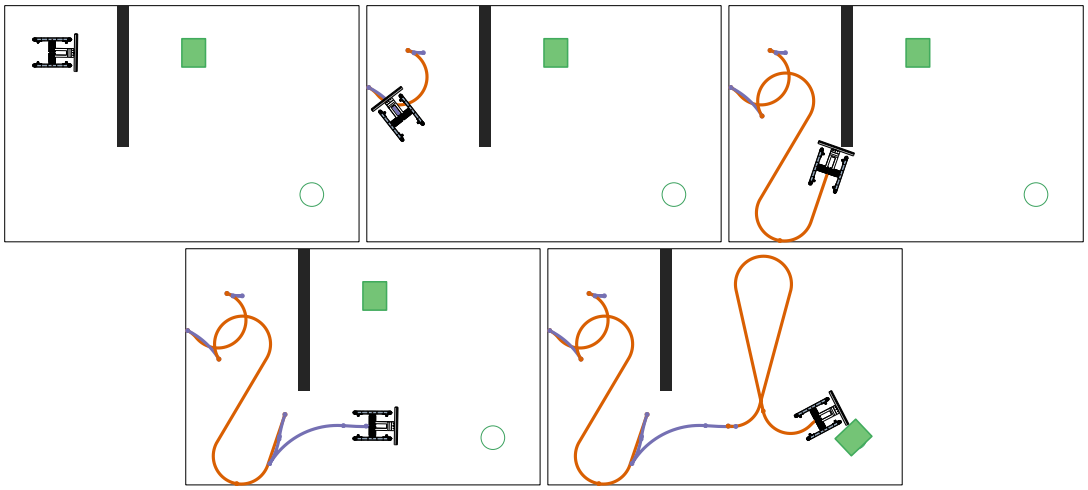


Figure 8.10 In this scene, the wall (black) serves as a static obstacle preventing application of the push primitive from the start configuration. Robot-centric actions are needed to grow the tree until such a primitive can be applied. Once available, the object-centric push primitive extends the tree to the goal. Top row: (left) start configuration; (center) a random robot-centric action is taken to reverse the robot away from the boundary; (right) a transit primitive moves the robot near the wall. Bottom row: (left) two robot-centric random actions are used to reverse the robot away from the obstacle and then drive the robot into open space; (right) an object-centric action is used to push the target to the goal region. Purple lines refer to robot-centric while orange refer to object-centric actions.

8.3.1.2 Difficult scenes

Next, we have tested the planner on more difficult scenes (i.e., scenes that require more than 60 s on average to be solved), where there are multiple static and movable obstacles. In this case, the problem cannot be solved by simply applying an object-centric primitive (see Figure 8.8). Figure 8.6, Figure 8.10 and Figure 8.11 report these scenes and an example of solutions found by our planner. Again, we run 50 times for each scene.

In the scenario depicted in Figure 8.10 the robot has to place the green box (i.e., the target object) into the green circle (i.e., the goal region). Unlike the scenario in Figure 8.8, an object-centric primitive is not able to solve the problem, since it would cause a collision with the wall depicted with a black box. Combinations of robot-centric and object-centric actions are used to solve the problem. In fact, in the solution depicted in Figure 8.10, two robot-centric primitives reverse the robot and moves it close to the wall. Then, another two robot-centric actions are aimed to move the robot away from the wall and place it in an open space. Finally, in the last snapshot, an object-centric primitive pushes the target object into the goal region. It is important to underline that even portion of object-centric primitive can be used within our framework, as it happens in the third snapshot of Figure 8.10. Even if the validation step fails in Algorithm 9 (line 15), one might save as node of the tree the last valid explored node (and the relative edge), also for avoiding to waste the computations computed so far (both for the physics propagation and the collision checks performed up to the last valid node).

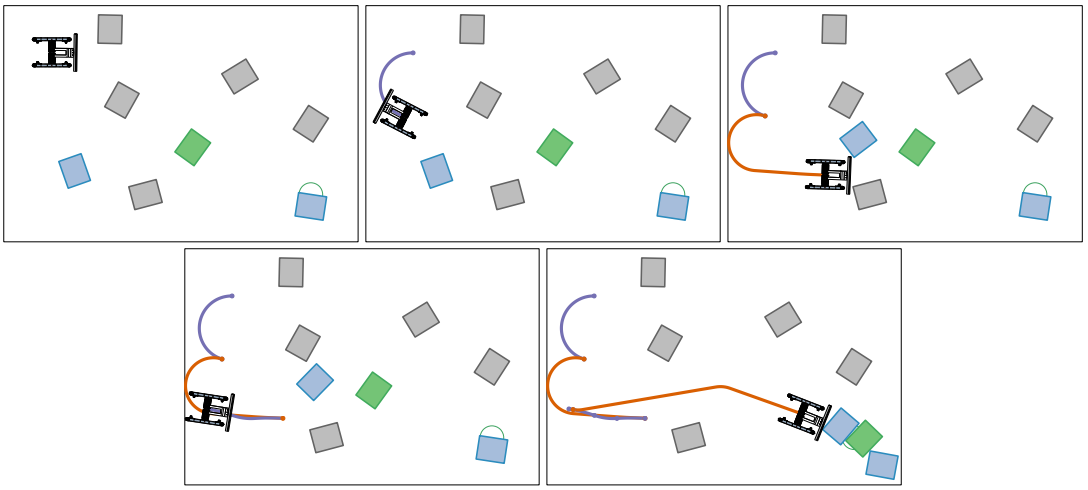


Figure 8.11 Example for a difficult scene (where object-centric actions alone fail) using KRex (mobile manipulator). The use of robot-centric actions help in repositioning the robot when near obstacles (gray boxes) or boundaries. Additionally, by using a physics model during plan time the planner can generate solutions that allow the target object (green box) to be moved into place by pushing another movable object (blue boxes) and creating a chain movement. Purple lines refer to robot-centric actions while object-centric lines refer to object-centric actions. Top row: (left) start configuration; (center) a random robot-centric action is taken to reorient the robot; (right) an object-centric action moves one box. Bottom row: (left) a robot-centric action is used to reverse the robot away from an obstacle; (right) an object-centric primitive is used to push the target in the goal region. Purple lines refer to robot-centric while orange refer to object-centric actions.

The other difficult scene on which we have tested our algorithm is depicted in Figure 8.11. As for the other scenario, the target object is the green box while the goal region is depicted with a green circle. The gray boxes are static obstacles while blue boxes are movable objects. Also in this case, a single object-centric action is not able to solve the problem, since it reveals with a collision with one of the static obstacles. The planner chooses a robot-centric action to reorient the robot (second snapshot of Figure 8.11) and a object-centric action to move one box (third snapshot of Figure 8.11). Finally, another robot-centric and an object-centric action is used to first move the robot away from a static obstacle and then to push the target object in the goal region (last two snapshots in Figure 8.11).

As it is evident from the above-mentioned experiments, our planner takes advantage of both robot-centric and object-centric primitives. This enable to find solutions that would not be able to find by using just one kind of these two actions. The success rate for the three planners in the difficult scenes (Figure 8.10 and Figure 8.11) is depicted in Figure 8.12. As can be seen, the *hybrid* planner outperforms the *object-centric* or *robot-centric* planners. A one-way ANOVA with Tukey HSD post-hoc analysis reveals that the difference in mean plan-time between the *hybrid* and *object-centric* planners is statistically significant ($p < 0.001$). The same holds between the *hybrid* and *robot-centric* planners ($p < 0.05$). This supports our hypothesis (H2): the *hybrid* planner achieves higher success rate and faster plan times than the *object centric* or *robot-centric* planners on difficult scenes.

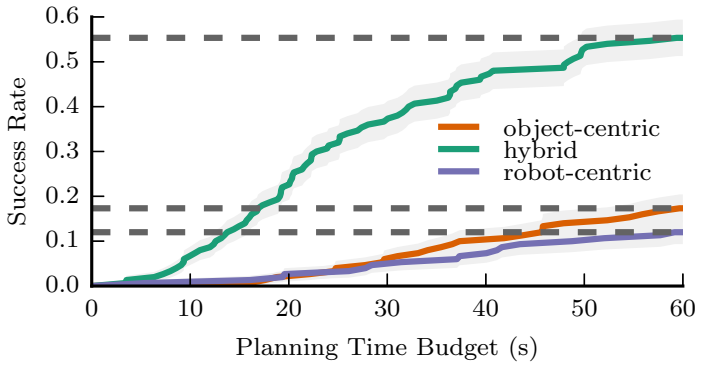


Figure 8.12 Success rate of the three planners for a difficult scene using KRex (mobile manipulator). The planning time budget refers to the maximum time given to a planner to find a solution.

