Humanoid sensory-based multi-contact planning in Real environment

Stanislas Brossette

Department of Engineering
Université de MontPellier

This dissertation is submitted for the degree of
Doctor of Philosophy

November 2015

I would like to dedicate this thesis to my loving parents ...

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Stanislas Brossette
November 2015

Acknowledgements

And I would like to acknowledge ...

Abstract

This is where you write your abstract ...

Table of contents

List of figures

List of tables

Chapter 1

Numerical Optimization

1.1 Introduction

In modern science, optimization has a very important place. Mechanical engineers optimize the shape of stuctural parts. Investors optimize the profit of a portfolio while minimizing the risks. Chemists optimize the efficiency and speed of reactions. When it comes to robotics, optimization is everywhere. From the design of a robot to its actuation. Any positioning of a robot requires to compute the articular parameters of each joint of the robot, finding such parameters might be possible by using analytical methods for very simple robots, but for robots as complex as humanoid robots, it is definitely not possible. And most often, an optimization process is used.

The goal of an optimization algorithm is to find an optimal solution to a problem. Optimal in the sense that the solution is an optimum of a given objective function. And solution of a problem in the sense that is satisfies a set of constraints defined by the problem. Both the constraints and the objective function are defined on the variable space, which is the space in which we search a solution, the space in which the variable lives.

In this chapter, we will use the following notations:

- $\mathscr{S}$ is the variable space (usually $\mathbb{R}^n$)
- $x \in \mathscr{S}$ is the vector of variables
- $f : \mathscr{S} \to \mathbb{R}$ is the objective function, or Cost function
- $c_i : \mathscr{S} \to \mathbb{R}$ is the i-th constraint function ($0 \leq i \leq m$)

Using these notations, an optimization problem can be written as follows:

$$\begin{aligned} &\min_{\mathbf{x}\in\mathscr{S}} \quad f(\mathbf{x}) \\ &\text{s.t.} \begin{cases} c_i = 0 & \forall i \in \mathscr{E} \\ c_i \leq 0 & \forall i \in \mathscr{I} \end{cases} \end{aligned} \tag{1.1}$$

We call $\mathscr{E}$ the set of index for which the constraints are equality conditions and $\mathscr{I}$ the set for which the constraints are inequality conditions.

This set of equations presents the optimisation process under a very general form. In order to present the principles of optimization, we will consider simpler problems. The first type of problem that we will talk about in this section is the unconstrained problem, that has the particularity to not have any constraint and its variable space is $\mathbb{R}^n$. Then we will considere the same problem, but with added constraints, and we will particularly detail one specific constrained problem optimization algorithm that is called the Sequential Quadratic Program(SQP). And finally we will study the implications of solving an optimization problem on a manifold that is different from $\mathbb{R}^n$.

In this section we will present the theory of optimization methods, starting from the "simplest" unconstrained optimization and work our way up to more complex thing by adding constraints and solving problems on non-Euclidian spaces.

1.2 Unconstrained Optimization

An unconstrained optimization problem is a problem that has an objective function but no constraints. It can be formulated as follows:

$$\min_{\mathbf{x}\in\mathscr{S}} \quad f(\mathbf{x}) \tag{1.2}$$

In order to solve this problem, we want to design an algorithm that, starting from an initial guess x_0, will converge toward the solution x^*. The objective function is not necessarily completely known. In the sense that we can't always have an explicit formula, often, the function f is computed by another program that is able to compute $f(x)$ and $\nabla f(x)$ for a given value of x. In order to have an efficient algorithm, we want to avoid any unnecessary computation of $f(x)$ and its derivatives. We will denote the values taken by x along the iterations as $x_0, x_1, x_2, \ldots x_i$. And $f(x_i)$ is denoted f_i.

Since our knowledge of the objective function is only partial, it would not be possible to guarantee that a point x^* is a global solution:

$$\forall x \in \mathscr{S}, f(x^*) \leq f(x) \tag{1.3}$$

Though, we can find a local minimizer of f:

$$\text{There exist a neighborhood } \mathscr{N} \text{ of } x^* \text{ such that } \forall x \in \mathscr{N},\ f(x^*) \leq f(x) \tag{1.4}$$

That is the kind of solution that we are looking for and that our algorithms will find.

Under the assumption that the objective function is smooth and sufficiently continuous ($\mathscr{C}_2$), then we have the following sufficient conditions for the optimality of x^* as presented in [**?**]:

Theorem 1.2.1 *If $\nabla^2 f$ is continuous in an open-neighborhood of x^* and that $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is positive definite. Then x^* is a strict local minimizer of f.*

During the resolution of an optimization problem, the algorithm will generate a sequence of iterates x_k starting from the initial iterate x_0 (Which is usually provided by the user). During the step k of the optimization process, the solver, the current point is x_k and we try to find x_{k+1} such that $f(x_{k+1}) < f(x_k)$. There are several strategies to do that but we will only focus on 2 of them that are the most popular, the line-search and trust-region methods.

1.2.1 The Line-Search Strategy

In the Line-Search strategy, given a point x_k, a descent direction p_k from this point is chosen and then a length of step is calculated to minimize the following 1-dimentional problem:

$$\min_{\alpha \in \mathbb{R}^+} \quad f(x_k + \alpha.p_k) \tag{1.5}$$

Once the best value of α has been found, the next iterate is computed: $x_{k+1} = x_k + \alpha p_k$ and the same process is repeated until a satisfying solution is found.

It is not always necessary to find the optimal value of α, especially if that is expensive. Indeed, α is only used to calculate the next iterate, from which another p_k and α will be calculated. And in the end, the imprecision on the computation of α will be errased in the other steps of the resolution.

There are several ways to choose a descent direction from an iterate x_k. The most obvious one is probably the steepest descent direction $-\nabla f_k$. This method provides the direction

along which f decreases most rapidly and only requires the evaluation of the first derivative of f, but that method can become extremely slow on complicated problems. Another popular approach is the Newton method, in which the objective function is approximated to the second order

$$f(x_k + p) = f_k + p^T \nabla f_k + p^T \nabla^2 f_k p \tag{1.6}$$

Then the chosen descent direction is the optimum of that approximated function, the Newton direction.

$$p_k^N = -(\nabla^2 f_k)^{-1} \nabla f_k \tag{1.7}$$

The choice of this descent direction implies that the $\nabla^2 f_k$ is positive definite, in which case an adaptation of the definition of p_k is required. Or an approximation B_k of $\nabla^2 f_k$ that guaranties definite positiveness can be used. For example, the symmetric-rank-one(SR1) formula and the BFGS(Broyden, Fletcher, Goldfarb, Shanno) formula. Then the step becomes

$$p_k = -B_k^{-1} \nabla f_k \tag{1.8}$$

1.2.2 The Trust-Region Strategy

The Trust-Region Strategy works in an opposite way than the line-search one in the sense that during a line-search step, a direction is chosen, and based on that direction, a step-length is chosen. Whereas with a Trust-Region approach, a maximum step-length is chosen, and based on it, the descent direction and lenght are chosen. The principle of the trust region is that along the optimization process, a model of the problem is constructed and enriched at every step and at each step, the next iterate is the optimum of the model, with the constraint that the step to get there lies inside the trust-region. For example, let us considere that the trust region is a sphere of center x_k and radius ρ_k, then the constraint on p_k is $\|p_k\| \leq \rho_k$. A usual model to take for the objective function is the quadratic model with approximated Hessian

$$m_k(p) = f(x_k + p) = f_k + p^T \nabla f_k + p^T B_k p \tag{1.9}$$

And the optimization problem to solve at each step of the optimization is

$$\begin{aligned} \min_p \quad & f_k + p^T \nabla f_k + p^T B_k p \\ \text{s.t.} \quad & \|p\| \leq \rho_k \end{aligned} \tag{1.10}$$

Once the solution p_k to this quadratic problem is found, its quality is estimated by evaluating the value of $f(x_k + p_k)$. If the actual decrease of f is satisfying (compared to the decrease predicted by the model) then the step is accepted and the size of the trust-region can be increased. Otherwise, the step is refused, the trust-region radius is reduced and a new step from x_k is computed on that smaller trust region.

1.3 Constrained Optimization

1.4 Constrained Optimization on Manifolds

1.5 Conclusion

Chapter 2

Posture Generation for Humanoid Robotic Systems

Chapter 3

Chapter 3

Chapter 4

Chapter 4

Chapter 5

Point-Cloud Multi-Contact Planning for Humanoids: Preliminary Results

5.1 abstract

We present preliminary results in porting our multi-contact non-gaited motion planning framework to operate in real environments where the surroundings are acquired using an embedded camera together with a depth map sensor. We consider the robot to have no a priori knowledge of the environment, and propose a scheme to extract the information relevant for planning from an acquired point cloud. This yield the basis of an egocentric on-the-fly multi-contact planner. We then demonstrate its capacity with two simulation scenarios involving an HRP-2 robot in various environment before discussing some issues to be addressed in our quest to achieve a close loop between planning and execution in an environment explored through embedded sensors.

5.2 Introduction

Humanoid robots are expected to move and achieve tasks in ways similar to humans. In cumbersome and unstructured environments, we humans move in a non-gaited acyclic way: we choose appropriate parts of our body to create contacts with the surrounding environment in order to support the motion of the remaining parts while avoiding obstacles. A whole motion is a sequence of contact creations and releases.

Since we are biped, we mostly use our feet to move. As the environment becomes more difficult to cross, hands may come into play together with feet to help with the motion. Narrow passages may even require other parts of our body (knees, elbows, back...) to make contact in order to support the motion.

Recently, we have dedicated considerable efforts in proposing a general multi-contact motion planner to solve such cases of non-gaited acyclic planning. Given a humanoid robot, an environment, a start and a final desired postures, our planner generates a sequence of contact stances allowing any part of the humanoid to make contact with any part of the environment to achieve motion towards the goal. A typical experiment with a HRP-2 robot achieving such an acyclic motion is presented in [15], and the planner is thoroughly described in [14]. Extensions of this multi-contact planner to multi-agent robots and objects gathering locomotion and manipulation are presented in [10], and preliminary validations with some DARPA challenge scenarios, such as climbing a ladder, ingress/egress a utility car or crossing through a relatively constrained pathway are presented in [11]. In [14] and [10], we describe works in multi-contact that are achieved by other colleagues in robotics.

All of our previous works, namely the planning scenarios experimented on the HRP-2 humanoid robot, use perfect 3D models of the environment to plan motion. In fact, our previous experiments use model-based, off-line, multi-contact planning. The results are then experimented on the real environment only after a very careful calibration assuming no changes in the object's positions occur during the trials. To achieve dynamic motions that account for whole body motion, we still use off-line planning, see e.g. in [23].

Practical implementation and use of our multi-contact planner will not be a plausible option if it cannot be extended to work with data acquired directly from the robot's embedded sensors. Namely, if the environments and composing object models are known, then the robot-environment careful calibration would be ideally done by the robot continuously. In contrast, if the environments and/or composing object models are not known, then the robot must acquire them progressively and build knowledge on which multi-contact planning can still be carried out. This paper examines the possibility to extend our multi-contact planner to work with information acquired by the humanoid embedded visual system, a camera with depth information, providing 3D point cloud data. We assume that we do not have the 3D models of the environment and composing objects.

Although preliminary, our first trials show promising results. We describe the changes made on our multi-contact planner to work with partial 3D point maps, and present two case studies of successful multi-contact planning simulations on 3D point clouds with the HRP-2 humanoid robot. As a preliminary work, we dedicate a large part of our contribution for discussion.

5.3 Background and motivation

Our multi-contact planning (MCP) is made of several modules. The Fig. 5.1 illustrates the

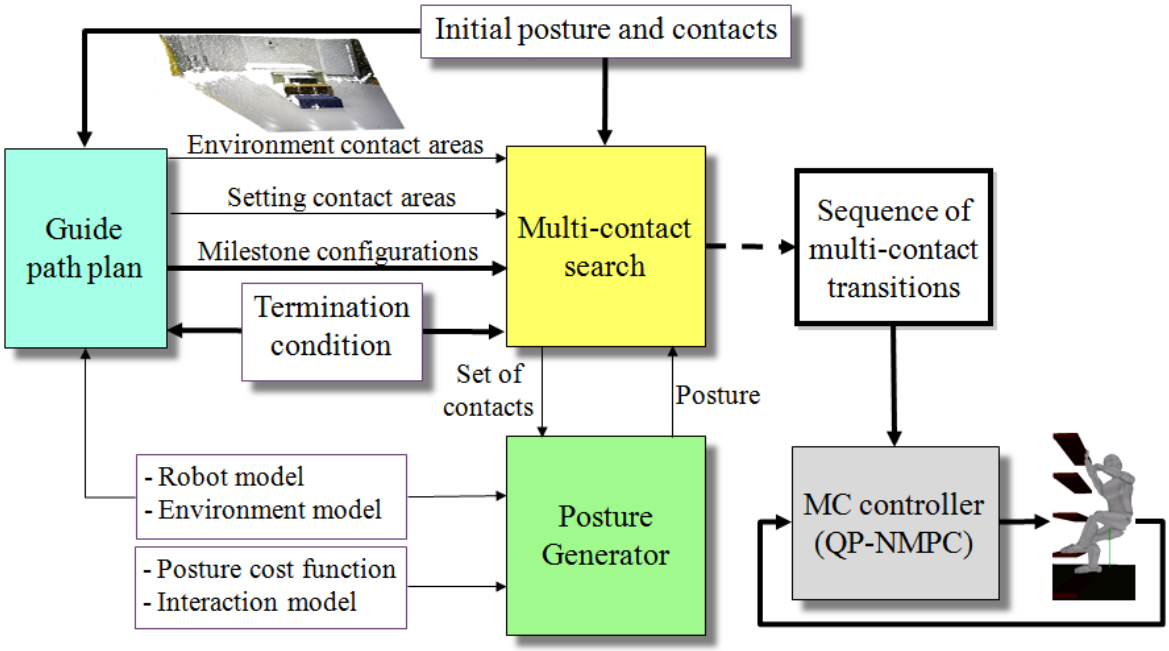

Fig. 5.1 Multi-contact planner architecture.

main components and their relation. The core of the planner consists in the multi-contact search and the posture generator. The former incrementally builds a tree of contact sets. The children of a node are obtained by adding or removing exactly one contact to its set. At each iteration of the search, the best leaf, according to a potential field, is expanded this way. The tentative sets of contacts are tested by the posture generator: for a given set of contacts, it attempts to find a configuration of the robot that satisfies these contacts while being statically stable, avoiding collision and staying within the robot's state and torques limits. This is done by automatically writing and solving a non-linear optimization problem. Upon success, the contact set is validated, and a new leaf is created. The goal is written as a set of constraints defining a sub-manifold of the configuration space. A node is the final node if its associated posture generation problem augmented by these constraints remains feasible. By backtracking from this final node to the initial root node, we obtain a sequence of nodes and thus a sequence of contact sets, that can be executed on the robot by a whole-body controller. We developed such a controller based on a quadratic programming (QP) formulation [9].