8.3.1.3 Analysis results

Next, we examine some qualitative aspects of the solutions. Neither the object-centric primitive nor the robot-centric primitive allow the robot to move in reverse. As a result, in difficult scenes like those in Figure 8.10 and Figure 8.11, we see random actions used to back the robot away from obstacles and boundaries (as in the third snapshot Fig. 8.10 or in the secondo snapshot of Figure 8.11).

Additionally, as mentioned before, in the difficult scenes (Figure 8.6, Figure 8.10 and Figure 8.11) a static obstacle blocks any path solving the problem by using a single object-centric action. Thus, the push primitive fails in most applications from the start. Furthermore, the static obstacles also cause many applications of the transit primitive to fail, as they often drive the robot into an obstacle. Here, the use of random actions available to the hybrid planner is advantageous because they can be used to start tree growth from the root. Then, after the tree begins growing, the push primitive and transit primitive can be applied more freely. In fact, if we analyse all solutions generated by the hybrid planner (across easy and difficult scenes), we see that 89.2% $(207/232)$ end in an object-centric primitive. This supports our intuition that *object-centric* actions help grow the tree to the goal.

8.3.2 Household manipulator

We have tested our framework on a manipulator called HERB, depicted in Figure 8.1. Information about the robot can be found in [107, 108]. In all the experiments, the task of HERB is to push a target object into a goal region, having radius 0.1 m. The motion is restricted to the top of a table, then it is defined into a bounded 2D world. We plan for 7 DOFs left arm of the robot. Then, $\mathcal{C}^R = \mathbb{R}^7$. A control input is defined as $\boldsymbol{u} = \dot{\boldsymbol{q}} \in \mathcal{U}$, with $\dot{\boldsymbol{q}}$ a vector of joint velocities. Following the example in [63], the end-effector of the robot is constrained in a 2D plane parallel to the top of a table. Any motion that pushes an object or the robot end-effector off of the table is considered invalid. For this problem the primitive set includes

- robot-centric primitive. This primitive (a transit primitive) moves the end-effector from a start pose to a goal pose. The motion of the end-effector follows a straight line in workspace along the plane parallel to the table surface. Within our planner, the goal pose is extracted randomly into the bounded world and it is reached by the manipulator following a straight line with a velocity randomly extracted in the range $[0.1, 0.5]$ m/s;
- object-centric primitive. This primitive (a push-object primitive) pushes an object along the straight line connecting a start and goal configuration for the object. The primitive returns a set of actions that first move the robot to a pose "behind" the object. In more details, this pose is chosen by computing a path (e.g., a line) between the starting robot pose and the starting object position and placing it with a distance randomly chosen in the range $[0.3, 0.4]$ m w.r.t. the latter. Finally, the primitive drives the robot straight along the ray, pushing the object (see Figure 8.7). The velocity of this primitive is chosen as for the robot-centric case. The motion of the end-effector is confined to the plane parallel to the table surface during the entire primitive.

The planner uses also random actions (another robot-centric action, see Algorithm 9, line 8). To this aim, we sample forward velocity from the range $[-0.5, 0.5]$ m/s and duration from the range 0.5s to 5s.

We use a quasistatic model of planar pushing as our physics model [82, 83]. Because we only model objects moving in the plane, $C^i = SE(2)$ for $i = 1, \ldots, m$. It is important to underline that more sophisticated primitive can be easily included in our planner. As example, we could implement a transit primitive by calling a motion planner for the arm and allowing the end-effector to move out of the plane (like a 3D transfer from one to another side of the table). The primitives we have used are selected to have computational complexity similar to that of sampling random actions, allowing us to more fairly compare plan times between the object-centric and robot-centric approaches.

8.3.2.1 Easy scenes

We first have tested the planner on simple scenes. Three scenes were used to this scope, like the one depicted in Figure 8.13. In these scenes, the rearrangement planning problem can be easily solved by using an object-centric primitive. We run each of the three planners (*robot-centric*, *object-centric*, *hybrid* planner) 50 times on each scene, for a total of 150 trials per planner.

Figure 8.14 reports the success rate as a function of plan time for up to 300 s of total plan time budget. As can be seen, both the *hybrid* and *object-centric* planners perform equivalently. A one-way ANOVA with Tukey HSD post-hoc analysis confirms the *hybrid* and *object-centric* planners to not differ significantly in mean plan time ($p = 0.782$). The *hybrid* and *robot-centric* do show significant difference ($p < 0.0001$) as do the *object-centric* and *robot-centric* ($p < 0.0001$). This further supports **H1**.

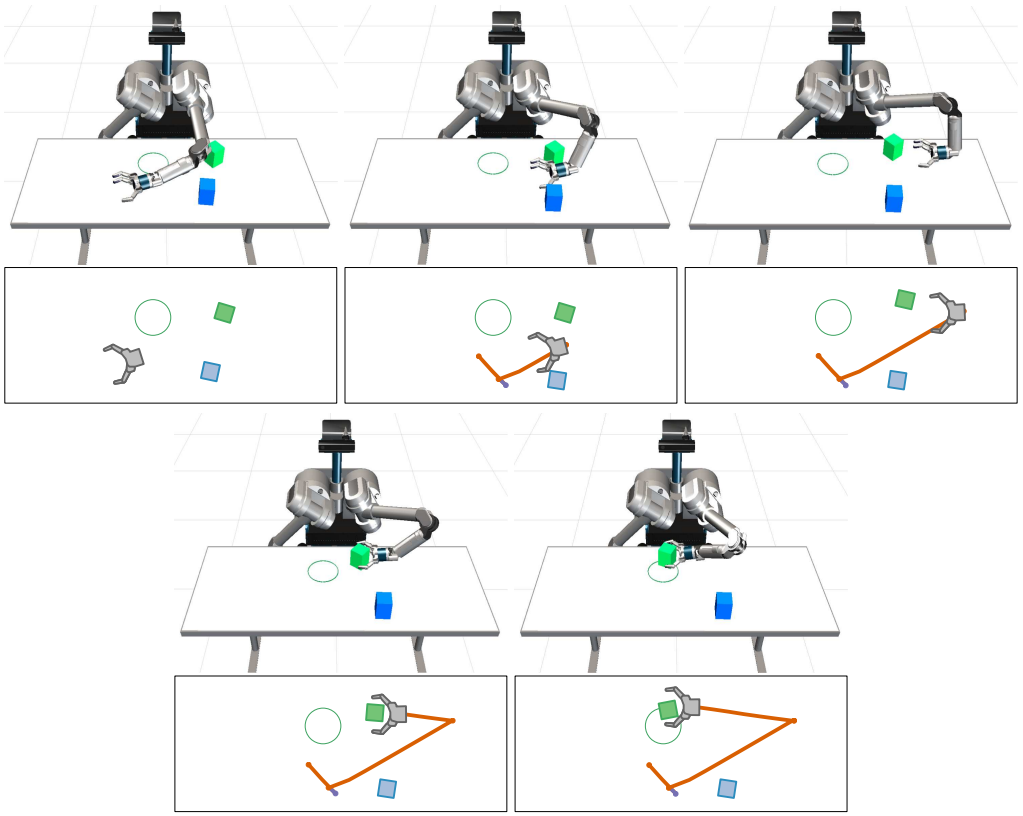


Figure 8.13 A 3D and a top view of the execution of a trajectory that moves the target object (green box) from its start configuration to the goal region (green circle). In easy scenes like this, a path for HERB (household manipulator) can be found by the hybrid planner that uses mostly robot-centric (transit) and object-centric (push) primitives. Purple lines refer to robot-centric actions while object-centric lines refer to object-centric actions. Top row: (left) start configuration; (center) the object-centric primitive is applied; (right) the same primitive moves the hand behind the target object. Bottom row: (left) the primitive moves the hand in the direction of pushing; (right) the hand pushes the target object in the goal region.