The potential field is derived from a crude path, made of a few key postures, that does not take contacts into account. Such a path is either user-defined or can be the output of a first dedicated planner [8].

The MCP relies largely on the 3D geometric models of the environment and robotic agents. In our previous work [14, 10], the geometric models are provided by the user. The contact transition for the robot are planned off-line and later executed by the robot assuming exactness of the models and their relative positioning. We aim at extending our MCP to deal directly with real data acquired by the robot. Subsequently, we must deal with two kinds of situations:

1. the models of the objects in the environment are known: in this case adapting the MCP consists mainly in dealing with recognition, model superposition and handling uncertainties. In brief, once model superposition is achieved, we can use the 3D model for MCP as in [14, 10], yet some adjustments are needed.

2. the models of the hurdles and the environment are not known (e.g. disaster or outdoor environments, for example related to the Fukushima disaster that inspired the DARPA Robotic Challenge), MCP is to be achieved in an ego-centric way with models built from the robot's embedded sensors. This paper deals with this case and we describe how the MCP is modified to achieve this goal. In a nutshell, we construct planar surfaces from the 3D point clouds data and feed them to the MCP.

In robotics, the use of 3D-based vision for recognition and navigation in environment known or partially unknown has first been used on mobile robots, evolving in flat environment, for example by coupling it with a SLAM system [39]. Another approach consists in extracting the surfaces from the point cloud, and then link them to the known environment or simply consider them as obstacles to be avoided [33]. Since working on raw point clouds is costly because of the high number of data points, this extraction has also been enhanced [6] in order to be run in real-time. This approach has been recently experimented on a humanoid robot in [25], that combines two methods: the surface extraction from a point cloud, and the voxel-based 3D decomposition of the environment [28]. Still, since the robot only navigates in a flat environment, and does not realize manipulation tasks, the surfaces extracted from the 3D point cloud are down projected to a 2D plan, on which are based the navigation and collision avoidance processes.

The use of humanoid robots allows to navigate in more complex environment, and recently, some work has been done to make a humanoid robot go down a ramp [24] or climb stairs [32]. Yet, those methods use laser-based vision rather than point-cloud-based vision, so as to have a precise analysis of a known environment.

In this work, we aim at enabling a robot to analyze and plan a motion into a 3D environment. Hence, we use the surface extraction of a point cloud to directly have a global picture of the environment and determine the convex planar surfaces that the robot can use at its advantage to progress using the MCP. In our approach to make such an extension, we intentionally seek for technical solutions that minimize changes to be done on our existing MCP software.

5.4 Building an understandable environment for our planner

Our first concern is to build an environment that our multi-contact planner is able to "understand" and that can be extracted from a point cloud scene.

The simplest entity that our planner would be able to deal with and that could correctly describe the robot's environment is a set of convex polygonal planar surfaces. Therefore, starting from an acquired point cloud, we try to extract a relevant set of such geometrical entities that will be a first description of the surroundings of the robot.

In this section, we present the different steps we follow to create a set of relevant convex polygonal plane surfaces out of an acquired point cloud.

The Fig. 5.2 illustrates the major steps of this point cloud treatment. We use Willow Garage's Point Cloud Library[1] (PCL) [34] extensively for processing the point cloud.

5.4.1 Acquisition of point cloud from a RGB and depth sensor

For our experiments, we use an Asus Xtion Pro camera, that provides a 640×480 pixel depth image, and the OpenNI Grabber. The obtained point cloud representing our scene contains points defined by their space coordinates and colors, and will be the entry of our point cloud treatment algorithm. We do not use the color information for now except for display purpose. It may however be useful in the future in matching object models with sensor data and to perform color-based region growing algorithms.

5.4.2 Filtering

In order to reduce the computation time and improve the efficiency of our point cloud treatment algorithm, it is necessary to reduce the overall size of the data set. We first remove

[1] http://pointclouds.org/

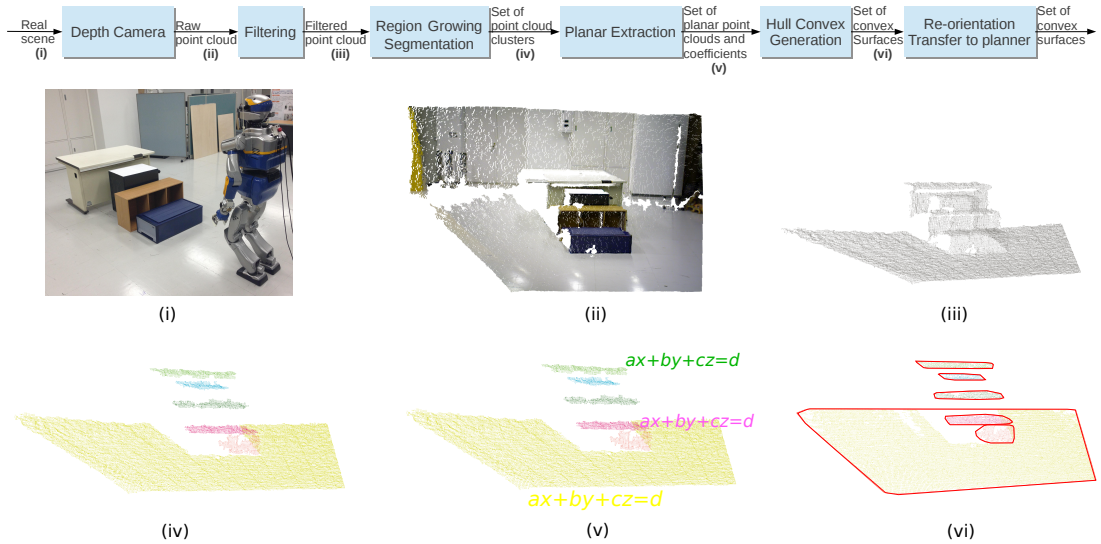

Fig. 5.2 Top: flowchart describing the main elements of our algorithm and the type of data that is passed between them.

Bottom: the data throughout the process, illustrated in the case of our first experiment (see Sec. 5.6.1).

all points further than 5 meters from the camera: such points are not reliable enough and thus a potential source of error for the following steps.

We then downsample the cloud, since it is excessively dense for our purpose. To do so, we use a voxelized grid approach (using the `pcl::VoxelGrid` class): we create a 3D voxel grid over the input point cloud data. Each voxel represents a group of points that are close enough from each other to be represented by a single point that would be their centroid. The centroid is chosen instead of the center of the voxel because it represents the real environment more accurately. Another advantage of this method is that the filter can easily be parametrized by choosing the size of the voxels. Those two steps allow us to greatly reduce the number of points in the data set. Typically, in our experiments, this divides the number of points by a factor 5 to 6.

5.4.3 Region growing segmentation

In order to divide a global point cloud scene into several sets of points that belong to the same flat area, we use a region growing algorithm [33] (implemented in `pcl::RegionGrowing` class). Its purpose is to merge the neighboring points that are close enough in terms of smoothness constraint. The output of this algorithm is a set of clusters representing sets of points that are considered to be a part of the same plane surface. This method is based

on the comparison of the angles between the points' normals. It is used right before the planar extraction algorithm in order to obtain an accurate list of points and plane models they represent.

5.4.4 Planar extraction

We use a plane segmentation to extract and group all the points that support a same plane model. Thus, we obtain a set of distinct point clouds. Indeed, this method finds the plane model that can carry the biggest set of points without regard for their proximity to each other. Then this first set of points is extracted from the initial point cloud and the algorithm can be applied to the remaining cloud recursively until the remaining cloud is small enough (in terms of number of points). For this purpose, we use the `pcl::SACSegmentation` class.

This process works better on mostly flat point clouds (composed of a small number of large plane surfaces). Indeed, if it is applied to a point cloud that represents some stairs, instead of finding a set of points and a plane model for each step, it will most likely find a set of points and a plane model that represent the average slope of the stairs, and the reason for that is that the criteria of choice of a plane model is the greater number of points it bears, no importance is given to the proximity between the points of a planar set. That is why we apply the region growing algorithm 5.4.3, that provides a list of clusters that represent the flat surfaces of the scene, before applying the planar extraction process on each of those clusters. It then computes a correct set of points and a plane model per region cluster.

5.4.5 Planar projection and hull convex generation

The exact knowledge of all the data points contained in the plane model of a flat area is not necessary for our planning; we can reduce the point cloud to its convex hull without loss of information (except if the surface is concave, but this issue will be tackled in future works). In order to obtain the convex hull of each set of points, we first project every point of the set on its plane model (`pcl::ProjectInliers` class) and then compute the 2D convex hull of the projected set of points (`pcl::ConvexHull` class). After this step, each plane surface of the scene is represented by a frame composed of 3 vectors that are respectively a tangent, bi-tangent and normal vector of the plane model, an origin point that is the barycentre of the set of points and a small set of points that define its convex hull (the points' coordinates being calculated in the referential defined by the frame and the origin of the surface in order to make further calculations easier).

5.4.6 Re-orientation and transfer to the planner

Before sending the previously computed data to the planner, it is necessary to take into account the initial orientation of the camera. Indeed, if the camera was not aiming in a perfectly horizontal direction, then the entire point cloud would be misoriented. Therefore, it is necessary to re-orientate the surfaces before sending them to the planner. To do so, we simply apply a rotation matrix (that is computed from the initial camera orientation) to each of our data set's frame and origin to settle that problem. From there, the transfer of the surfaces to the planner can be done without any specific issue.

Re-orientation is done as a last step for the sake of performance: obviously we need to re-orient only a few frames, compared to an early re-orientation of the point cloud that would require to apply a transformation on thousands of points.

5.5 Planning on point clouds

5.5.1 Convex surfaces inclusions

We adjusted slightly our planner and posture generator to handle contacts between convex polygonal plane surfaces. The main modification made in the posture generator deals with properly writing the constraints that enforce the inclusion of one surface into another one. In our previous implementation, contacts are searched between rectangular patches attached to the robot body or the environment. Now any polygonal convex-shaped patch can be checked for inclusion in another; that is to say, the vertices of a convex 2D polygon are indexed counter-clockwise around its carrying plane surface's normal vector. A point that is inside the polygon is on the left side of all its edges. On the opposite, a point that is outside of the polygon will be on the right side of at least one of the polygon's edges. The following is the algorithm we use for the constraint generation:

Algorithm 1 Surface inclusion constraints for $S_i \subset S_j$

Let S_i and S_j be two coplanar plane surfaces
$S_i = p_0, p_1, \ldots, p_n$ and $S_j = q_0, q_1, \ldots, q_m$
$\vec{N}$ is S_i's normal vector
for $k = 0 \rightarrow n$ **do**
  for $l = 0 \rightarrow m$ **do**
    Constraint : $[\overrightarrow{q_l p_k} \times \overrightarrow{q_l q_{l+1}}] . \vec{N} \leq 0$
  end for
end for

For a couple of coplanar surfaces S_i, S_j respectively represented by n and m points, we create $n \times m$ constraints to ensure that the former is included in the latter one.

Once the surfaces are defined, it is possible to choose which ones are suitable for the robot to make contact with. Although it is not a mandatory step, it allows to reduce the exploration during the planning phase by removing undesired or inappropriate pairs of robot/environment surfaces. For the time being, this is determined by heuristics that are defined depending on the situation. For example, if we want the robot to walk on various surfaces, only surfaces that have a normal vector closely aligned with the gravity field would be selected as potential candidates (so as to eliminate the walls and other surfaces on which the robot cannot walk). Similarly, only surfaces located at a certain height can be considered for hand contact, etc.

5.5.2 Collision detection

In order to generate a feasible plan, we need to ensure that the robot avoids collisions with its environment and with itself. To do so, we consider each surface generated by our point cloud treatment algorithm as a thin 3D body. Basically, we extrude each surface by few centimetres in the direction opposite to its normal (provided that this normal is pointing toward the outside of the real body) and create a convex hull surface using `QHull` [4]. The collision avoidance is then computed by using the `GJK` algorithm implemented efficiently for several convex shapes in [5].

5.6 Results

In order to illustrate our method, we present two experiments in which the HRP-2 robot is asked to move 2m forward. In both scenarios, this results in climbing on a table (The first one is 0.71m-high and the second one is 0.53m-high) with the help of various surrounding objects. The knowledge about the environment and surrounding objects is obtained from a point cloud captured by an ASUS Pro Xtion camera in one single shot. The camera was placed at the height of the robot's head, see Fig. 5.2.

All the computations of the following experiments are performed on a single thread of an Intel(R) Core(TM) i7-3840QM CPU at 2.80GHz, with 16Go of RAM.

5.6.1 Plan 1: irregular stairs

In this first experiment, the robot has to walk up some irregular stairs made of several random pieces of furniture to reach its goal. The filtered point cloud contained 60286 points, that were split into 6 plane surfaces. The whole cloud processing was done in 2.7 s. The planner

computations generates a path of 11 nodes, some of which are depicted in Fig. 5.3. We notice that the robot climbs the stairs one by one without ever putting its two feet on the same step and without any noticeable problem. In total, 23 nodes were generated and the planning time was 98.4 s.

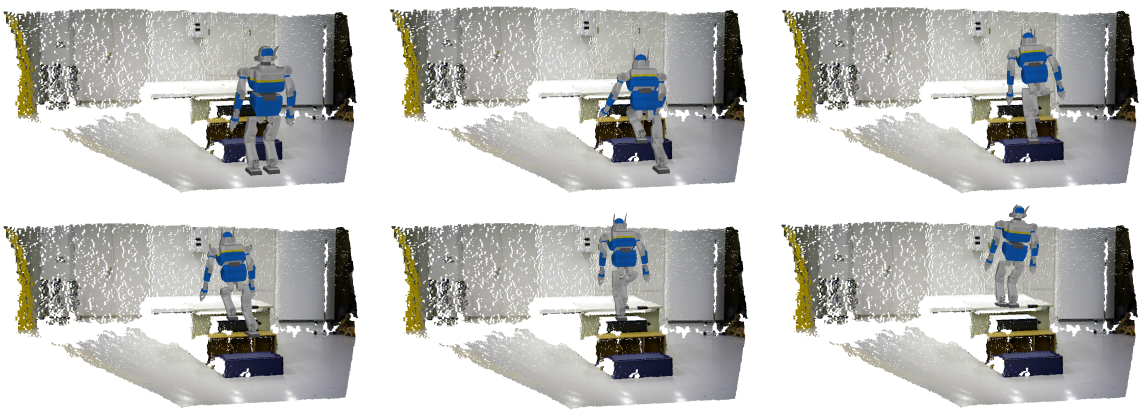

Fig. 5.3 Table climbing simulation using irregular stairs. Of the 11 nodes of the path, we depicted the nodes 1, 4, 6, 8, 10 and 11.

5.6.2 Plan 2: helping motion with the hand

This second experiment was designed to showcase a more complex plan involving the use of the HRP-2 upper limbs. In this experiment, the filtered point cloud contained 66907 points, from which 11 surfaces were extracted. The whole cloud processing was done in 2.7 s. The planner computation generates a path of 19 nodes, some of which are depicted in Fig. 5.4. To climb on the table, the robot uses its left arm and walks on an inclined slope before climbing a step at the end of the slope, once again, with the help of its left arm as support. In total, 40 nodes were generated and the planning time was 122.3 s.

5.7 Discussion

This work is a first step toward a fully sensory-perception-based multi-contact planner. It raises several interesting questions on the way to adapt our MCP.

One advantage of our approach is to avoid having to precisely position the robot in the environment prior to the plan execution. Yet, for now the positioning is done once and for all with the acquisition of the point cloud, before planning. When executing the plan, the robot might still deviate from it, for example a support might move, or a foot might contact a few

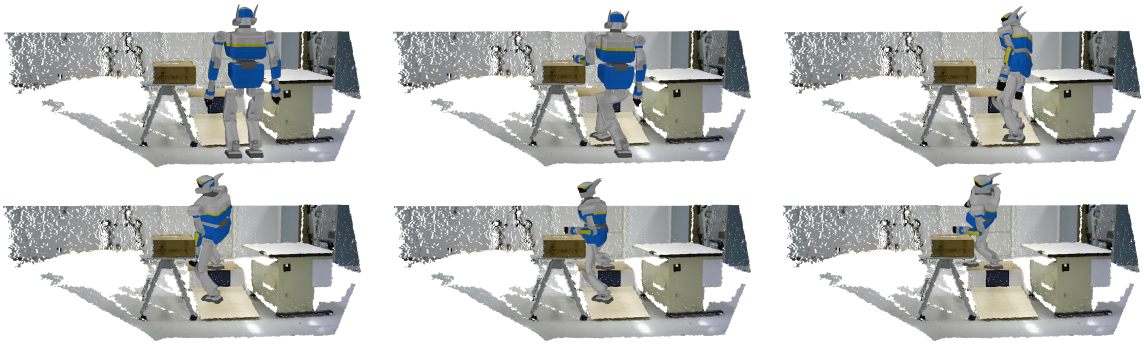

Fig. 5.4 Slope and step climbing simulation. Of the 19 nodes of the path, we depicted the nodes 1, 7, 12, 15, 17 and 18.

centimetres away from where it was planned to, or the support might not be at its expected position because of measurement errors in the acquired point cloud. We thus need to close the loop between the planning and its execution. To do so, we need robust detection of discrepancies to be made by the robot. This can be achieved by combining point-cloud-based SLAM with contact sensing. Once the robot has acquired knowledge of its deviation or of the position error of the support, it has to adapt to it. This adaptation should not be time-consuming so as to not interrupt the execution for too long. A slight deviation can be recovered by simply positioning with care the next contacts and the closed-loop multi-contact controller shall work on guarded-motions basis. However, a bigger deviation might make the next contact stances infeasible. In this later case, re-planning the next contacts is necessary to go back to the plan. How many contacts have to be re-planned depends on the context. In difficult situations, changes in contacts might cascade up to requiring an entire re-planning. Recognizing the situation should be the task of a local planner that re-plans as few steps as possible. In case too much contacts must be reprocessed, the re-planning phase can be stopped before it ends and resumed at the next step.

This partial planning approach can also be seen in the context of semi-autonomous motion: an operator gives the overall direction with, for example, a joystick, and the planner reactively finds a sequence of a few contact sets to move as closely as possible in this desired direction. The operator is thus in charge of preventing the robot from getting stuck while the planner only concentrates on finding the correct contacts over a short time window.

Another question stems from the partial knowledge of the environment: it is not possible to give a guide path as we used to do with the 3D models. This guide path will necessarily be very crude, either a line to a desired position, or a plan in a known environment before it was changed (for example in the case of a disaster in a plant). The planning must then be driven also by the need of getting information about the environment, for example reaching a

viewpoint allowing to see parts of the environment that were hidden before, filling empty spaces and possibly adding new supports on-the-fly. Planning is then only partial since necessary part of the environment might be unknown.

Later on, the discovery of the environment might be improved by the use of other sensors. One can then imagine having the robot test for a contact to ensure a given surface is fit for support or that it is precisely at the position measured by vision. If the surface is not, this is another kind of discrepancy in the plan than needs to be handled by re-planning.

5.8 Conclusion

We present preliminary results in extending our previous work in multi-contact planning to operate directly on environment that is built on-the-fly from a robot's embedded sensors. This first study makes use of depth camera and implemented modules from the PCL which extract planar surfaces that are fed to our MCP. We intentionally seek for technical implementations that minimize the changes on our existing MCP software and illustrate successful plans generated on the basis of 3D point clouds solely.

The simulation results revealed that indeed our MCP does not require major adjustments to handle egocentric sensory data. Of course, this does not mean that we are fully satisfied since our results also suggest additional future work.

First, although we choose to treat the case of not having 3D models, we believe that the implementation of an MCP with knowledge of the 3D model is necessary. For example, even in a disaster situations as in Fukushima nuclear power plants, the inside exploration videos available show that many objects kept their shape and were not totally destroyed (e.g. door, stairs, ladders, etc.). So having their model would then still permit our MCP to rely on 3D models to plan contacts for motion. PCL provides only partial information, it is then necessary to drive the planner by the mission objective and also a perceptual one (e.g. SLAM). Of course, our ultimate future plan is to handle uncertainties, control and planning recovery of discrepancies when they occur.

Chapter 6

Integration of Non-Inclusive Contacts in Posture Generation

6.1 abstract

In this paper we propose a simple way to formulate geometric contact formation to have an arbitrary intersection shape in a robotic (humanoid) posture generation problem. The contact shape is the outcome of our posture generator that is formulated as a non-linear optimization programming to fulfill a large variety of robot intrinsic limitations (e.g. joint and torque limits) and tasks (e.g. desired contact). Starting by defining convex areas of contact on the robot's body and the environment, that we call contact patches, we can generate contacts with arbitrary intersection of a pair of any of these predefined patches. Our geometric contact modeling writes very simply as additional constraints and variables added to the optimization problem, translating the search for an ellipse inscribed in the intersection of the pair of patches we want in contact. The result of our posture generator is then a configuration where contact patches are not necessarily included in one another. This allows our posture generator to propose contacts of different shapes with a non-predefined number of contact points (used later to compute reaction/contact forces). We illustrate the efficiency of our method in multi-contact posture generation with the HRP-2 and ATLAS humanoid robots with results that can not be generated automatically by existing methods.

6.2 Introduction

Generating viable robotic postures is a common problem encountered in sampling-based planning techniques and simulation of virtual characters. Generating desired initial, interme-

diary or finale posture configurations requires defining static task goals (e.g. reach a target point in 6D) to be done under intrinsic constraints such as joint limits, torque limits, avoiding non-desired self-collisions... and perceptual or extrinsic ones such as keeping an object in the embedded camera field-of-view, avoiding non-desired collisions with surrounding objects, etc. A common task objective assigned to virtual characters, avatars, or humanoid robots is to contact one or more of its links with the environment (e.g. feet touching the ground). Since our main applications target humanoid robots, we consider here posture generation problems inherent to these robots. However, the formalism applies to any kind of robots achieving contacts with its surroundings or itself.

Our problem is to generate multi-contact viable postures. As far as humanoids are concerned, we may add to the previously cited constraints, equilibrium and non-sliding. This paper is dedicated to the focused issue of how contact constraints can be written geometrically. Generating postures often uses state-of-the-art enhanced inverse kinematics or general-purpose non-linear optimization programming (where inverse kinematics can be seen as a particular solver case).

In general, desired contacts write as hard constraints to fulfill in an optimization problem. Yet, we need to write the maths for, say, put the gripper on the wall and the left foot on the ground. In general, the maths of a gripper is a complex geometric description, so is often that of the environment. A contact is generally defined by a pair of points (one on each object in contact) and a pair of normal vectors. A hard contact constraint boils down to finding a posture in which the predefined authorized contact points and normal of each body match [40][21]. Likewise, in [31], the position of the feet of the NAO robot is manually tuned in order to obtain statically stable position during the climbing of a spiral staircase. In [13], the surface in contact is chosen according to two criteria: the position of the force sensors of the feet, and the type of contact desired. In [36], the problem of contact discovery in not considered. In [26], two surfaces are considered in contact as soon as the center point of one of them touches the other one and their normals match. Although used in many papers, it is not difficult to see that this definition of contact excludes a series of possibilities that would have been obtained if the predefined points were placed in different configurations within their respective patches. Once the contact is established, one determines the intersection of contacting surfaces in order to find points on which reaction forces are to be computed. Several approaches require fixing the number of contact points or to have inclusive contact (i.e. one patch is fully included in the other) [10].

We provide a simple solution that relaxes hard contact constraints and gets rid of pre-defining the contact points by allowing them to travel within the patches. We consider that a contact is valid if the intersection between two distinct patches has an area greater

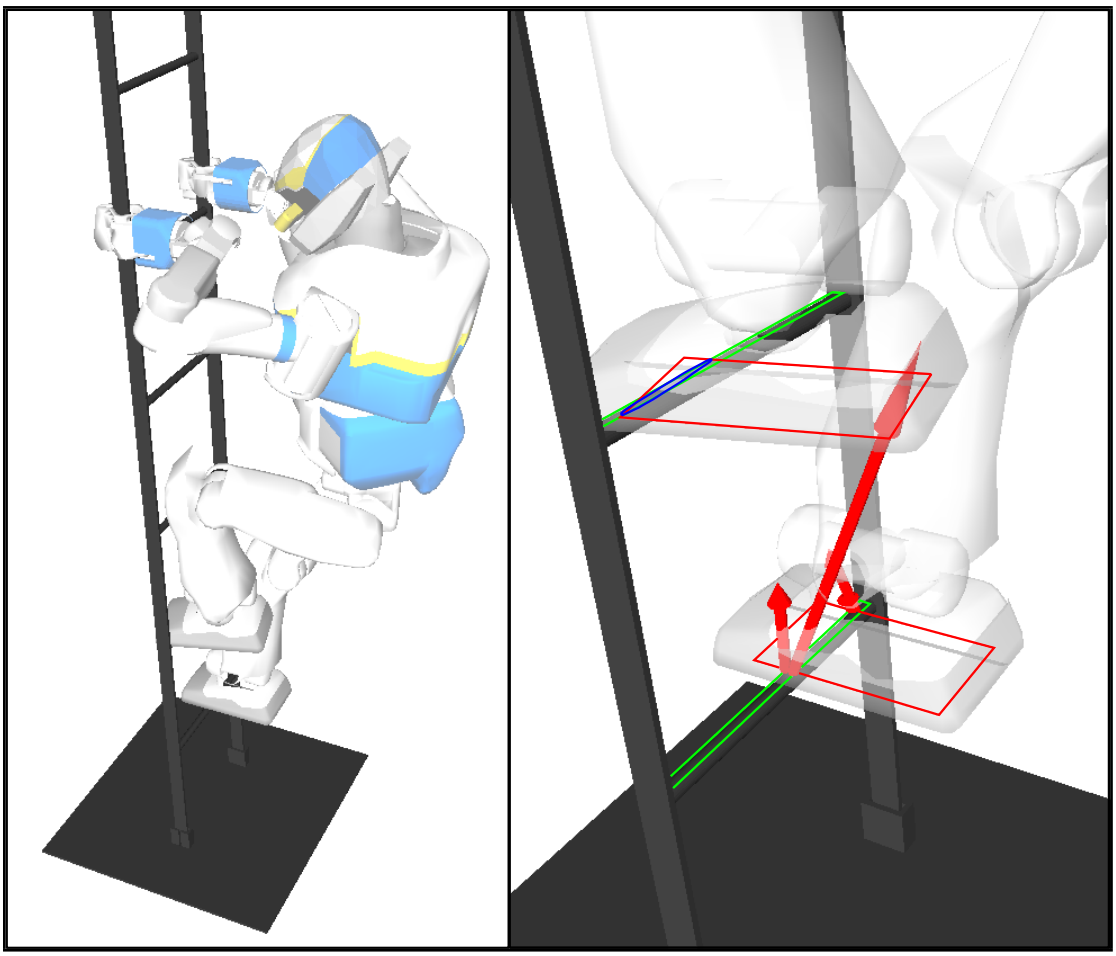

Fig. 6.1 Using non-inclusive contacts for ladder climbing (green/red: contact polygons; blue: contact ellipse; red arrows: contact forces resultants)

than a given threshold. To enforce this, we require this intersection to contain an ellipse whose surface can possibly be maximized. Convex patches allow writing the inscription constraints easily by means of half-spaces. We finally implement our solution and present some examples for complex multi-contact posture generations with the HRP-2 and ATLAS robots, as shown in Fig. 6.1.

6.3 Contact geometry formulation

For our formulation, we consider a situation where a set of contacts between the robot and its environment are already made, and therefore fixed, and we want to add a new contact to this set. This doesn't induce any loss of generality since it just comes down to adding the contacts one by one. To ensure that the new contact can be reached in a quasi-static way, we look for a configuration where the new contact is "barely" made: the position of contact is reached, but that contact does not support any contact forces (this is a necessary step to generate a sequence of quasi-static transitions). We call it a geometric contact. Let us consider that the contact to add is defined by two flat surfaces S_1 and S_2 which are respectively delimited by two convex polygons P_1 and P_2. For this contact to be valid, it is obviously necessary that the intersection $P_1 \cap P_2$ is not empty. We propose a method in which the size of the contact area is approximated by the size of an ellipse that is inscribed in it. If such an ellipse is found and is of a sufficient size, then the contact is valid. This allows to consider contacts between surfaces that do not necessarily include each other.

An important remark is that the number of sides of the intersection polygon is not known a priori and, as shown on Fig. 6.2, this number can change depending on the configuration. Each time this number changes, the gradient of the area of the intersection is discontinuous. This is an issue for integrating any constraint or objective based on the area because we use a solver for smooth optimization problems. This issue could be dealt with by using non-smooth optimization routines, but such algorithms are slower and less available, and our posture generator is not designed to use them. Moreover, supposing that we want to write constraints based on the sides of the contact area, then, the number of constraints would change with the number of vertex of $P_1 \cap P_2$. The large majority of the optimization softwares cannot deal with a non-constant number of constraints. The solution proposed in section 6.5 overcomes these issues by defining a set of constraints that is independent from the topology of the intersection area.