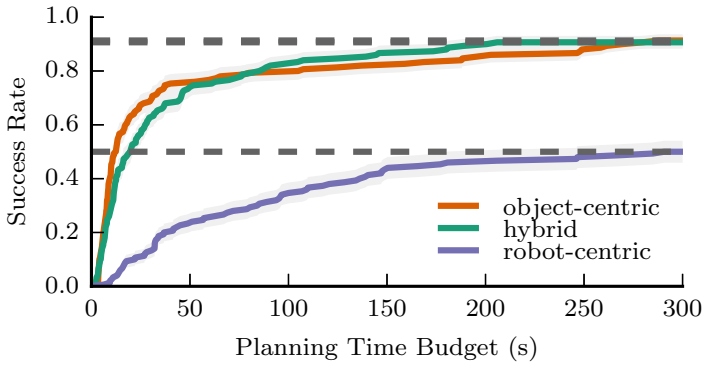


Figure 8.14 Success rate of the three planners for an easy scene using HERB (household manipulator). The planning time budget refers to the maximum time given to a planner to find a solution.

8.3.2.2 Difficult scenes

We finally have tested our planner on four difficult scenes (i.e., scenes that require more than 300 s on average to be solved), where an object-centric action is not sufficient to solve the problem.

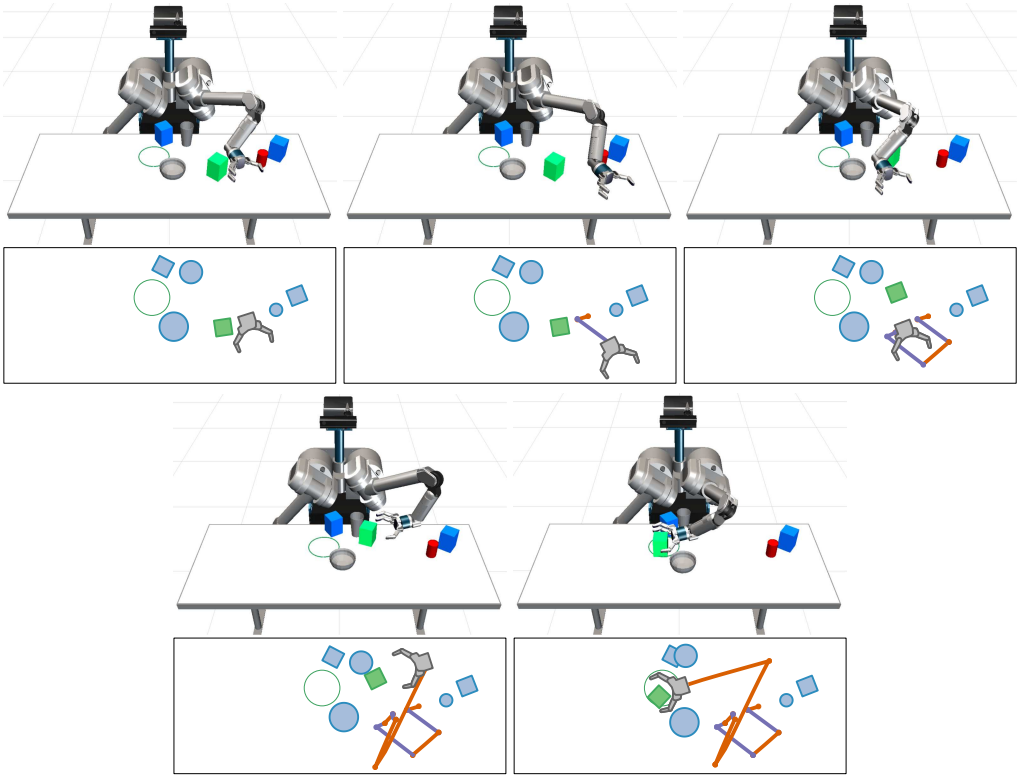


Figure 8.15 A 3D and a top view of the execution of a trajectory that moves the target object (green box) from its start configuration to the goal region (green circle). In hard scenes like this, a path for HERB (household manipulator) can be found by combining robot-centric and object-centric primitives. Purple lines refer to robot-centric actions while object-centric lines refer to object-centric actions. Top row: (left) start configuration; (center) a short robot-centric action; (right) the robot uses the back of the forearm to move the box to a location where a push primitive can be applied. Bottom row: (left) an object-centric push primitive is used to move the box to the goal region; (right) the robot moves the object, the blue box and the glass simultaneously to achieve the goal.

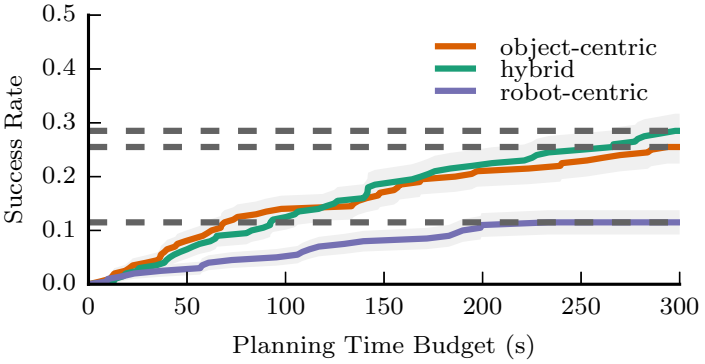


Figure 8.16 Success rate of the three planners for an difficult scene using HERB (household manipulator). The planning time budget refers to the maximum time given to a planner to find a solution.

These scenes either have obstacles blocking the path to the goal, or the goal object near the edge of the reachable workspace of the robot. Again we run each planner 50 times on each scene. As example, consider the scenario depicted in Figure 8.15. Here, the robot has to push the target

object (green box) into the goal region (green circle), as usual, but it cannot be achieved by applying a single object-centric action. In order to fulfil the task, the planner chooses to first apply a short random robot-centric action and then the robot uses the back of the forearm to move the box to a location where a push primitive can be applied (third snapshot of Figure 8.15). Finally, the planner ends with a object-centric primitive that first place the hand of the robot behind the object in the direction of the push (forth snapshot of Figure 8.15) and then push it in the goal region (last snapshot of Figure 8.15). Again, note how the planner takes advantage from both robot-centric and object-centric primitive, as well from the random motion. Combining those actions, our planner is able to solve this kind of scenes.

Figure 8.16 shows the success rate of all three planners as a function of plan time. These scenes are difficult and each planner struggles to find a solution. The *hybrid* planner performs slightly better than the other planners but a one-way ANOVA with Tukey HSV post-hoc analysis reveals the difference in mean plan time is not significant when compared to the *object-centric* ($p = 0.629$) or *robot-centric* ($p = 0.566$) planners. Even if the *hybrid* planner outperforms the *object-centric* planner, we cannot assert that this data is statistically meaningful.

We believe this result is strongly tied to the expressiveness of our primitives enabled by the physics propagation. More in details, the planner eliminates, by propagating the primitives through a physics model, many of the failure cases that would prevent application of primitives in our scenes, i.e., collisions with other movable objects. As a result, the *object-centric* planner behaves well.

8.3.2.3 Analysis results

Even for the household manipulator, we have found similar results w.r.t. the mobile manipulator presented in Section 8.3.1. In fact, random robot-centric actions are particularly helpful for moving an object in a position where an object-centric primitive can be applied. As example, this happens in Figure 8.15, where the target object cannot be moved using an object-centric primitive at the beginning. Similar to KRex, 80.3% $(155/193)$ of paths across all scenes end with an object-centric primitive, further supporting our intuition that object-centric primitives are important to goal achievement. In fact, both Figure 8.13 and Figure 8.15 end by using an object-centric primitive.

8.3.2.4 Experiment with HERB

Finally, we have validated our planner on the real robot HERB on a easy scene (see Figure 8.17. Even if our simulations were very accurate (at least at the best of the author's capabilities), some trajectories fail to achieve the goal when executed due to uncertainties in the object perception, robot forward kinematics and physical model of robot-object interaction. Prior work has shown that actions similar to the "push-primitive" used in Section8.3.2 can be uncertainty reducing [26]. Including such primitives, and guiding the planner to select them, may improve the robustness of