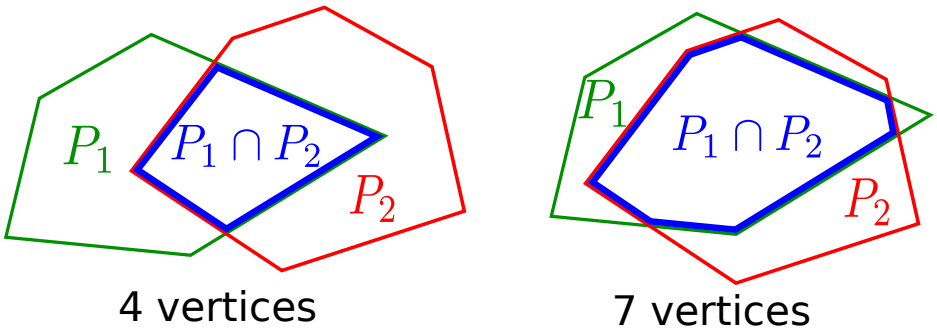

Fig. 6.2 Topological instability of $P_1 \cap P_2$

6.4 Posture generation

The posture generation process aims at finding a posture that satisfies a set of tasks $\{\mathscr{T}_i\}$ and that minimizes a cost function *Cost* by solving the following problem:

$$\min_{\mathbf{q},\mathbf{f},\tau} \quad Cost(\mathbf{q},\mathbf{f},\tau)$$
$$\text{s.t.} \begin{cases} q_i^- \leq q_i \leq q_i^+ \ \ \forall i = 1,...,n \\ \tau_i^- \leq \tau_i \leq \tau_i^+ \ \ \forall i = 1,...,n \\ \varepsilon_{ij} \leq d(r_i(\mathbf{q}), r_j(\mathbf{q})), \ \ \forall (i,j) \in \mathscr{I}_{auto} \\ \varepsilon_{ik} \leq d(r_i(\mathbf{q}, O_k)), \ \ \forall (i,k) \in \mathscr{I}_{coll} \\ \tau + J(\mathbf{q})^T \mathbf{f} = \mathbf{g}(\mathbf{q}) \\ s(\mathbf{q},\mathbf{f}) \leq 0, \\ g_i(\mathbf{q},\mathbf{f},\tau) = 0 \ \ \forall \mathscr{T}_i, \\ h_i(\mathbf{q},\mathbf{f},\tau) \leq 0 \ \ \forall \mathscr{T}_i. \end{cases} \tag{6.1}$$

where the optimization variables $\mathbf{q}$, $\mathbf{f}$ and τ stand for the configuration, contact forces and joint torques of the robot. Those constraints are illustrated in Fig. 6.3 and are, in order of appearance:

- Joint limits
- Torque limits
- Auto-collisions, with $d(X,Y)$ the signed distance between objects X and Y and $r_i(\mathbf{q})$ is the i-th body of the robot at configuration $\mathbf{q}$. $\mathscr{I}_{auto}$ is the set of pairs of bodies to monitor.
- Collisions with the environment, O_k being the k-th object in the environment and $\mathscr{I}_{coll}$ the set of pairs to monitor

- Equation of static stability, with J the Jacobian matrix of all points where the contact forces are applied, and $\mathbf{g}$ the gravity term.
- Stability constraints describing the friction cones
- Equality constraints describing the task $\mathscr{T}_i$
- Inequality constraints describing the task $\mathscr{T}_i$

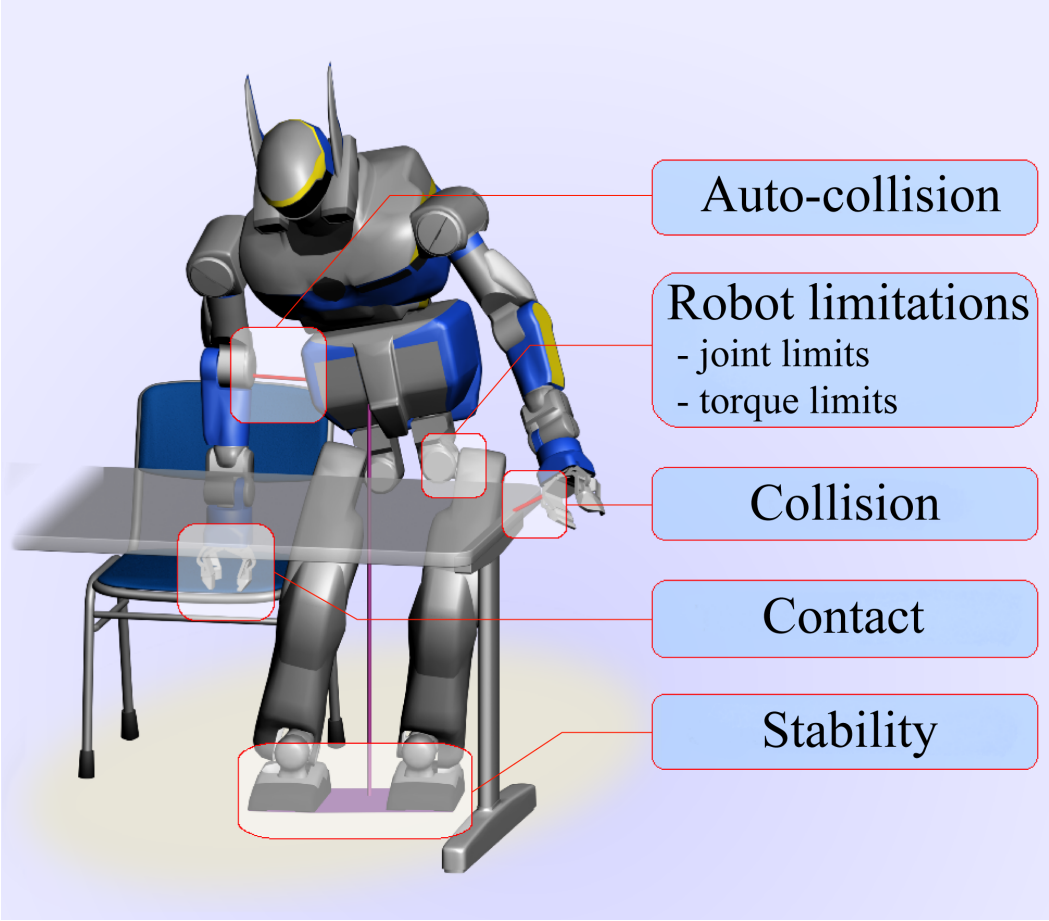

Fig. 6.3 Posture Generation's usual constraints

A usual task $\mathscr{T}_i$ in the posture generation is to ensure that two surfaces are in contact. We consider 2 polygons P_1 and P_2 described in 2 frames $\mathscr{F}_1$ and $\mathscr{F}_2$, respectively. Each frame is defined by an origin point $\mathbf{O_i} = (o_i^x, o_i^y, o_i^z)$ and 3 orthogonal vectors $[\mathbf{T_i}, \mathbf{B_i}, \mathbf{N_i}]$. In those frames, the polygons are described as a set of n_i bi-dimensional points $[p_i^0, p_i^1, \ldots, p_i^{n_i-1}]$ lying in the $[\mathbf{O_i}, \mathbf{T_i}, \mathbf{B_i}]$ plane. To ensure the contact between those two surfaces, it is necessary that the planes defined by $[\mathbf{O_1}, \mathbf{T_1}, \mathbf{B_1}]$ and $[\mathbf{O_2}, \mathbf{T_2}, \mathbf{B_2}]$ are coplanar. This is expressed by the

set of equations (6.2). Those equations define a floating contact, where the co-planarity is ensured and the surfaces can translate along $\mathbf{T_1}$ and $\mathbf{B_1}$ and rotate around $\mathbf{N_1}$.

$$\begin{cases} (\mathbf{O_2} - \mathbf{O_1}).\mathbf{N_1} = 0 \\ \mathbf{T_2}.\mathbf{N_1} = 0 \\ \mathbf{B_2}.\mathbf{N_1} = 0 \\ -\mathbf{N_2}.\mathbf{N_1} \leq 0 \end{cases} \tag{6.2}$$

However, this is not sufficient since nothing ensures that the intersection of P_1 and P_2 is not empty. Up to now, to avoid the problems described in section 6.3, we were requiring that one of the two polygons was completely included into the other (which restricted the contact configuration possibilities) or by defining a fixed contact position by hand, in which case the non-emptiness is ensured by the user. The next section presents a more general formulation

6.5 Non inclusive contact constraints

6.5.1 Main Idea

We present our main contribution: a smooth formulation of the non empty intersection between two contact surfaces. We assume that co-planarity of S_1 and S_2 is obtained by using the constraints presented in the previous section. Here we focus on the intersection of the two polygons P_1 and P_2, respectively describing the contours of S_1 and S_2. As we pointed out earlier, a problem with computing the area of intersection of two polygons comes from the fact that depending on their positions in space, the number of edges of their intersection can change (cf. Fig. 6.2), which induces discontinuity of the gradient of the area and change of the number of constraints associated with this contact.

To avoid dealing with these changes of topology, we consider using an ellipse $\mathscr{E}$ included in $P_1 \cap P_2$ to estimate the area of the intersection. Since P_1 and P_2 are convex polygons, then $P_1 \cap P_2$ is also a convex polygon. A convex polygon can be seen as an intersection of half-planes based on the lines supporting its edges. Thus, an ellipse is inside a convex polygon if it lies entirely in the corresponding half-planes. Having the ellipse be included in the intersection of two polygons is equivalent to having it included in both polygons:

$$\mathscr{E} \subset P_1 \cap P_2 \iff \mathscr{E} \subset P_1 \ \wedge \ \mathscr{E} \subset P_2 \tag{6.3}$$

Even if the number of edges of $P_1 \cap P_2$ can change, the numbers of edges of P_1 and P_2 respectively are fixed. To assert that an ellipse lies in a half-plane, we need a function that

is positive when the ellipse is in it (with zero value when the ellipse is on the edge) and negative if not. The signed distance to the line defining the half-space is a good candidate (distance ellipse-line if the ellipse is in the half-plane, opposite of the penetration distance if not), but actually, any pseudo-distance does the job. And a sufficient condition for the ellipse to be inside the polygons intersection is that the pseudo-distance between the ellipse and each edge of both polygons is positive (see Fig. 6.4). By considering each edge separately as opposed to the (pseudo-)distance of the ellipse to a whole polygon, we can write smooth constraints with a simple pseudo-distance function. We develop such a pseudo-distance in the next subsection.

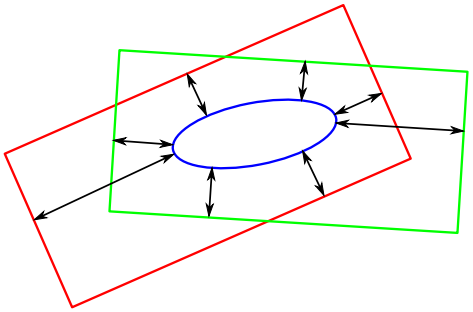

Fig. 6.4 Distance between $\mathscr{E}$ and $P_1 \cap P_2$

6.5.2 Pseudo-distance

To estimate the constraint of inclusion of the ellipse $\mathscr{E}$ in both polygons P_1 and P_2, we need to compute the signed distance between $\mathscr{E}$ and each segment of the polygons. Computing the distance between an ellipse and a line is not straightforward, whereas the distance between a line and a circle is very easy to compute. Also, we note that in the frame $F_{\mathscr{E}}$ defined by the ellipse's axes and pseudo-radius, the ellipse is a circle of radius $r_{\mathscr{E}} = 1$ (The x-unit along the first axis of the ellipse is r_x, the first radius of the ellipse, the y-unit along the second axis of the ellipse is r_y, the second radius). The transformation from the original frame F_0 in which the ellipse and the polygons are described to the ellipse's frame $F_{\mathscr{E}}$ is just the composition of a rotation and a scaling of the space along the axes of the ellipse with a scaling vector $[\frac{1}{r_x}, \frac{1}{r_y}]$. The effect of such a transformation applied to an ellipse and two polygons is shown in Fig. 6.5. We thus defined the following pseudo-distance from an ellipse to a half-plane as the signed Euclidean distance from the corresponding unit circle to the transformed half-plane in the frame $F_{\mathscr{E}}$.

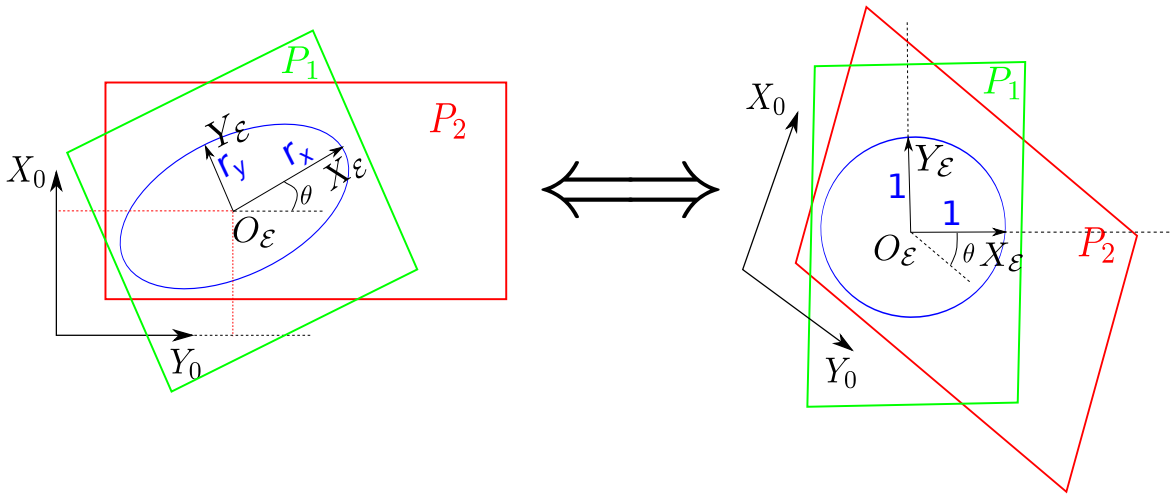

Fig. 6.5 Transformation from $F_0(O_0, X_0, Y_0)$ to $F_\mathscr{E}(O_\mathscr{E}, X_\mathscr{E}, Y_\mathscr{E})$

Now let us consider a single segment $p_i p_j$ and an ellipse $\mathscr{E}$ defined in F_0. The expression of a vector $\mathbf{v}_{F_0}^T = [v_x, v_y]_{F_0}$ in $F_\mathscr{E}$ is obtained by applying the formula (6.4)

$$\mathbf{v}_{F_\mathscr{E}} = \begin{pmatrix} \frac{1}{r_x} & 0 \\ 0 & \frac{1}{r_y} \end{pmatrix} . \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} . \mathbf{v}_{F_0} \tag{6.4}$$

In $F_\mathscr{E}$, the distance between the circumference of $\mathscr{E}$ and the segment $p_i p_j$ is:

$$d_s(\mathscr{E}, p_i p_j) = \frac{\left(\overrightarrow{p_i p_j}\right)_{F_\mathscr{E}} \times \left(\overrightarrow{p_i O_\mathscr{E}}\right)_{F_\mathscr{E}}}{\|\overrightarrow{p_i p_j}\|_{F_\mathscr{E}}} - r_\mathscr{E} \tag{6.5}$$

where $\times$ denotes the cross product and $O_\mathscr{E}$ is the center of the ellipse.