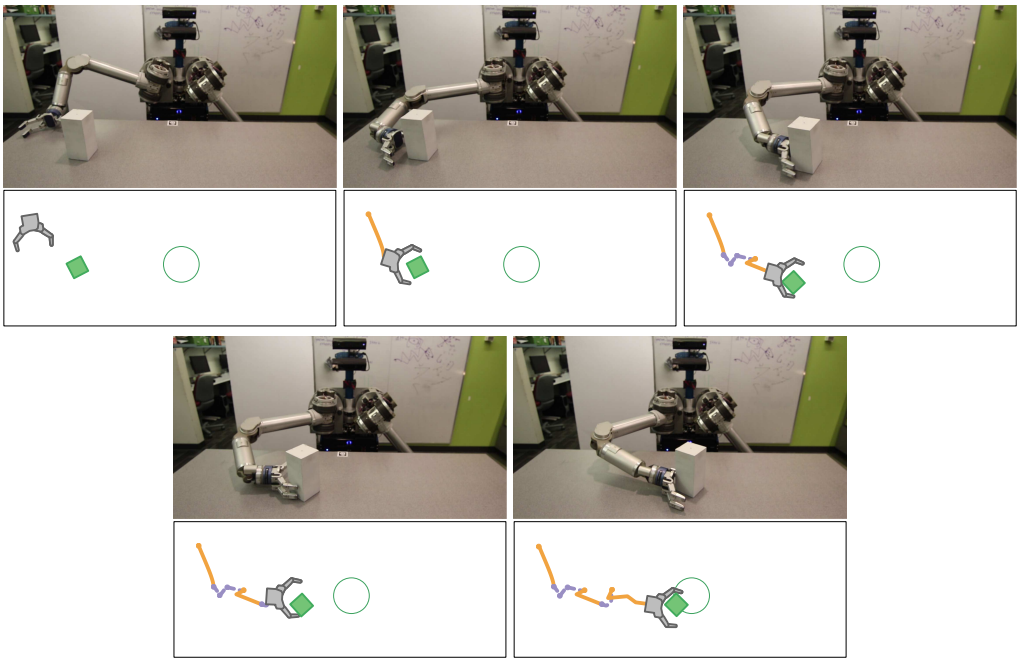


Figure 8.17 A 3D and a top view of the execution of a trajectory that moves the target object (green box) from its start configuration to the goal region (green circle). This experiment validates the proposed approach for the real robot HERB (household manipulator) for an easy scene. In the execution, the robot take advantage of both *object-centric* (orange lines) and *robot-centric* (purple lines) actions.

the generated trajectories to uncertainty, increasing the planner's applicability in everyday use. However, this proves the validity of our approach even for the real robot case. We will further investigate this aspect, searching for approaches that will alleviate this problem. As example, the robust planning approaches presented in [66, 105] can be used to this scope.

8.4 Conclusions

In this section, we have presented an approach for solving the rearrangement planning problem. This approach consists in combining robot-centric and object-centric action spaces. Our experiments have showed that by using a combination of the two action types, we are able to improve success rate and plan time when compared to planners that use only a single action type (e.g., in [63]). Finally, by using a physics model to forward propagate object-centric primitives, we are able to allow the primitives to express simultaneous object interaction and whole arm manipulation without explicit encoding.

In all results, we used a value of $p_{rand} = 0.5$ for our hybrid planner. This value allows object-centric primitives and robot-centric random actions to be selected with equal probability on each extension (see Algorithm 9). However, we can vary this value to possibly improve performance. Figure 8.18 shows the success rate as p_{rand} is varied for one of the easy KRex scenes and one of the difficult KRex scenes. The results seem to indicate that low values of p_{rand} are move

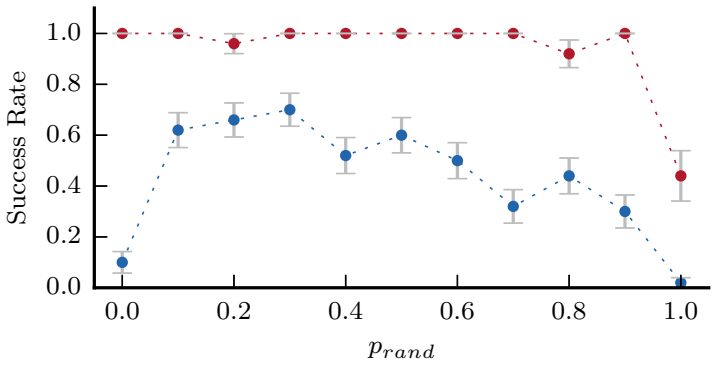


Figure 8.18 Success rate as a function of p_{rand} for the scenes in Figure 8.8 (red) and Figure 8.10 (blue).

effective. Our idea is to investigate more, in the next future, the impact of changing p_{rand} and its influence in the planning time and success rate of the proposed planner.

Note that the planner presented in this section assumes perfect knowledge of the environment. In reality, the robot has to face several sources of uncertainty when executing the trajectories generated by the planner. Prior works have shown that actions similar to the "push-primitive" used in Section 8.3 can be uncertainty reducing [26, 66]. Including such primitives, and guiding the planner to select them, may improve the robustness of the generated trajectories to uncertainty, increasing its applicability in activities of everyday lives.

Finally, it is important to underline that the randomized planner presented in Algorithm 9 has no guarantee about the optimality of the solution. One approach for dealing with it is shortcutting the paths as in [41, 98, 100, 120]. These techniques are usually performed in a post-processing phase, since they need a path to be shortcutted. The optimality of the solution comes with a cost, that is the planning time. In fact, searching for the optimal solution is harder than searching for a generic one, causing higher planning times. Ideally, a planner should be able to find a solution as fast as possible, embedding information about a criteria to be minimized. A good compromise between the optimal solution and a generic solution is to assign a budget time to the planner, finding first a solution as fast as possible and trying to increase the quality of the solution (w.r.t. criteria) within the allotted time budget. In this way, there is still no guarantee that the optimal solution is found but the quality of the solution increases as the allotted time budget increases.

To this aim, we are currently implementing the AnytimeRRT [34]. Roughly speaking, the AnytimeRRT searches for a first, fast solution. In this phase, it is almost equal to a classical RRT. The only differences are: (i) in addition to the total time budget, there is a maximum planning time for searching a solution, after which the tree is reinitialized; (ii) instead of computing the nearest neighbour, the algorithm searches for k nearest neighbours, as in the RRT* framework[59]; (iii) the k neighbours are ordered using a selection cost function and expanded with ascending order (an iteration of the algorithm stops at the first successful expansion). We use as cost function the weighted sum of the distance (`Dist`) between the selected node to the random configuration extracted (as for a RRT) and the cost (`Cost`) between the starting node and the selected node,

as suggested in [34]. For the first search, the selection cost function takes into account only the distance (ignoring the cost function), as for a classical RRT. Note that many choices for the cost function are reasonable, such as the manipulator length or mechanical work.

When a solution is found, its cost $\mathcal{C}_s$ is computed as the sum of the costs of the edges in the final path (available from the tree backtracking). At this point, the algorithm creates a new RRT. The new tree searches for solutions that improve the quality of the best solution found so far (w.r.t. the cost function defined above), by inserting only those nodes that could possibly contribute to decrease the cost. To do that, the sum between the cost-to-go of the node selected for the expansion and a heuristic function (that computes the heuristic cost from the selected node to the goal) is computed. If this value is less than $\mathcal{C}_s$ and the expansion is feasible, the node is added to the tree (since it might improve the quality of the solution). The cost-to-go is simply the sum of the edges' costs in the tree for connecting the tree root to the selected node.

At the time a new solution appears, the weight assigned to the cost in the selecting cost function is gradually increased, while the one associated to the distance function is gradually decreased. This results in expanding, with more probability, nodes having small costs that lead to low cost solutions. Note that the more solutions are found, the more importance is given to the cost (w.r.t. the distance function). This is important because the algorithm will produce more costly solutions at the beginning, paying attention to the cost if additional time is available to search for other solutions. Finally, $\mathcal{C}_s$ is gradually decreased, to ensure that the next solution will have a smaller cost than the previous one. The reader is referred to [34] for further details about the AnytimeRRT framework.

Even if the AnytimeRRT does not ensure any form of convergence to the optimal solution, it embeds three main advantages. First, it ensures to increase, from one solution to the next one, the quality of its output. Second, it combines the ability to find a fast solution (typical of the RRT) with the usage of a cost function, leading to find less and less costly solutions within a time budget. Finally, the time needed to for an optimal planner to find the optimal solution might be considerable. Within the planning time, there is no intermediate solution that the robot can execute. With this framework, a solution can be asked anytime, pointing to solutions that minimize the cost function.

Instead of using the shortcutting as a post-processing technique, our idea is to run it in parallel to the AnytimeRRT. The main advantage is that no additional time is requested. We motivate the introduction of the shortcutter saying that it might find solutions that are extremely hard to be found with the AnytimeRRT. In this way, we increase the number of solutions that can be found, adding just a small overhead.