To avoid numerical problems when the segment $p_i p_j$ is small, (see sec. 6.5.6), it is preferable to multiply this distance by $\|\overrightarrow{p_i p_j}\|_{F_\mathscr{E}}$ before using it as a constraint. Then we get the following constraint:

$$-\left(\overrightarrow{p_i p_j}\right)_{F_\mathscr{E}} \times \left(\overrightarrow{p_i O_\mathscr{E}}\right)_{F_\mathscr{E}} + \|\overrightarrow{p_i p_j}\|_{F_\mathscr{E}} r_\mathscr{E} \leq 0 \tag{6.6}$$

The combination of these equations (6.4) and (6.6) applied for each edge of the polygons gives us all the necessary tools to develop a set of constraints that ensures that an ellipse is in the intersection of two polygons.

6.5.3 Modification of the optimization problem

To include the above idea in our posture generation, we need to modify the optimization problem (6.1) as follows. Each non inclusive geometrical contact adds five variables to the optimization vector, corresponding to the position, orientation and radiuses of the ellipse (x, y, θ, r_x and r_y). One constraint of ellipse inclusion (as described above) is added to the problem for each edge of the polygons. The parameters r_x and r_y are given lower positive bounds to ensure that the ellipse is not empty. The existence of a contact between S_1 and S_2 is thus transformed into the existence of r_x and r_y respecting their bounds. In summary, this kind of constraint adds 5 variables and $card(P_1) + card(P_2)$ constraints to the optimization problem, while the "usual" inclusion constraint adds 0 variable and $card(P_1)card(P_2)$ constraints. The existence of the contact can alternatively be enforced by imposing a minimum area for the ellipse.

6.5.4 Maximization of the contact area

The formulation in the above section only ensures the existence of a contact of minimal size. However, one could want to make sure to find a contact area as large as possible, so that it is more likely to be able to support strong forces and ensure strong friction forces, which is helpful to ensure the stability of the robot. Therefore, it seems appropriate to try and maximize the area of contact between two polygons. As explained before, computing the area of the intersection surface is not a good practice in our case. But we know that the ellipse computed as above gives a lower bound of the contact area.

$$\mathscr{E} \subset P_1 \cap P_2 \Longrightarrow \mathscr{A}(\mathscr{E}) \leq \mathscr{A}(P_1 \cap P_2) \tag{6.7}$$

with $\mathscr{A}(X)$ being the area of X.
Therefore we can maximize the size of the ellipse in order to maximize the contact area. This is readily obtained by minimizing the value $Cost = -\pi r_x r_y$ in the modified problem (6.1). In case there are other cost functions, the above cost can be added to them with a desired weight. This requires however to scale properly the cost so as to have a meaningful and easy-to-tune weight: the range of value of the ellipse's area goes from 0 to $\mathscr{A}(P_1 \cap P_2) \leq \min(\mathscr{A}(P_1), \mathscr{A}(P_2))$. This latter quantity can be small (a typical area of contact of a humanoid robot is about $0.01m^2$, some environment surfaces can be smaller). To get a basic cost (before weighting) of magnitude around 1, we use the following scaling:

$$\text{cost}(\mathscr{E}) = -\frac{\pi r_x r_y}{\min(\mathscr{A}(P_1), \mathscr{A}(P_2))} \tag{6.8}$$

This cost's absolute value will always be less than 1, but not much less around the optimum, in most cases.

6.5.5 Using a non inclusive contact to maintain stability

The method we presented so far allows finding a configuration in which a new non-inclusive contact is added, but this contact does not bear any force. It is found as a geometrical contact, but will eventually have to bear some forces, and thus, become a stability contact. Usually, for a stability contact, each vertex of the contact area is considered as the application point of a force that has to be in a friction cone. Since our method allows dealing with surfaces that are intersecting each other, the contact surface is not known beforehand. Therefore, as soon as a non inclusive contact is going to be used for the stability, we compute the intersection of the two polygons P_1 and P_2 that are involved, and that intersection $P_1 \cap P_2$ is the contact surface, and its vertices will bear the forces. We do not present here the algorithm to compute the intersection of two convex polygons, as it can be found in the literature easily.

6.5.6 Extension to singular cases

Our method can be extended to be used to approximate singular situations, such as finding an optimal contact with a linear or even punctual surface. This is done by giving a slight width to the point or the line. This approximation is physically grounded: in terms of real contacts, linear or punctual contacts do not exist. In fact, since all objects are deformable, even slightly, the contact area between two objects cannot be a perfect line, and must have a non-null area. Which justifies that linear and punctual contacts can be modeled as thin contact surfaces. By defining such a surface, we impose partly the orientation of the contact. Here again, one must be careful with numerical issues. Dealing with small numbers (here we would like to take width of a fraction of centimeter) may induce conditioning problems. Also, having two close parallel constraints of opposite direction (i.e. $g(x) \leq \alpha$ and $-g(x) \leq \alpha$ with α small) is not a good practice in optimization as it will lead the solver to take small steps. Therefore, it is best to apply a scaling to the constraints by applying a geometrical scaling to P_1 and P_2 in the appropriate direction.
Likewise, constraints on the area of the ellipse should be based on the same formulation as in equation (6.8).

6.6 Simulation results

We dedicated considerable efforts in proposing a general multi-contact motion planner to solve cases of non-gaited acyclic planning. Given a humanoid robot, an environment, a start and a final desired postures, our planner generates a sequence of contact stances allowing any part of the humanoid to make contact with any part of the environment to achieve motion towards the goal. Our planner is thoroughly described in [14]. Extensions of this multi-contact planner to multi-agent robots and objects gathering locomotion and manipulation are presented in [10], and preliminary validations with some DARPA challenge scenarios, such as climbing a ladder, ingress/egress a utility car or crossing through a relatively constrained pathway are presented in [11]. In [14] and [10], we describe works in multi-contact that are achieved by other colleagues in robotics. In order to illustrate our method, we present some examples starring the HRP-2 and ATLAS humanoid robots, that are typical posture generations encountered in multi-contact planning. For the implementation of our posture generator, we use the RobOptim optimization framework [27] relying on the IPOPT solver [38].

6.6.1 Inclined ladder climbing

In this first example, we generate a posture that is part of an inclined ladder climbing planning. We consider that the robot HRP-2 reached a posture in which its right foot is on the first step and its right hand is grasping the right guardrail, both of those contacts are bearing forces. Those contacts are fixed, and we search a posture that adds to it a geometrical contact between the left foot and the second step. We require the contact to include an ellipse with both radiuses bigger than 40% of the ladder step's width. The resulting posture and a close-up view of the contact areas are shown in Fig. 6.6. The latter shows clearly how an ellipse of sufficient size is found, included in the contact area between the left foot and the second step. It also shows that the contact forces on the right foot are located on the vertex of the intersection of the contact surfaces between the right foot and the first step, which was also generated with our method, in a prior posture generation. One can note that we use a contact area slightly smaller than the actual surface under the foot of the robot. We use indeed safety margins to account for modeling errors so that the obtained posture is achievable by the real robot.

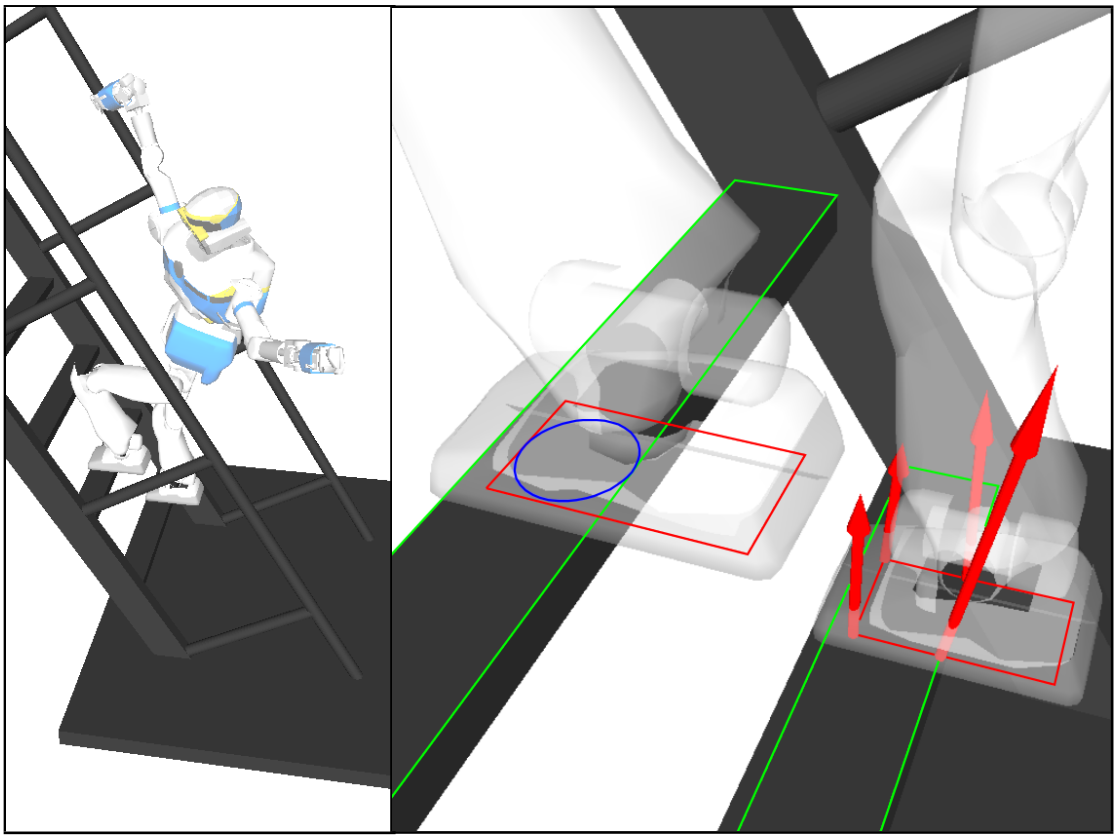

Fig. 6.6 HRP2-10 ladder climbing posture and up close view of the contact areas (green/red: contact polygons; blue: contact ellipse; red arrows: contact forces resultants)

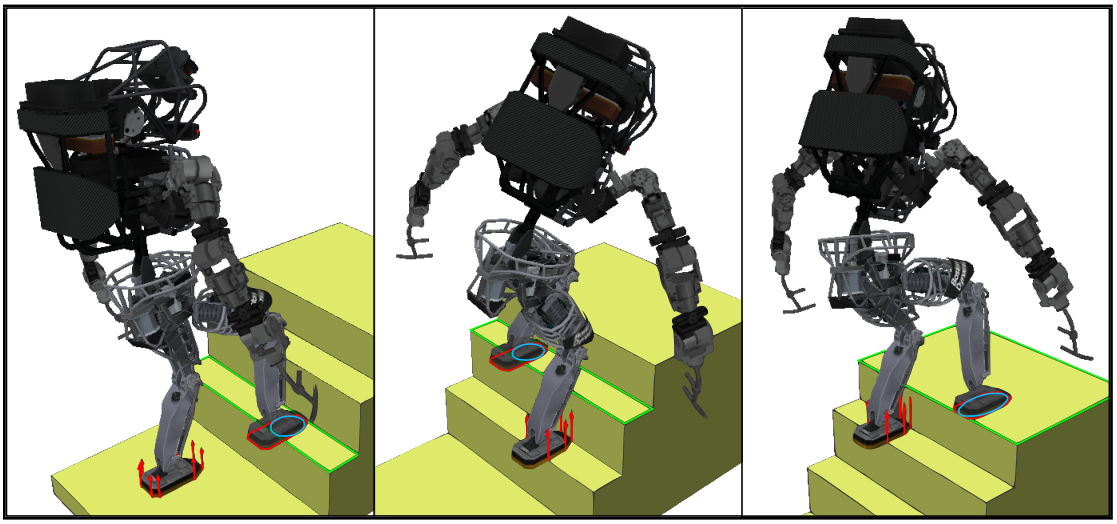

Fig. 6.7 Atlas climbing stairs with small steps by maximizing the size of the contact areas (green/red: contact polygons; blue: contact ellipse; red arrows: contact forces resultants)

6.6.2 Vertical ladder climbing

In the second example we generate a posture in which the robot climbs a vertical ladder. In this particular step, the robot is using its right foot and both hands to maintain its stability on top of the first rung of the ladder. We search a posture that keeps those previous contacts and adds a geometrical contact between the left foot and the second rung of the ladder. The result of that optimization can be observed on Fig. 6.1, with the robot posture on the left and a close-up look at the contact areas on the right. The difficulty of this situation is that a contact has to be made with a very thin surface of the environment (the ladder rung). Usual contact generation method would reduce a lot the span of possible contact position (by patching the robot's foot with a very small surface or by imposing a set of authorized contact positions). Whereas with our method, the contact configuration is found during the optimization process without requiring any extra human work. The contact chosen by our software includes an ellipse which first axis is the width of the robot's foot and second axis is as thin as the ladder rung. One problem to expect is that numerical instability might happen if the surface of the rung is given too thin. But that would also happen with full inclusion constraints. This example also illustrates one limitation of our method: it only considers planar contacts and if one wants to model a purely linear contact an other contact model must be used, since our modeling of those singular cases is approximative.

6.6.3 Climbing Stairs

In a third simulation, the ATLAS robot climbs a flight of stairs. All the steps are too small for the robot to put its entire foot on. Therefore, it has to make a non inclusive contact and we propose to maximize the size of the contact area with the ellipse included in it, as explained in 6.5.4. The size of the contact area is limited by the fact that the foot cannot penetrate the wall behind each step. On Fig. 6.7, we present 3 postures generated on this environment. On each of those postures, we see that the ellipse's size is maximized until the foot enters in collision with the vertical wall behind each step. And when possible, like on the last step, the contact area is maximized without collision limitation and the foot is positioned as fully included in the support surface. We can see here that even when the size of the ellipse is maximized while competing with other non-linear constraints like collision avoidance, our method still works well and leads us to a satisfactory solution.

6.7 Discussion and conclusion

Generating arbitrary shaped contact areas proved to be doable very simply in an optimization-based posture generation module. We focused on writing constraints that have continuous gradients, since the posture generate problem is dominantly smooth. Hence, our geometric contact model can be useful for other optimization-based purposes, for example control or trajectory optimization, and any gradient-based descent scheme which handles inequalities (e.g. [16]). While we were expecting an increase of computation time due to the addition of new variables in the problem, we noticed that the timings obtained with this method are sensibly the same that our previous version of the posture generator with full contact surface inclusion. Consequently, this method offering a richer contact search (exploration) during planning comes without degrading computation time. In fact, it truly allows us to substantially reduce the time spent by the user in ad-hoc tuning the shapes of the contact patches, or fixing the contact positions that were previously done by hand. Also, it is fairly easy to implement and extends a multi-contact planning algorithms like the one described in [14] to give it richer planning possibilities.

There are several opportunities for future work. For example, we could improve the generality of our contact constraint formulation even further to manage linear and punctual contacts without the approximations we currently use. Also we could extend our method to allow dealing with non-convex surfaces. And finally we would like to apply our method to other fields that use contact generation, like trajectory or control optimization. Our methods extends straightforwardly to point cloud data as far as polygonal patches can be extracted.