As mentioned, the resulting algorithm is composed by the AnytimeRRT with a shortcutter running in parallel. The AnytimeRRT continuously searches for less and less costly solutions. In the meantime, the shortcutting algorithm gets the solution having the smallest cost (if no solution is found yet, it just waits for the first solution to appear). Then, it it gets two random samples from the path and it tries to solve the two-points BVP between the two samples, by means of a

Algorithm 10: Kinodynamic AnytimeRRT and shortcutting

```
T ← {nodes = {x_0 = x(t_i)}, edges = ∅, C_s = ∞} sols = ∅, t = 0;
ShortcuttingThread(sols, timeout);
while t < timeout do
    t ← GetCurrentTime();
    T ← {nodes = {x_0 = x(t_i)}, edges = ∅};
    [C_s, T] ← AnytimeRRT(C_s, T);
    path ← ExtractPath(T);
    sols ←AddSolution(path, C_s);
end
path ← ExtractBestPath(sols);
```

Algorithm 11: ShortcuttingThread ($sols$, `timeout`)

```
while t < timeout do
    t ← GetCurrentTime();
    counter ← 0, sol_found ← false;
    bestPath ←GetSmallestCostPath(sols);
    if bestPath = ∅ then
        continue;
    end
    newPath ← bestPath;
    while counter < MAX_ITERATION do
        counter ← counter + 1;
        [x_i, x_j] ← SelectRandomSamples(newPath);
        [x̄_i, x̄_j] ← Shortcut(x_i, x_j);
        if Cost(x̄_i, x̄_j) < Cost(x_i, x_j) then
            checkPath ← newPath \ [x_i, x_j] ∪ [x̄_i, x̄_j];
            if ValidatePath(checkPath) then
                newPath ← checkPath;
                sol_found ← true;
            end
        end
    end
    if sol_found then
        C_sh ← ComputeCost(newPath);
        sols ←AddSolution(newPath, C_sh);
    end
end
```

shortcutting algorithm (any of the above-mentioned approach, e.g. the one in [63] can be used to this purpose). If the cost of the new path is less than the cost of the extracted samples, that portion of the path is substituted and the new cost is computed. The path is again validated (and propagated using the physics model) to check if it achieves the goal. In a positive case, the new path is added to the solution list.

As remark, note that the smallest cost solution might come from both the AnytimeRRT or the shortcutter itself (in case the path was already shortcutted). The pseudocode is illustrated in

Algorithm 10 and Algorithm 11.

It is important to underline that we do not want to depict a complete description of the framework in this section, since it is a work in progress algorithm. The aim of the final part is to propose one possible improvement of the randomized planner presented in this chapter.

The planner presented in this chapter has been accepted in [64] and it will be presented at the *2016 IEEE Int. Conference on Robotics and Automation, Stockholm, Sweden, 16–21 May 2016*.

Chapter 9

Conclusions

In this thesis, the task-constrained motion planning problem has been faced. Here, the robot must execute an assigned task, possible requiring stepping, in environments cluttered by obstacles. This problem is particularly challenging for a humanoid robot since it is typically embedded with an high number of degrees of freedom. Moreover, it is not a free-flying system, and the motion must be appropriately generated. Finally, the implicit requirement that the robot maintains equilibrium, either static or dynamic, typically constrains the trajectory of the robot center of mass. Note also that the task constraint drastically reduces the size of the admissible planning space. We have faced this problem by means of probabilistic motion planning techniques that do not separate locomotion from task execution. We believe that the generation of the motion in a unique phase allows to fully take advantage of the humanoid capabilities. The proposed method explores the submanifold of the configuration space that is admissible with respect to tasks and all other constraints, including humanoid equilibrium. Expansion of the search tree within the constrained manifold is obtained through a hybrid scheme that generates simultaneously feet positions and whole-body motions, which are then validated via collision checking. This framework has been extended, replacing the foot displacements with movements of the center of mass. Assuming that a catalogue of primitive is available to the planner, solutions are built by concatenating feasible whole-body motions that realize such primitives and simultaneously accomplish portions of the task. The new version of the planner outperforms the previous approach in the sense that it can indifferently handle tasks specified as trajectories or as simple destinations in the task space. In addition, the new planner is able to handle a wider variety of scenarios, thanks also to the possibility of handling composite tasks (composition of manipulation and navigation tasks). Building on this planner, we have presented a framework that is able to deform the task, if needed. The framework tries to solve the motion planning problem on the first assigned task and, in case it is too hard to be fulfilled, it appropriately deforms the latter. By repetitive calls on the deformation mechanism and the planner, our framework is able to solve the motion planning problem. Note that both the detection and the deformation mechanism is automatic, without any input from a user. We have validated our framework through planning experiments and dynamic playbacks in V-REP.

However, there are situations in which a humanoid robot cannot plan its motion in an offline way but it has to be equipped with reactive capabilities. As example, this is needed when a robot and moving obstacles (e.g., humans) share the same environment. In order to guarantee a basic level of safety (for both the humanoid and the obstacles), the robot must execute, as fast as possible, an evasion maneuver whose aim is to prevent a collision. Our approach goes through several conceptual steps. Once the entrance of the moving obstacle in the safety area is detected, its approach direction relative to the robot is determined. On the basis of this information, a suitable evasion maneuver represented by footsteps is generated using a controlled unicycle as a reference model. From the footstep sequence, we compute an appropriate trajectory for the Center of Mass of the humanoid, which is finally used to generate joint motion commands that track such trajectory. In the interest of safety, it is obviously essential that the reaction time (from detection of the moving obstacle to start of the evasive motion) is as small as possible. This is achieved by making use of closed-form expressions throughout the method, and results in an algorithm suitable for real-time implementation. We have validated the proposed method through experiments and dynamic playbacks in V-REP.

Finally, probabilistic planners aimed at solving the rearrangement planning problem has been proposed. In this problem, a robot must execute a manipulation task (e.g., move a target object in a goal region) whose execution requires the interaction with movable objects that are in the environment. The inclusion of physics models in the planners have revealed the opportunity to create plans that exhibit robot manipulation and simultaneous object interaction. Moreover, object-centric and robot-centric primitives were introduced to reduce the planning time. The combination of such primitives have had great impact on the types of solutions the planner might produce (with respect to using just one of the two primitive types). As for the other planners, experiments were performed to validate the proposed planner.

During the Ph.D., we have also faced the problem of identification and control of an heterogeneous multi-robot team. Here, the team is composed by a single Unmanned Aerial Vehicle (UAV) and multiple Unmanned Ground Vehicles (UGVs). The first step was the identification of the ground robots, since we do not applied any form of tagging on the UGVs. This means that the UGVs look like the same from the point of the view of the UAV, that is able to detect them by means of a monocular camera. The unknown data association problem was tackled with the Probability Hypothesis Density (PHD) filter. The information about the identities of the UGVs is essential also for the control of the team, since one has to know, given a command, which is the robot to which it has to be sent. Once the UGVs identities are known, our attention was devoted to the control of the heterogeneous team. We have defined three tasks that the team has to fulfil. First, the UGVs should be always in the field of view of the camera mounted on the UAV. This is ensured by controlling the centroid and the variance of the features (a feature corresponds to a UGV) in the image plane. This would ensure that the relative localization filter (PHD) will provide position estimates of the UGVs. Second, a typical task of interest for ground robots is formation control (e.g., place the ground robots on a circular formation). Finally, the last task

is the navigation, in order to let the entire multi-robot system to navigate and then explore the environment. Another important remark is related to obstacle avoidance, an essential feature for any multi robot system including ground robots. We assume that each UGV is equipped with an obstacle avoidance algorithm that runs separately on each robot. In our framework, we propose two different control schemes aimed at fulfilling the above-mentioned tasks, depending on the robots devoted to the accomplishment of the visual task. The research on the heterogeneous system have produced some publications [18, 97, 109].

In the future, we want to try to extend the probabilistic planner for humanoid robots by introducing additional center of mass movement primitives and considering other heuristic for the deformation mechanism. At the same time, we would like to develop a gazing strategy for faster detection of the moving obstacle and formulate additional evasive strategies. A great improvement would be to develop the replanning scheme in the case of finite-size feet. Finally, regarding the rearrangement problem, we would like to improve the planning time by means of parallelization at the planning stage (e.g., [49]). On the other side, a bidirectional version of the proposed planner would be interesting to develop. Finally, we want to try to implement and compare other algorithms, such as the Anytime plus shortcutting algorithm mentioned in Section 8.4, in order to improve the quality of the solution.

Bibliography

[1] P. Agarwal, J. Latombe, R. Motwani, and P. Raghavan. Nonholonomic path planning for pushing a disk among obstacles. In *1997 IEEE Int. Conf. on Robotics and Automation*, pages 3124–3129, 1997.

[2] J. Barry, K. Hsiao, L. P. Kaelbling, and T. Lozano-Pérez. Manipulation with multiple action types. In *Int. Symposium on Experimental Robotics*, 2012.

[3] O. Ben-Shahar and E. Rivlin. Practical pushing planning for rearrangement tasks. In *1998 IEEE Int. Conf. on Robotics and Automation*, pages 549–565, 1998.

[4] O. Ben-Shahar and E. Rivlin. To push or not to push: On the rearrangement of movable objects by a mobile robot. *IEEE Trans. on Systems, Man, & Cybernetics. Part B: Cybernetics*, 28(5):667–679, 1998.

[5] K. Bouyarmane and A. Kheddar. Using a multi-objective controller to synthesize simulated humanoid robot motion with changing contact configuration. In *2011 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 4414–4419, 2011.

[6] K. Bouyarmane and A. Kheddar. Humanoid robot locomotion and manipulation step planning. *Advanced Robotics*, 26(10):1099–1126, 2012.

[7] O. Brock and O. Khatib. Elastic strips: A framework for motion generation in human environments. *International Journal of Robotics Research*, 21(12):1031–1052, 2002.

[8] F. Burget, A. Hornung, and M. Bennewitz. Whole-body motion planning for manipulation of articulated objects. In *2013 IEEE Int. Conf. on Robotics and Automation*, pages 1656–1662, 2013.

[9] T. Buschmann, S. Lohmeier, M. Bachmayer, H. Ulbrich, and F. Pfeiffer. A collocation method for real-time walking pattern generation. In *7th 2007 IEEE-RAS Int. Conf. on Humanoid Robots*, pages 1–6, 2007.

[10] M. Cefalo and G. Oriolo. Dynamically feasible task-constrained motion planning with moving obstacles. In *2014 IEEE Int. Conf. on Robotics and Automation*, pages 2045–2050, 2013.

[11] M. Cefalo, G. Oriolo, and M. Vendittelli. Planning safe cyclic motions under repetitive task constraints. In *2013 IEEE Int. Conf. on Robotics and Automation*, pages 3807–3812, 2013.

[12] M. Cefalo, G. Oriolo, and M. Vendittelli. Task constrained motion planning with moving obstacles. In *2013 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 5758–5763, 2013.

[13] L. Chang, S. Srinivasa, and N. Pollard. Planning pre-grasp manipulation for transport tasks. In *2010 IEEE Int. Conf. on Robotics and Automation*, pages 2697–2704, 2010.

[14] P. C. Chen and Y. K. Hwang. Practical path planning among movable obstacles. In *1991 IEEE Int. Conf. on Robotics and Automation*, pages 444–449, 1991.

[15] J. Chestnutt, M. Lau, G. Cheung, J. Kuffner, J. Hodgins, and T. Kanade. Footstep planning for the honda ASIMO humanoid. In *2005 IEEE Int. Conf. on Robotics and Automation*, pages 629–634, 2005.

[16] H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Cambridge, MA, 2005.

[17] M. Cognetti, P. Mohammadi, G. Oriolo, and M. Vendittelli. Task-oriented whole-body planning for humanoids based on hybrid motion generation. In *2014 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 4071–4076, 2014.

[18] M. Cognetti, G. Oriolo, P. Peliti, L. Rosa, and P. Stegagno. Cooperative control of a heterogeneous multi-robot system based on relative localization. In *2014 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 350–356, 2014.

[19] M. Cognetti, P. Mohammadi, and G. Oriolo. Whole-body motion planning for humanoids based on com movement primitives. In *2015 15th IEEE-RAS Int. Conf. on Humanoid Robots*, pages 1090–1095, 2015.

[20] M. Cognetti, D. De Simone, L. Lanari, and G. Oriolo. Real-time planning and execution of evasive motions for a humanoid robot. In *2016 IEEE Int. Conf. on Robotics and Automation*, 2016. Accepted paper.

[21] M. Cognetti, V. Fioretti, and G. Oriolo. Whole-body planning for humanoids along deformable tasks. In *2016 IEEE Int. Conf. on Robotics and Automation*, 2016. Accepted paper.

[22] S. Dalibard, A. Nakhaei, F. Lamiraux, and J. Laumond. Manipulation of documented objects by a walking humanoid robot. In *2010 10th IEEE-RAS Int. Conf. on Humanoid Robots*, pages 518 –523, 2010.

[23] S. Dalibard, A. El Khoury, F. Lamiraux, A. Nakhaei, M. Taïx, and J.-P. Laumond. Dynamic walking and whole-body motion planning for humanoid robots: An integrated approach. *International Journal of Robotics Research*, 32(9–10):1089–1103, 2013.

[24] A. De Luca and F. Flacco. Integrated control for phri: Collision avoidance, detection, reaction and collaboration. In *Biomedical Robotics and Biomechatronics (BioRob), 2012 4th IEEE RAS EMBS International Conference on*, pages 288–295, 2012.

[25] J. Denavit and R. S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *Journal of Applied Mechanics*, 22:215–221, 1955.

[26] M. Dogar and S. Srinivasa. Push-grasping with dexterous hands: Mechanics and a method. In *2010 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 2123–2130, 2010.

[27] M. Dogar and S. Srinivasa. A framework for push-grasping in clutter. In *2007 Robotics: Science and Systems*, 2011.

[28] M. Dogar and S. Srinivasa. A planning framework for nonprehensile manipulation under clutter and uncertainty. *IEEE Trans. on Systems, Man, & Cybernetics. Part B: Cybernetics*, 33(3):217–236, 2012.

[29] M. R. Dogar, K. Hsiao, M. Ciocarlie, and S. Srinivasa. Physics-based grasp planning through clutter. In *2012 Robotics: Science and Systems*, 2012.

[30] L. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79(3):497–516, 1957.

[31] J. Englsberger, C. Ott, and A. Albu-Schaffer. Three-dimensional bipedal walking control using divergent component of motion. In *2013 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 2600–2607, 2013.

[32] A. Escande, A. Kheddar, and S. Miossec. Planning support contact-points for humanoid robots and experiments on hrp-2. In *2006 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 2974–2979, 2006.

[33] A. Faragasso, G. Oriolo, A. Paolillo, and M. Vendittelli. Vision-based corridor navigation for humanoid robots. In *2013 IEEE Int. Conf. on Robotics and Automation*, pages 3190–3195, 2013.

[34] D. Ferguson and A. Stentz. Anytime RRTs. In *2006 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 5369–5375, 2006.

[35] F. Flacco and A. D. Luca. Safe physical human-robot collaboration. In *2013 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, page 2072, 2013.

[36] C. V. Geem, T. Simeon, and J.-P. Laumond. Mobility analysis for feasibility studies in cad models of industrial environments. In *1999 IEEE Int. Conf. on Robotics and Automation*, pages 4071–4076, 1999.

[37] R. Geraerts and M. H. Overmars. Creating high-quality paths for motion planning. *International Journal of Robotics Research*, 26(8):845–863, 2002.

[38] M. Gupta and G. Sukhatme. Using manipulation primitives for brick sorting in clutter. In *2010 IEEE Int. Conf. on Robotics and Automation*, pages 3883–3889, 2012.

[39] K. Harada, M. Morisawa, K. Miura, S. Nakaoka, K. Fujiwara, K. Kaneko, and S. Kajita. Kinodynamic gait planning for full-body humanoid robots. In *2008 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 1544–1550, 2008.

[40] K. Hauser and V. Ng-Thow-Hing. Fast smoothing of manipulator trajectories using optimal bounded-acceleration shortcuts. In *2010 IEEE Int. Conf. on Robotics and Automation*, pages 2493–2498, 2010.

[41] K. Hauser and V. Ng-Thow-Hing. Fast smoothing of manipulator trajectories using optimal bounded-acceleration shortcuts. In *2010IEEE Int. Conf. on Robotics and Automation*, pages 2493–2498, 2010.

[42] K. Hauser and V. Ng-Thow-Hing. Randomized multi-modal motion planning for a humanoid robot manipulation task. *International Journal of Robotics Research*, 30(6):678–698, 2011.

[43] K. Hauser, T. Bretl, K. Harada, and J.-C. Latombe. Using motion primitives in probabilistic sample-based planning for humanoid robots. In *WAFR*, 2006.

[44] K. Hauser, T. Bretl, J.-C. Latombe, K. Harada, and B. Wilcox. Motion planning for legged robots on varied terrain. *International Journal of Robotics Research*, 27(11–12):1325–1349, 2008.