Chapter 7

Humanoid Posture Generation on non-Euclidean Manifolds

7.1 abstract

We present a reformulation of the posture generation problem that encompasses non-Euclidean manifolds. Such a formulation allows a more elegant mathematical description of the constraints, which we exemplify through some scenarios in the simulation results section. In our previous work, the posture generation problem is formulated as a non-linear optimization program with constraints expressed only through Euclidean manifolds; we solve the latter problem using on-the-shelf solvers. Instead, we decided to implement a new SQP solver that is most suited to non-Euclidean manifolds structural objects. By doing so, we have a better mastering in the way to tune and specialize our SQP solver for robotic problems.

7.2 Introduction

Computing robot configurations to meet the requirements of a given set of tasks, within a viable state, is a recurrent problem whose complexity grows with that of the robot. In this paper, we are interested in the following generalized inverse kinematics problem: we search a configuration for which the robot fulfills tasks under constraints of joint limits, auto-collision and non-desired collision avoidance, balance, torque limits, etc. We coined it posture generation. Such a problem is encountered in both planning and control. In both cases, computation time and robustness are critical issues.

We have already proposed various implementations of the humanoid posture generation problem. All of our implementations formulate the problem as a non-linear optimization

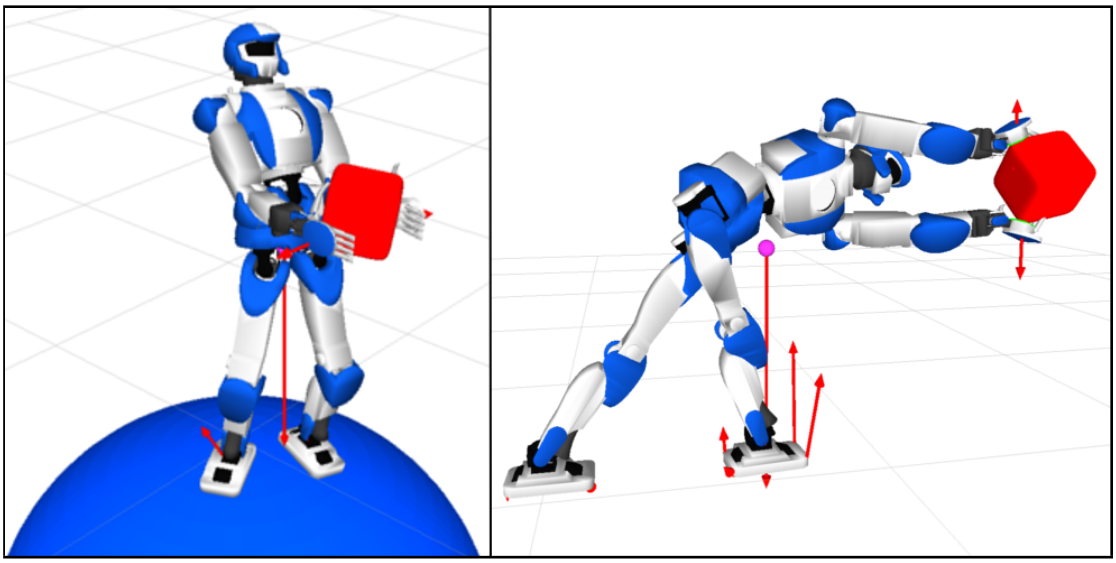

Fig. 7.1 HRP-4 carrying a 2-kg cube. Left: feet on a sphere, objective function is to maintain the cube at a given position. Right: right foot free to move on the floor, objective is to put the cube as far as possible in a given direction

program to address multi-contact planning. In [14], the multi-contact planner explores the contact space using thousands of HRP-2 humanoid posture generator (PG) queries; we used the FSQP solver [22]. In [10], the PG is extended to handle various humanoid robots and multiple agents, the solver used is IPOPT [38]. In [37] the PG is extended to various contact models and used to generate multiple related postures at once. The latter work and the DRC participation revealed that re-planning on the fly is necessary and having a robust PG is crucial in many situations. Other works also make use of PG, e.g. in [20][2].

Posture generation has been formulated as a problem over a Euclidean space. Robots variable may however be more naturally expressed over non-Euclidean manifolds. The archetypes for this are the rotation part of the root body for a humanoid robot, and ball joints, whose variables live in $SO(3)$. Some typical tasks are also naturally formulated on different manifolds. For example for making contact with any object that can be mapped on a sphere, the contact point position for this object can be parametrized in $S2$. Human shoulder can be elegantly parametrized on $S2 \times \mathbb{R}$, as proposed in [3].

Formulating the problem over $\mathbb{R}^n$ leads either to discontinuities that can prevent the convergence of the optimization solver, or to cumbersome writing to specify that the variable is actually living on a manifold (see [11]).

In this paper, we propose a new optimization solver able to work on generic smooth manifolds. We take inspiration from the approach used for unconstrained optimization on manifold [1] and adapt it to constrained optimization. To the best of our knowledge,

constrained optimization on manifold has drawn few research for now. This is likely due to the fact that in most problems the only constraint is to be on the manifold. We are only aware of the work of Schulman *et al.* [35], where the authors explain the adaptation of their solver to work on $SE(3)$. This adaptation is however not valid for general manifolds without more care about hessian computation.

The second contribution of this paper is a Posture Generation framework developed to ease the writing of functions, so that the user can focus on the problem formulation without having to care about the tedious bookkeeping inherent to optimization problems of this size.

A background motivation for this work is to have our own optimization solver, instead of a black box. We will now be able to specialize the solver specifically to robotic problems, by leveraging modeling properties and approximations, for a gain in time and robustness. We also look forward to using this solver for problems with a varying number of constraints along the iterations (such as when complex collision constraints are considered).

The rest of the paper is organized in a classical way: we start with a bit of math to describe the foundations; then we introduce the PG *per se*, the problem formulation followed with illustration of successful generations.

7.3 Optimization on Manifolds

In this section, we describe a Sequential Quadratic Programming (SQP) approach [29] to solve the following non-linear constrained optimization program

$$\begin{aligned} \min_{x \in \mathscr{M}} \quad & f(x) \\ \text{subject to } \; & l \leq c(x) \quad \leq u \end{aligned} \tag{7.1}$$

where $\mathscr{M}$ is a n-dimensional smooth manifold and c is a m-dimensional real-valued function.

7.3.1 Representation problem

When $\mathscr{M} = \mathbb{R}^n$, the problem (7.1) is solved iteratively, starting from an initial guess x_0 and performing successive steps $x_{i+1} = x_i + \mathbf{p_i}$ where $\mathbf{p_i}$ is the increment found at the i-th iteration, until convergence is achieved. The strategy to compute $\mathbf{p_i}$ depends on the solver.

This classical scheme cannot be readily applied to optimization over non-Euclidean manifolds. First of all, only (a subset of) the real numbers can be stored in computers. To

manipulate elements of $\mathscr{M}$ we need to choose a way to represent them in memory. This boils down to choosing a representation space $\mathbb{E} = \mathbb{R}^r$ (with $r \geq n$) and a map

$$\psi : \begin{array}{ccc} x & \mapsto & \mathbf{x} \\ \mathscr{M} & \to & \mathbb{E} \end{array}$$

In the following, we identify $\mathscr{M}$ with the set $\psi(\mathscr{M}) \subseteq \mathbb{E}$.

With this representation, it is tempting to simply transform problem (7.1) as an optimization over $\mathbb{R}^r$ with objective $f \circ \psi^{-1}$ and constraint $c \circ \psi^{-1}$, and solve it with a usual solver. But depending on the representation choice, one of the two following problems arises:
(i) $r = n$, then it is not possible in the general non-Euclidean case to find ψ without derivative discontinuities. This can lead to critical convergence problems,
(ii) $r > n$, then most elements of $\mathbb{E}$ do not represent an element of $\mathscr{M}$ and ψ cannot be surjective. Constraints need to be added to force the solution on $\mathscr{M}$. As a result, the problem has more variables and constraints w.r.t (i). Moreover, the additional constraints are unlikely to be met along the iteration process (even if x_i is an element of $\mathscr{M}$, $x_i + \mathbf{p_i}$ is likely not, as nothing enforces it). This means that in order to evaluate $f \circ \psi^{-1}$ and $c \circ \psi^{-1}$ at a given x_i, one has to project it on $\psi(\mathscr{M})$ first, effectively computing $f \circ \psi^{-1} \circ \pi$ and $c \circ \psi^{-1} \circ \pi$, where π is the projection. The composition by π is an additional burden in programming (see e.g. in [12]).

As a simple example, the set of 3D-rotations $SO(3)$ is a manifold of dimension 3. The following (classical) choices can be made

- Rotation matrix $\mathbf{R} \in \mathbb{R}^{3\times3} \approx \mathbb{R}^9$, additional constraints: $\{\mathbf{R}^t\mathbf{R} = I \ , \ \det(\mathbf{R}) = 1\}$, projection by orthogonalization,
- Quaternion $\mathbf{q} \in \mathbb{R}^4$, additional constraints: $\{\|\mathbf{q}\| = 1\}$, projection $\pi(\mathbf{x}) = \mathbf{x}/\|\mathbf{x}\|$,
- Euler angles ($\mathbb{E} = \mathbb{R}^3$), singularities when reaching gimbal lock.

7.3.2 Local parametrization

By definition, there is always, at a point x of a smooth n-dimensional manifold $\mathscr{M}$, a smooth map φ_x between an open set of $T_x\mathscr{M}$, the tangent space to $\mathscr{M}$ at x, and a neighborhood of x, with $\varphi_x(0) = x$. $T_x\mathscr{M}$ can be identified with $\mathbb{R}^n$. This gives us a local parametrization for $\mathscr{M}$. The driving idea of the optimization on manifolds is to change the parametrization at each

iteration. Applying this idea, we can reformulate Problem (7.1) around x_i as

$$\begin{aligned} \min_{\mathbf{z}\in T_{x_i}\mathscr{M}} \quad & f\circ\varphi_{x_i}(\mathbf{z}) \\ \text{subject to } \; l \leq c\circ\varphi_{x_i}(\mathbf{z}) \quad & \leq b \end{aligned} \tag{7.2}$$

This is an optimization problem on $\mathbb{R}^n$. If we perform one iteration of a classical solver starting from $\mathbf{z_0} = 0$, we get an iterate $\mathbf{z_1}$, which corresponds to the iterate $x_{i+1} = \varphi_{x_i}(\mathbf{z_1})$. We can then reformulate Problem (7.1) around x_{i+1}, perform a new iteration and repeat the process until convergence.

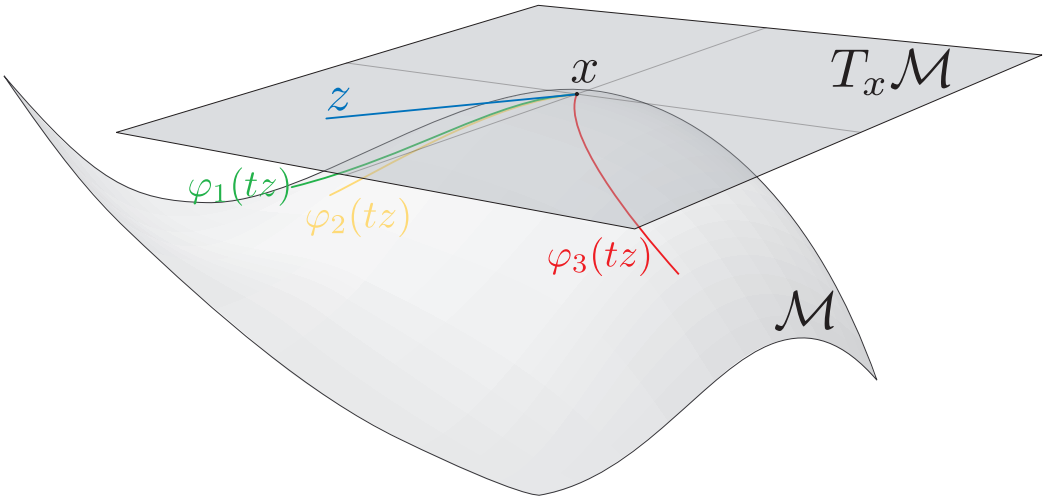

Fig. 7.2 There are many possible choices for φ_x but not all yield a curve $\varphi_x(t\mathbf{z})$ which is going in the same direction as $\mathbf{z}$: φ_1 and φ_2 are correct choices, φ_3 is not.

However, convergence cannot be achieved without care on the choice of φ_{x_i}: it must be such that for any $\mathbf{z}$, the curve $t \mapsto \varphi_{x_i}(tz)$ is tangent to $\mathbf{z}$, see Fig. 7.2, so that the update $x_{i+1} = \varphi_{x_i}(\mathbf{z_1})$ is made in the direction given by $\mathbf{z_1}$.

The exponential map is a good theoretical candidate, but it is often impractical or expensive to compute. Depending on the manifold, cheaper maps can be chosen.

With the iterative formulation approach described above, we do not have any parametrization issue, do not need additional constraints, and have the minimum number of optimization parameters. But we still need a map ψ and real space $\mathbb{E}$ to represent the x_i and keep track of them in a global way. The $\mathbf{x_i}$ are guaranteed to be on $\mathscr{M}$ so we can choose a representation with $r > n$ where ψ is singularity-free without any drawback. Also, the programmer can write the function $f' = f \circ \psi^{-1}$ as if it was a function from $\mathbb{E}$ to $\mathbb{R}$ without the need to project on $\psi(\mathscr{M})$ first (same goes for $c' = c \circ \psi^{-1}$). For example if $\mathscr{M} = SO(3)$ and $\mathbb{E} = \mathbb{R}^{3\times 3}$, $\mathbf{x_i}$ is automatically a rotation matrix and can be used directly as such when writing the function.

7.3.3 Local SQP on manifolds

We choose to adopt an SQP approach to solve our problem. We first define the Lagrangian function

$$\mathscr{L}_x(\mathbf{z},\lambda) = f \circ \varphi_x(\mathbf{z}) - \lambda^T c \circ \varphi_x(\mathbf{z}) \tag{7.3}$$

with $\lambda \in \mathbb{R}^m$ the vector of Lagrange multipliers, and note H_k the Hessian matrix $\nabla^2_{zz}\mathscr{L}_{x_k}$. Taking $\mathbf{z_0} = 0$, the first SQP step for Problem (7.2) is computed by solving the following quadratic program

$$\begin{aligned} \min_{\mathbf{z}\in\mathbb{R}^\mathbf{n}} \quad & \frac{\partial f \circ \varphi_{x_k}}{\partial \mathbf{z}}(0)^T \mathbf{z} + \frac{1}{2}\mathbf{z}^T H_k \mathbf{z} \\ \text{subject to } & 1 \leq c \circ \varphi_{x_k}(0) + \frac{\partial c \circ \varphi_{x_k}}{\partial \mathbf{z}}(0)\mathbf{z} \leq \mathrm{u} \end{aligned} \tag{7.4}$$

The basic SQP approach adapted to manifolds can be summarized as follows

1. set $k = 0$ and x_k to the initial value
2. compute $\mathbf{z}$ from Problem (7.4) for current x_k
3. set $x_k = \varphi_{x_k}(\mathbf{z})$
4. if convergence is not yet achieved go-to step 2

Computations of function values and derivatives are based on the fact that $f \circ \varphi = f' \circ \psi \circ \varphi$ (and same for c), and

$$\begin{aligned} f' &: \mathbb{E} \rightarrow \mathbb{R} \\ \psi \circ \varphi &: \mathbb{R}^n \rightarrow \mathbb{E} \end{aligned}$$

are representable functions. The gradient of $f \circ \varphi$ is

$$\frac{\partial f \circ \varphi_x}{\partial \mathbf{z}} = \frac{\partial f'}{\partial y}(\psi \circ \varphi_x) \times \frac{\partial (\psi \circ \varphi_x)}{\partial \mathbf{z}} \tag{7.5}$$

7.3.4 Practical implementation

The above SQP algorithm works locally, *i.e.* when starting close enough to the solution. In practice, various possible refinements are made to ensure convergence from any starting point. We detail hereafter our choices.

Maps φ_{x_i} are only valid locally, and we need to account for this: a step $\mathbf{z}$ found by Problem (7.4) should not be outside the validity region of the map. We could enforce this by adding a constraint $\mathbf{z}^-_{\text{map}} \leq \mathbf{z} \leq \mathbf{z}^+_{\text{map}}$ in (7.4). This leads naturally to trust region methods that we therefore favor over line-search approaches.

To know if a step $\mathbf{z}$ is acceptable or not, one usually uses a penalty-based merit function. In our early tests, the update of the penalty parameters proved to be difficult with our types of problems. We now use a filter instead.

Our algorithm is an adaptation of Fletcher's filter SQP [17] to the case of manifolds: we use an adaptive trust-region that is intersected with the validity region of φ_{x_i}, and a new iterate $x_{i+1} = \varphi_{x_i}(\mathbf{z})$ is accepted if either the cost function or the sum of constraint violations is made better than for any previous iterates.

Aside from the manifold adaptation, our main departure from Fletcher is in the Hessian computation where we used an approximation, since the exact one is too expensive to compute in our problems. After testing several possibilities, we settled for a self-scaling damped BFGS update [30, 29], adapted to the manifold framework. More precisely, given the Hessian approximation H_k at iteration k, we compute the approximation H_{k+1} as follows

$$
\begin{aligned}
&s_k = \mathscr{T}_z(z), \quad y_k = \nabla_z \mathscr{L}_{x_{k+1}}(0, \lambda_{k+1}) - \mathscr{T}_z(\mathscr{L}_{x_k}(0, \lambda_k)) \\
&\theta_k = \begin{cases} 1 & \text{if } s_k^T y_k \geq 0.2 s_k^T \tilde{H}_k s_k \\ \frac{0.8 s_k^T \tilde{H}_k s_k}{s_k^T \tilde{H}_k s_k - s_k^T y_k} & \text{otherwise} \end{cases} \\
&r_k = \theta_k y_k + (1 - \theta_k) \tilde{H}_k s_k \quad \text{(damped update)} \\
&\tau_k = \min\left(1, \frac{s_k^T r_k}{s_k^T \tilde{H}_k s_k}\right) \quad \text{(self-scaling)} \\
&H_{k+1} = \tau_k \left(\tilde{H}_k - \frac{\tilde{H}_k s_k s_k^T \tilde{H}_k}{s_k^T \tilde{H}_k s_k}\right) + \frac{r_k r_k^T}{s_k^T r_k}
\end{aligned}
$$

where $\mathscr{T}_{\mathbf{z}}$ is a vector transport along $\mathbf{z}$ (see [1]) and $\tilde{H}_k$ is such that for $\mathbf{u} \in T_{x_{k+1}}\mathscr{M}$, $\tilde{H}_k \mathbf{u} = \mathscr{T}_{\mathbf{z}}\left(H_k \mathscr{T}_{\mathbf{z}}^{-1}(\mathbf{u})\right)$.

Despite Powell's update, H_k might not be positive definite (but still symmetric). We regularize it as follows: we first perform a Bunch-Kaufman factorization $P_k H_k P_k^T = L_k B_k L_k^T$ where P_k is a permutation matrix, L_k is unit lower triangular and B_k is block diagonal with blocks of size 1×1 or 2×2 (obtaining B_k as a diagonal matrix is not numerically stable for Cholesky-like decomposition of indefinite matrices), see [19]. The eigenvalue decomposition $B_k = Q_k D_k Q_k^T$ is immediate and cheap to compute. From the diagonal matrix D_k we form D'_k such that $d'_{ii} = \max(d_{ii}, \mu_{\min})$ where $\mu_{\min} > 0$ is user-defined (we typically set it to 0.1). Defining $L'_k = L_k Q_k (D'_k)^{1/2}$, we get a regularized matrix $H'_k = P_k^T L_k L_k^T P_k$. In our case, we

use `LSSOL` [18] for solving the QP (7.4), which directly accepts the factorized form (P_k, L'_k). This avoids an internal Cholesky factorization so that our regularization does not add too much time to the overall process of building and solving the QP.

The code for $\psi \circ \varphi$, its gradient and the vector transport needs only to be implemented once for each elementary manifold (it is then trivial to get those functions for Cartesian products of manifolds). The composition with f' and c' is done automatically. The expression of those functions is adapted from [7].

7.4 Posture Generation, variables and architecture

Writing a posture generation problem can easily become cumbersome without the appropriate tools. Common pitfalls are for example writing the derivative of a function, managing how the Jacobian matrices of the already implemented functions are modified when a variable is added to the problem, adding a new type of constraint, or correctly writing a function on a sub-manifold of the problem manifold. A fair amount of bookkeeping is always necessary, which should not be the charge of the user writing the constraints. In our PG, we propose an architecture automating most of the problematic tasks, so that the user can focus on the mathematical formulation of the problem.

7.4.1 Geometric expressions

Most constraints are geometric. In order to simplify the writing of functions, we use a dedicated system of expression graph encapsulated in a set of geometric objects. The main idea is to separate the purely mathematical logic from the geometric one. As an example if P_r and V_r are a point and a vector attached to the camera of the robot, and P_e is a fixed point in the environment, the constraint $(P_e - P_r).V_r = 0$ can be used to have the robot look at P_e. With our system, the user creates only those objects and write the code `(Pe-Pr).dot(Vr)` to get the value needed. The geometric layer takes care that all the quantities are expressed in the correct frame, the mathematical layer performs the corresponding operations. If q is a variable object, `(Pe-Pr).dot(Vr).diff(q)` returns automatically the differential of the expression w.r.t. q. This makes the writing of the constraints very easy.

At the mathematical level, we consider 5 types of expressions which can be either variables or constants:

- Scalar, a 1-dimensional element of $\mathbb{R}$
- Coordinates, a 3-dimensional element of $\mathbb{R}^3$

- Rotation, a 3×3 matrix representing a 3D rotation
- Transformation, a 4×4 matrix representation of a 3D isometry
- Array, a dynamic size array

The meaningful unary (inverse, opposite, norm...) and binary (multiplication, addition, subtraction, dot product...) operations (with their derivatives by chain rule) are implemented. We also have a Function class for more complicated expressions, for example expressing $q \mapsto T_i(q)$ where T_i is the transformation between the reference frame of the robot and the frame of its i-th body [1]. The combinations of those elementary operations defines a computation graph.

The geometric layer consists of physical or geometric objects, named features, which exist independently of their mathematical expression in a given reference frame. We have so far 4 objects:

- A Frame, defined by a Transformation expression and a reference frame.
- A Point and a Vector, defined by a Coordinates expression and a reference frame.
- A Wrench, defined by a pair of Coordinates expressions and a reference frame.

We have a special World Frame object to serve as starting reference frame.

For each feature, one can get its expression in a given frame. Basic operations are defined between those features (when applicable). For example, the subtraction between two Points gives a Vector. The geometric logic resides in the change of frame and those operations.

7.4.2 Automatic mapping

The manifold $\mathscr{M}$, on which the optimization takes place, is a Cartesian product of several sub-manifolds. Same goes for their representation spaces:

$$\begin{aligned} \mathscr{M} &= \mathscr{M}_1 \times \mathscr{M}_2 \times \mathscr{M}_3 \times \dots \\ \mathbb{E} &= \mathbb{E}_1 \times \mathbb{E}_2 \times \mathbb{E}_3 \times \dots \end{aligned} \tag{7.6}$$

From the solver's viewpoint, the entry space of each function is the complete manifold. But for the developer, writing a function on the complete $\mathbb{E}$ is cumbersome because (i) of the need to manage indexes, and (ii) when the function is implemented, the complete

[1] The kinematics of rigid body systems is handled by the RBDyn library (https://github.com/jorisv/RBDyn)

$\mathbb{E}$ may not be known. A user-written function f is usually defined on a subset of $\mathbb{E}$, say $\mathbb{E}_I = \mathbb{E}_i \times \mathbb{E}_j \times \mathbb{E}_k \ldots$, that is minimalist for that function, and should not account for unrelated manifolds. One does not want to think about the values of the forces when writing a geometric constraint for example. Our automatic mapping tool generates the correct projection functions π_I such that the developer can write a function f on $\mathbb{E}_I$ while the solver receives it as a function $f \circ \pi_I$ on $\mathbb{E}$. This idea is illustrated by the example in Fig. 7.3

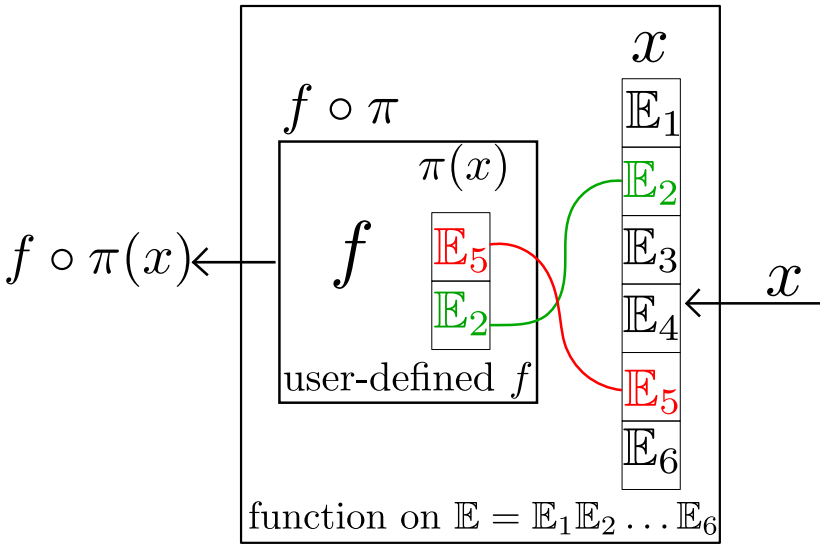

Fig. 7.3 automatic variable mapping

7.4.3 Problem Generator

The problem generator is the tool constructing the optimization problem. It registers all the variables and the functions related to a given problem. Each function is likely to bring additional variables with it. For each contact contributing to the balance, a variable on $\mathbb{R}^3$ representing the contact force is added to the problem. The associated wrench is added to the stability constraints. Once the registration is complete, the complete manifold of the problem is generated and uses the information of the Automatic mapping to “plug" each function with the correct sub-manifold. Subsequently, the optimization problem can be generated and passed to the solver. The communication between the solver and the generated problem is made through the RobOptim framework[2].

[2]http://www.roboptim.net/

7.5 Problem formulations

Let $q = [q_F^T; q_r^T] \in \mathbb{R}^3 \times SO(3) \times \mathscr{M}_r$ be the combination of the free-flyer of the robot $q_F \in \mathbb{R}^3 \times SO(3)$ and the articular parameters $q_r \in \mathscr{M}_r$. Let $\mathscr{W}_i(p) = \{f_i, m_i(p)\}$ be the wrench (force+moment) applied by the environment onto the robot at contact i and expressed on point p. A frame F is composed of a reference point and an orthonormal basis of 3 vectors $F = \{O, (x,y,z)\}$.

Here is a list of constraints that we consider in our problem (implementation of other ones is on-going):

- Joint limits $q^- \leq q_r \leq q^+$:
  These cannot be directly translated on manifolds other than $\mathbb{R}^n$. For example, spherical joints can be parametrized on $S2 \times \mathbb{R}$, then the $S2$ part can be limited by a cone, and the $\mathbb{R}$ part can have real bounds.

- The contact constraint consists in identifying the features of two frames F_1 and F_2. For example, for a planar contact, we get the set of equation 7.7.

$$\begin{aligned} (O_2 - O_1).z_1 &= 0 \\ z_2.z_1 &\leq 0 \\ z_2.x_1 &= 0 \\ z_2.y_1 &= 0 \end{aligned} \tag{7.7}$$

  Note that on F_2 only the point O_2 and the vector z_2 are necessary. Other types of contacts can be created that way, by equalizing other features, as explained in [14].

- The stability constraint ensures that the Euler-Newton equation (7.8) is balanced for the set of external wrenches applied to the robot (gravity $\mathscr{W}_G$ and contact forces $\mathscr{W}_i$).

$$\sum_i \mathscr{W}_i(p) + \mathscr{W}_G(p) = 0 \tag{7.8}$$

  For each contact that bears forces ("stability" contact), a wrench applied on the robot at the contact point is added to the problem. That wrench is parametrized on a subset of $\mathbb{R}^6$ depending on the type of contact. For punctual contacts, the moment part is null on the application point. Only a parametrization of the force part on $\mathbb{R}^3$ is needed. We model planar contacts as a combination of punctual forces applied at each vertex of the contact polygon. In the case of interaction forces between 2 robots, only one wrench is

created and it is used as is in the stability equation of one robot and its opposite is used for the stability of the second robot.

- The friction cone constraint limits the tangential part of every forces to avoid slippage. We write it as 7.9 (with μ the friction coefficient)

$$\begin{aligned} \mu^2 f_z^2 - f_x^2 - f_y^2 &\geq 0 \\ f_z &\geq 0 \end{aligned} \tag{7.9}$$

The frame in which the constraints are written matters critically. Most often, the frame's configuration depends on a part of the optimization variables, that must be accounted for in computing the constraints' Jacobian. Our framework computes such dependencies automatically.

Our current PG (i.e. coding state) does not include yet collisions and auto-collisions, nor torque limits. Their implementation is on-going and is simply the matter of coding time. Another important part is its cost function. We only mention the cost function that have specificities when dealing with manifolds, the distance to a reference posture q_0. On a robot that has all its articulations parametrized on $\mathbb{R}$ the distance can be expressed simply with the Euclidean norm $d = \left\| q_j - q_0 \right\|^2$. Since we work on non-Euclidean manifolds, the logarithm function on the manifold must be used. It gives the distance vector between two points in the tangent space, the norm of this vector can be used as a distance. So we get $d = \left\| \log_{q_0}(q_r) \right\|^2$.