[45] J. Haustein, J. King, S. Srinivasa, and T. Asfour. Kinodynamic randomized rearrangement planning via dynamic transitions between statically stable states. In *2015 IEEE Int. Conf. on Robotics and Automation*, pages 3075–3082, 2015.

[46] A. Herdt, H. Diedam, P.-B. Wieber, D. Dimitrov, K. Mombaur, and M. Diehl. Online walking motion generation with automatic footstep placement. *Advanced Robotics*, 24 (5-6):719–737, 2010.

[47] A. Hof, M. Gazendam, and W. Sinke. The condition for dynamic stability. *Journal of Biomechanics*, 38(1):1–8, 2005.

[48] C. K. Hsee, Y. Tu, Z. Y. Lu, and B. Ruan. Approach aversion: Negative hedonic reactions toward approaching stimuli. *Journal of Personality and Social Psychology*, 106(5):699–712, 2014.

[49] J. Ichnowski and R. Alterovitz. Parallel sampling-based motion planning with superlinear speedup. In *2012 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 1206–1212, 2012.

[50] S. Jentzsch, A. Gaschler, O. Khatib, and A. Knoll. Mopl: A multi-modal path planner for generic manipulation tasks. In *2015 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 6208–6214, 2015.

[51] S. Kajita, F. Kanehiro, K. Kaneko, K. Yokoi, and H. Hirukawa. The 3d linear inverted pendulum mode: a simple modeling for a biped walking pattern generation. In *2001 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, volume 1, pages 239–246, 2001.

[52] S. Kajita, O. Matsumoto, and M. Saigo. Real-time 3d walking pattern generation for a biped robot with telescopic legs. In *2001 IEEE Int. Conf. on Robotics and Automation*, pages 2299–2308, 2001.

[53] S. Kajita, F. Kanehiro, K. Kaneko, K. Fujiwara, K. Yokoi, and H. Hirukawa. A realtime pattern generator for biped walking. In *2002 IEEE Int. Conf. on Robotics and Automation*, pages 27–31, 2002.

[54] S. Kajita, F. Kanehiro, K. Kaneko, K. Fujiwara, K. Harada, K. Yokoi, and H. Hirukawa. Biped walking pattern generation by using preview control of zero-moment point. In *2003 IEEE Int. Conf. on Robotics and Automation*, pages 1620–1626, 2003.

[55] S. Kajita, H. Hirukawa, K. Harada, and K. Yokoi. *Introduction to Humanoid Robotics*. Springer Publishing Company Inc., 2014.

[56] O. Kanoun, F. Lamiraux, P.-B. Wieber, F. Kanehiro, E. Yoshida, and J.-P. Laumond. Prioritizing linear equality and inequality systems: Application to local motion planning for redundant robots. In *2009 IEEE Int. Conf. on Robotics and Automation*, pages 2939 – 2944, 2009.

[57] O. Kanoun, F. Lamiraux, and P. B. Wieber. Kinematic control of redundant manipulators: Generalizing the task-priority framework to inequality task. *IEEE Trans. on Robotics*, 27 (4):785–792, 2011.

[58] O. Kanoun, J.-P. Laumond, and E. Yoshida. Planning foot placements for a humanoid robot: A problem of inverse kinematics. *International Journal of Robotics Research*, 30 (4):476–485, 2011.

[59] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *International Journal of Robotics Research*, 30(7):846–894, 2011.

[60] S. M. Khansari-Zadeh and A. Billard. A dynamical system approach to realtime obstacle avoidance. *Autonomous Robots*, 32(4):433–454, 2012.

[61] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *1985 IEEE Int. Conf. on Robotics and Automation*, volume 2, pages 500–505, 1985.

[62] J. King, M. Klingensmith, C. Dellin, M. Dogar, P. Velagapudi, N. Pollard, and S. Srinivasa. Pregrasp manipulation as trajectory optimization. In *2013 Robotics: Science and Systems*, 2013.

[63] J. King, J. Haustein, S. Srinivasa, and T. Asfour. Nonprehensile whole arm rearrangement planning on physics manifolds. In *2015 IEEE Int. Conf. on Robotics and Automation*, pages 2508–2515, 2015.

[64] J. King, M. Cognetti, and S. S. Srinivasa. Rearrangement planning using object-centric and robot-centric action spaces. In *2016 IEEE Int. Conf. on Robotics and Automation*, 2016. Accepted paper.

[65] T. Koolen, T. de Boer, J. Rebula, A. Goswami, and J. Pratt. Capturability-based analysis and control of legged locomotion, part 1: Theory and application to three simple gait models. *International Journal of Robotics Research*, 31(9):1094–1113, 2012.

[66] M. Koval, J. King, N. Pollard, and S. Srinivasa. Robust trajectory selection for rearrangement planning as a multi-armed bandit problem. In *2015 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 4348–4355, 2015.

[67] T. Kruse, A. K. Pandey, R. Alami, and A. Kirsch. Human-aware robot navigation: A survey. *Robotics and Autonomous Systems*, 61(12):1726 – 1743, 2013.

[68] J. Kuffner, S. Kagami, K. Nishiwaki, M. Inaba, and H. Inoue. Dynamically-stable motion planning for humanoid robots. *Autonomous Robots*, 12:105–118, 2002.

[69] J. Kuffner, K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue. Motion planning for humanoid robots. In *Proc. 11th Int. Symp. of Robotics Research (ISRR 2003)*, 2003.

[70] J. Kuffner, J.J., K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue. Footstep planning among obstacles for biped robots. In *2001 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 500–505, 2001.

[71] J. J. Kuffner and S. M. Lavalle. Rrt-connect: An efficient approach to single-query path planning. In *2000 IEEE Int. Conf. on Robotics and Automation*, volume 2, pages 995–1001, 2000.

[72] B. Lacevic, P. Rocco, and A. Zanchettin. Safety assessment and control of robotic manipulators using danger field. *IEEE Trans. on Robotics*, 29(5):1257–1270, 2013.

[73] L. Lanari and S. Hutchinson. Inversion-based gait generation for humanoid robots. In *2015 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2015.

[74] L. Lanari and S. Hutchinson. Planning desired center of mass and zero moment point trajectories for bipedal locomotion. In *2015 15thIEEE-RAS Int. Conf. on Humanoid Robots*, pages 637–642, 2015.

[75] L. Lanari, S. Hutchinson, and L. Marchionni. Boundedness issues in planning of locomotion trajectories for biped robots. In *2014 14thIEEE-RAS Int. Conf. on Humanoid Robots*, pages 951–958, 2014.

[76] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, USA, 1991. ISBN 079239206X.

[77] S. LaValle and J. Kuffner. Randomized kinodynamic planning. *International Journal of Robotics Research*, 20(5):378–400, 2001.

[78] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical Report TR 98-11, Computer Science Dept., Iowa State University, 1998.

[79] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006. ISBN 0521862051.

[80] S. Lengagne, P. Mathieu, A. Kheddar, and E. Yoshida. Generation of dynamic multi-contact motions: 2d case studies. In *2010 10th IEEE-RAS Int. Conf. on Humanoid Robots*, pages 14–20, 2010.

[81] T. Lozano-Pérez, M. Mason, and R. Taylor. Automatic synthesis of fine-motion strategies for robots. *International Journal of Robotics Research*, 3(1):3–24, 1984.

[82] K. Lynch and M. Mason. Stable pushing: Mechanics, controllability, and planning. In *Workshop on Algorithmic Foundations of Robotics*, 1995.

[83] K. Lynch and M. Mason. Stable pushing: Mechanics, controllability, and planning. *International Journal of Robotics Research*, 15(6):533–556, 1996.

[84] M. Mason. Automatic planning of fine motions: Correctness and completeness. In *1984 IEEE Int. Conf. on Robotics and Automation*, pages 492–503, 1984.

[85] K. Mombaur, A. Truong, and J.-P. Laumond. From human to humanoid locomotion – an inverse optimal control approach. *Autonomous Robots*, 28:369–383, 2010.

[86] M. Morisawa, K. Harada, S. Kajita, K. Kaneko, J. Sola, E. Yoshida, N. Mansard, K. Yokoi, and J.-P. Laumond. Reactive stepping to prevent falling for humanoids. In *2009 9thIEEE-RAS Int. Conf. on Humanoid Robots*, pages 528–534, 2009.

[87] D. Nieuwenhuisen, A. van der Stappen, and M. Overmars. An effective framework for path planning amidst movable obstacles. In S. Akella, N. Amato, W. Huang, and B. Mishra, editors, *Algorithmic Foundation of Robotics VII*, volume 47 of *Springer Tracts in Advanced Robotics*, pages 87–102. Springer Berlin Heidelberg, 2008.

[88] K. Nishiwaki and S. Kagami. Online walking control system for humanoids with short cycle pattern generation. *International Journal of Robotics Research*, 28(6):729–742, 2009.

[89] K. Nishiwaki and S. Kagami. Simultaneous planning of com and zmp based on the preview control method for online walking control. In *2011 11thIEEE-RAS Int. Conf. on Humanoid Robots*, pages 745–751, 2011.

[90] G. Oriolo and C. Mongillo. Motion planning for mobile manipulators along given end-effector paths. In *2005 IEEE Int. Conf. on Robotics and Automation*, pages 2154–2160, 2005.

[91] G. Oriolo and M. Vendittelli. A control-based approach to task-constrained motion planning. In *2009 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 297–302, 2009.

[92] G. Oriolo, M. Ottavi, and M. Vendittelli. Probabilistic motion planning for redundant robots along given end-effector paths. In *2002 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 1657–1662, 2002.

[93] M. Otte and E. Frazzoli. RRT^x : Real-time motion planning/replanning for environments with unpredictable obstacles. In *Int. Workshop on the Algorithmic Foundations of Robotics*, 2014.

[94] C. Park, J. Pan, and D. Manocha. ITOMP: Incremental trajectory optimization for real-time replanning in dynamic environments. In *Int. Conf. on Automated Planning and Scheduling*, 2012.

[95] N. Perrin, O. Stasse, L. Baudouin, F. Lamiraux, and E. Yoshida. Fast humanoid robot collision-free footstep planning using swept volume approximations. *IEEE Trans. on Robotics*, 28(2):427–439, 2012.

[96] J. Pettré, J.-P. Laumond, and T. Siméon. A 2-stages locomotion planner for digital actors. In *2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 258–264, 2003.

[97] L. Rosa, M. Cognetti, A. Nicastro, P. Alvarez, and G. Oriolo. Multi-task cooperative control in a heterogeneous ground-air robot group. In *3rd IFAC Workshop on Multivehicle Systems*, 2015.

[98] G. Sánchez and J.-C. Latombe. On delaying collision checking in PRM planning - application to multi-robot coordination. *International Journal of Robotics Research*, 21(1): 5–26, 2002.

[99] S. Sekhavat, P. Svestka, J.-P. Laumond, and M. H. Overmars. Multi-level path planning for nonholonomic robots using semi-holonomic subsystems. *International Journal of Robotics Research*, 17:840–857, 1996.

[100] S. Sekhavat, P. Švestka, J.-P. Laumond, and M. Overmars. Multi-level path planning for nonholonomic robots using semi-holonomic subsystems. *International Journal of Robotics Research*, 17(8):840–857, 1998.

[101] B. Siciliano. Kinematic control of redundant robot manipulators: A tutorial. *International Journal of Robotics Research*, 3(3):201–212, 1990.

[102] B. Siciliano and O. Khatib. *Springer Handbook of Robotics*. Springer, 2008.

[103] B. Siciliano and J.-J. Slotine. A general framework for managing multiple tasks in highly redundant robotic systems. In *Fifth International Conference on Advanced Robotics*, pages 1211–1216, 1991.

[104] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics: Modelling, Planning and Control*. Springer, 2009.

[105] A. Sieverling, N. Kuhnen, and O. Brock. Sensor-based, task-constrained motion generation under uncertainty. In *2014 IEEE Int. Conf. on Robotics and Automation*, pages 4348–4355, 2014.

[106] T. Siméon, J.-P. Laumond, J. Cortés, and A. Sahbani. Manipulation planning with probabilistic roadmaps. *International Journal of Robotics Research*, 23(7–8):729–746, 2004.

[107] S. Srinivasa, D. Ferguson, C. Helfrich, D. Berenson, A. Collet, R. Diankov, G. Gallagher, G. Hollinger, J. Kuffner, and M. Weghe. HERB: A Home Exploring Robotic Butler. *Autonomous Robots*, 28(1):5–20, 2010.

[108] S. S. Srinivasa, D. Berenson, M. Cakmak, A. Collet, M. Dogar, A. Dragan, R. A. K. amd T. D. Niemueller, K. Strabala, J. M. Vandeweghe, and J. Ziegler. HERB 2.0: Lessons Learned from Developing a Mobile Manipulator for the Home. *Proceedings of the IEEE*, 100(8):1–19, 2012.

[109] P. Stegagno, M. Cognetti, L. Rosa, P. Peliti, and G. Oriolo. Relative localization and identification in a heterogeneous multi-robot system. In *2013 IEEE Int. Conf. on Robotics and Automation*, pages 1857–1864, 2013.

[110] M. Stilman. Task constrained motion planning in robot joint space. In *2007 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 3074–3081, 2007.

[111] M. Stilman. Global manipulation planning in robot joint space with task constraints. *IEEE Trans. on Robotics*, 26(3):576–584, 2010.

[112] M. Stilman and J. Kuffner. Navigation among movable obstacles: Real-time reasoning in complex environments. In *2004 IEEE Int. Conf. on Robotics and Automation*, pages 322–341, 2004.

[113] M. Stilman, J. Schamburek, J. Kuffner, and T. Asfour. Manipulation planning among movable obstacles. In *2007 IEEE Int. Conf. on Robotics and Automation*, pages 3327–3332, 2007.

[114] I. Sucan, M. Moll, and L. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012.

[115] T. Sugihara, Y. Nakamura, and H. Inoue. Realtime humanoid motion generation through zmp manipulation based on inverted pendulum control. In *2002 IEEE Int. Conf. on Robotics and Automation*, pages 1404–1409, 2002.

[116] T. Takenaka, T. Matsumoto, and T. Yoshiike. Real time motion generation and control for biped robot-1^{st} report: Walking gait pattern generation. In *2009 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 1084–1091, 2009.

[117] T.-V.-A. Truong, D. Flavigne, J. Pettre, K. Mombaur, and J.-P. Laumond. Reactive synthesizing of human locomotion combining nonholonomic and holonomic behaviors. In *3rd IEEE/RAS-EMBS International Conference on Biomedical Robotics and Biomechatronics*, pages 632–637, 2010.

[118] J. van den Berg, M. Stilman, J. Kuffner, M. Lin, and D. Manocha. Path planning among movable obstacles: a probabilistically complete approach. In *Workshop on the Algorithmic Foundations of Robotics*, pages 322–341, 2004.

[119] J. van den Berg, M. Stilman, J. Kuffner, M. Lin, and D. Manocha. Path planning among movable obstacles: A probabilistically complete approach. In *Algorithmic Foundation of Robotics VIII*, volume 57 of *Springer Tracts in Advanced Robotics*, pages 599–614. 2010.

[120] C. Van Geem, T. Siméon, and J.-P. Laumond. Mobility analysis for feasibility studies in cad models of industrial environments. In *1999 IEEE Int. Conf. on Robotics and Automation*, volume 3, pages 1770–1775, 1999.

[121] J. Vannoy and J. Xiao. Real-Time Adaptive Motion Planning (RAMP) of Mobile Manipulators in Dynamic Environments with Unforeseen Changes. *IEEE Trans. on Robotics*, 24(5):1199–1212, 2008.

[122] M. Vukobratovic and B. Borovac. Zero moment point – thirty five years of its life. *International Journal of Humanoid Robotics*, 1(1):157–173, 2004.

[123] G. Wilfong. Motion planning in the presence of movable obstacles. *Annals of Mathematics and Artificial Intelligence*, 41(4):764–790, 1991.

[124] Y. Yang and O. Brock. Elastic roadmaps - motion generation for autonomous mobile manipulation. *Autonomous Robots*, 28(1):113–130, 2010.

[125] E. Yoshida and F. Kanehiro. Reactive robot motion using path replanning and deformation. In *2011 IEEE Int. Conf. on Robotics and Automation*, pages 5456–5462, 2011.

[126] E. Yoshida, I. Belousov, C. Esteves, and J.-P. Laumond. Humanoid motion planning for dynamic tasks. In *2005 5th IEEE-RAS Int. Conf. on Humanoid Robots*, pages 1–6, 2005.

[127] M. Zucker, N. Ratliff, A. D. Dragan, M. Pivtoraiko, M. Klingensmith, C. M. Dellin, J. A. Bagnell, and S. S. Srinivasa. CHOMP: Covariant Hamiltonian optimization for motion planning. *International Journal of Robotics Research*, 32(9-10):1164–1193, 2013.