7.6 Simulation Results

Here, we present several posture generation problems resolution that leverage the specific capabilities of our software.

7.6.1 Application to plan-sphere contact

When we consider a planar contact, having a frame F_S fixed in reference to F_B is sufficient because the equations describing that contact are invariant w.r.t. the point's location. But for different contact topologies, the location of the contact point in the body's frame F_B matters. We propose to parametrize the location and normal of the contact point with an additional variable.

We consider the contact between a body's flat surface S_B of normal n_B, with the surface of a sphere S_s of center c_s, radius r_s, and let p_s and n_s be a point and its normal to S_S. The most general way to express such a constraint is to ensure that p_s is on S_B and that n_s and

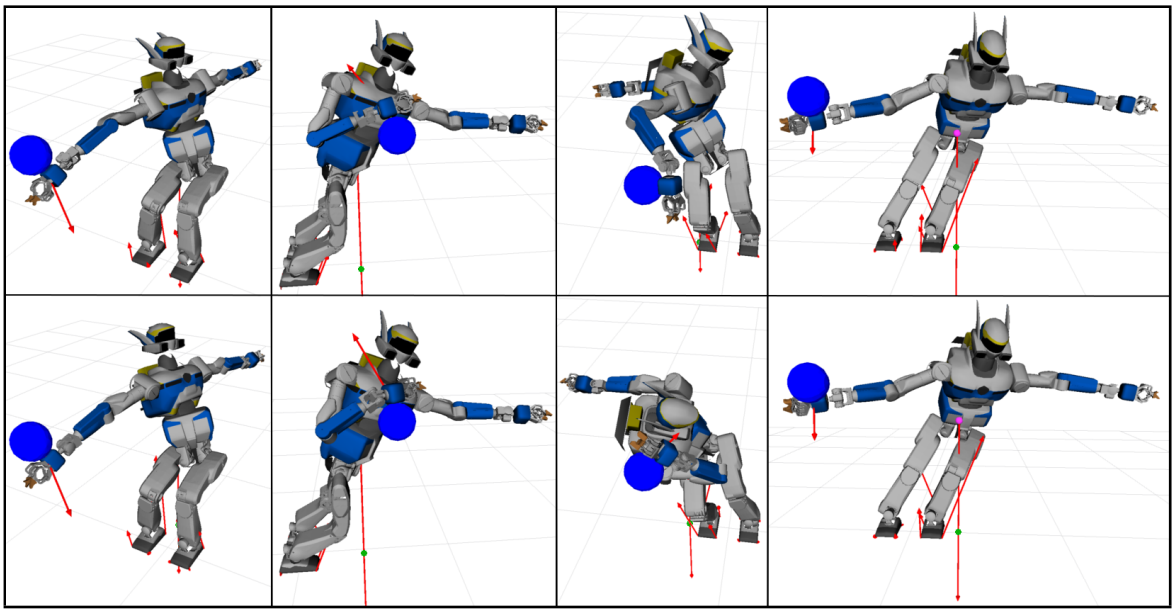

Fig. 7.4 HRP2-Kai leaning on sphere with right wrist to point the left gripper as far as possible in 4 cardinal directions. Top row: semi-predefined contact; Bottom row: free contact with parametrized wrist. Projection of the CoM on the ground (green dots)

n_B are opposite. This means creating a variable v_{S2} on the manifold $S2$ and map p_S and n_S on it. In our framework, this constraint is expressed exactly as the contact between 2 planar surfaces, once the mappings of $p_s(v_{S2})$ and $n_s(v_{S2})$ are done. In a framework that does not handle manifolds (as we do), it would require to setup a specific constraint, ensuring that the distance between c_s and S_B is equal to r_S.

In Fig. 7.4 we show the results obtained by solving a problem where the HRP2-Kai robot has to keep its feet in contact with the ground at fixed positions, touch a sphere with a side of its right wrist and point as far as possible in a given direction d with its left hand, under balance constraints. The top row of Fig. 7.4 shows the results for this problem with several different d. In every situation, the projection of the CoM is outside the polygon of support, meaning that such postures would not be reached without leaning on the sphere.

7.6.2 Contact with parametrized wrist

Being able to choose the location of the contact point on the sphere is interesting, but a limitation of this formulation is that the contact point on the wrist of the robot is restricted to one single user-defined face. Instead, we describe the shape of the wrist body as a parametric function and let the contact point on the wrist as well as its counterpart on the sphere, result from the optimization process. The section of HRP2-Kai's wrist is a square with rounded

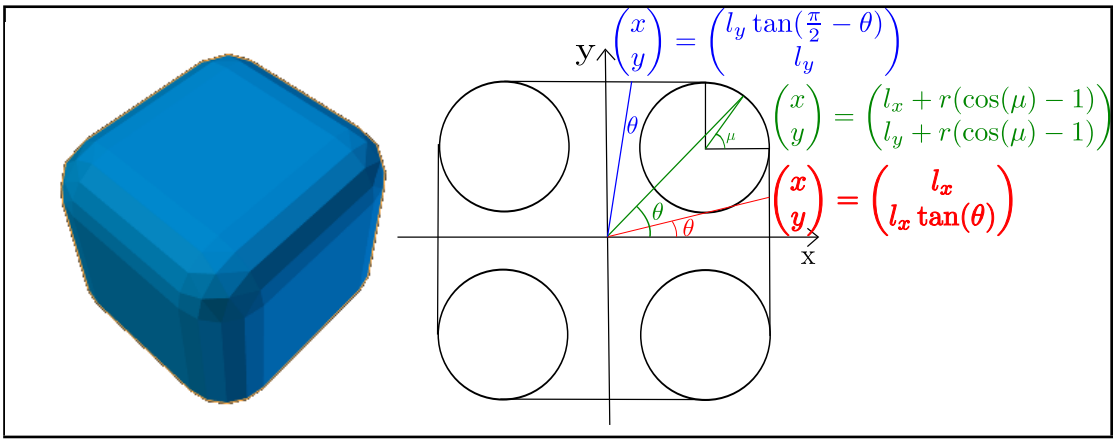

Fig. 7.5 Parametrization of the wrist of HRP2-Kai

edges. We parametrize this shape as shown in Fig. 7.5: we consider the angular coordinate θ of the point on the section. It is added as a variable to the problem. The shape of a quarter of section $[0;\pi/2]$ is a succession of a vertical line, a quarter of circle and a horizontal line. This pattern is repeated for the 3 other quarters. The equations are given in Fig. 7.5. In our framework, we define the function describing the shape of the wrist, create a frame parametrized by that function and then define the contact between that frame and the point and normal on the sphere. This formulation not only is very easy to implement, but most importantly, allows for richer posture generations. The optimization algorithm chooses the contact point on the sphere as well as the contact point on the wrist, which leads to a wider accessibility range, and a better satisfaction of the cost function. The bottom row of Fig. 7.4 displays the results of this simulation for the robot pointing in 4 directions. Notice that on the 2nd and the 4th (pointing forward and to the left) images, the results for the 2 types of models are nearly identical. Whereas in the 1st and 3rd images, different faces of the wrist have been chosen(On the 1st, the wrist is rotated by 180°, and 90°) on the 3rd). In these 4 cases, the contact with parametrized wrist gives a better cost of the objective function. This observation scales: we solved this problem for 5000 random pointing directions, and in average, the contact with parametrized wrist allows to reach 5mm further. The success rate of the solver is 98.5% in the parametrized wrist case against 99.9% when the face is fixed. The numbers of iterations are similar.

This method is certainly scalable, and can be used for any kind of humanoid robot and environment. Yet, it requires to have a parametric equation of the surface. We plan to implement a method to generate a parametrized surface point and its normal directly from the 3D mesh of an object. The accompanying video shows the optimization process for the

problem with parametrized wrist. Notice on the video that the contact point on the wrist changes sides all along the iterations.

7.6.3 Contact with an object parametrized on $S2$

In this simulation case, we want the HRP-4 robot (anther model) to carry a cube with its two hands. The most general way to do it is to select a face of the cube for each contact, and enforce the contact between that face and the hand's surface. We propose to approximate the cube with a superellipsoid and to parametrize the resulting shape on $S2$. The implicit equation of a superellipsoid is $S(x,y,z) = 0$, with

$$S(x,y,z) = \left(\left|\frac{x}{A}\right|^r + \left|\frac{y}{B}\right|^r\right)^{\frac{t}{r}} + \left|\frac{z}{C}\right|^t - 1 \tag{7.10}$$

A point in $S2$ is represented by a vector $v = (x,y,z)$ in $\mathbb{E} = \mathbb{R}^3$. To a given unit vector v we associate a point αv on the surface of the superellipsoid by solving $S(\alpha v) = 0$ for α. At this point, the normal is given by $\frac{\nabla S(\alpha v)}{\|\nabla S(\alpha v)\|}$ which simplifies into $\frac{\nabla S(v)}{\|\nabla S(v)\|}$. Given this parametrization, we write a contact constraint between the frame of the hand of the robot and the point and normal on the surface of the superellipsoid.

In Fig. 7.1 we present some results for a posture generation problem with manipulation: On the left side, the feet are free to move on a sphere, and, on the right side, the left foot position is fixed and the right foot is free to move on the ground. The hands must be in contact with the cube. The cube is free to move (parametrized by $\mathbb{R}^3 \times SO(3)$) and has it own set of Euler-Newton equations, which must be fulfilled. On the accompanying video, one can observe how the contact points on the cube evolve along with the optimization.

7.6.4 Posture Generation with a human model

The geometric model of a human is much more complex than the one of a humanoid robot in terms of topology. Even with the simplest models, the shoulders, wrists, ankles, or hips need to be described as spherical joints, and therefore be parameterized on $SO(3)$. We showcase that our solver is able to handle such complex models as a human avatar in Fig. 7.6. Our human model has spherical joints on its wrists, shoulders, torso, hips and ankles. With the addition to the free-flyer, the manifold that contains the articular space of that human model is: $\mathscr{M}_H = SO(3) \times \mathbb{R}^3 \times (SO(3))^9 \times \mathbb{R}^4$. We fixed the neck's joints as well as the fingers to avoid unnecessary variables. In this simulation, we require the human to stand on an inclined slope while leaning on the left side's wall. Since we have not yet implemented the boundary

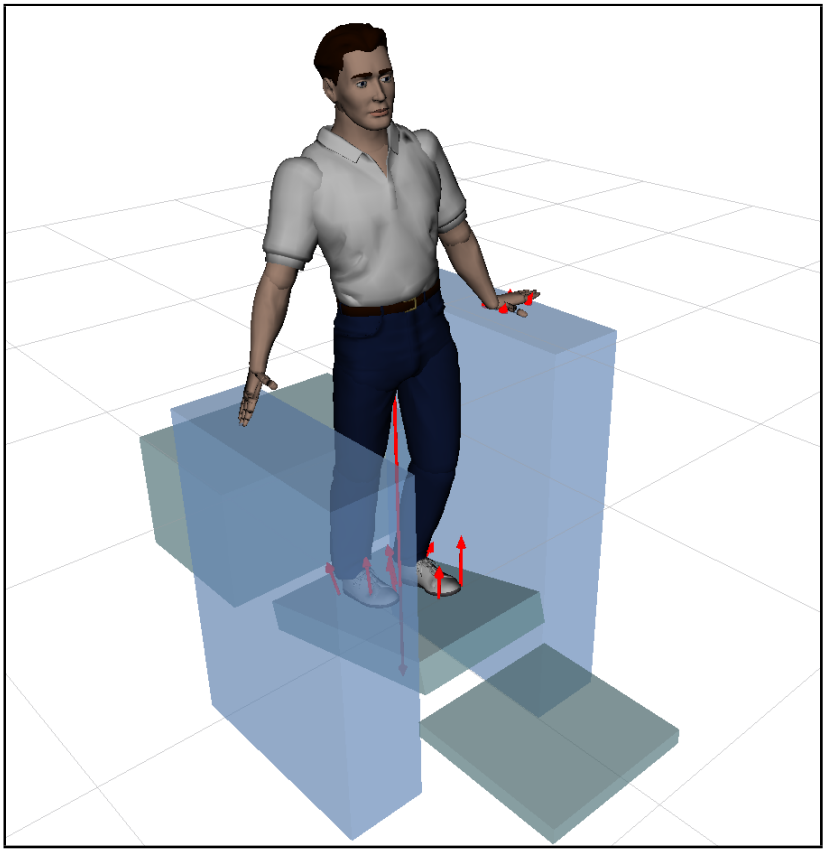

Fig. 7.6 Posture generation for a human avatar

limits on spherical joints, the model could reach non desired configurations. That issue limits for the time being the number of scenarios that we can solve. But the use of a cost function on the posture of type $d = \left\|\log_{q_0}(q_j)\right\|^2$, attracting the avatar to a basic standing posture, allowed us to have acceptable results.

7.7 Discussion and conclusion

Writing the posture generation problem as a non-linear optimization problem on non-Euclidean manifolds proves to be an elegant approach in terms of code structuring and user interface, in addition to mathematical readability. This work is still on-going and only preliminary results of the current state of the implementation are shown. We illustrate some posture generation problems with the HRP2-Kai and the HRP-4 humanoid robots; the results are very promising indeed. Our future (on-going) work is focused on the following streams:

- complement other functionalities as ready-to-use templates (e.g. constraints on task forces, collision avoidance, etc.);
- specialize the solver to humanoid PG problems and benchmark various numerical approaches (e.g. for the choice of the Hessian approximation, the trust region, tuning some parameters, etc.) and exploiting, if any, robotic model properties;
- improve convergence, numerical robustness and computation time of the PG. Currently, the resolution of any of the presented problems takes a few seconds on a laptop with Intel Core i7-3840QM CPU @ 2.80GHz. We aim at reducing it to a tenth of a second.

Once the code is more stable and finalized, we plan to make it open-source.

References

[1] Absil, P.-A., Mahony, R., and Sepulchre, R. (2008). *Optimization Algorithms on Matrix Manifolds*. Princeton University Press.

[2] Aristidou, A. and Lasenby, J. (2009). Inverse Kinematics: a review of existing techniques and introduction of a new fast iterative solver. Technical report, University of Cambridge.

[3] Baerlocher, P. and Boulic, R. (2001). Parametrization and range of motion of the ball-and-socket joint. In *Deform. Avatars*, pages 180–190.

[4] Barber, C. B., Dobkin, D. P., and Huhdanpaa, H. (1996). The Quickhull algorithm for convex hulls. In *ACM Transactions on Mathematical Software, Vol. 22*, pages 469–483, University of Minnesota.

[5] Benallegue, M., Escande, A., Miossec, S., and Kheddar, A. (2009). Fast $\mathscr{C}^1$ proximity queries using support mapping of sphere-torus-patches bounding volumes. In *IEEE International Conference on Robotics and Automation*.

[6] Biswas, J. and Veloso, M. M. (2012). Depth camera based indoor mobile robot localization and navigation. In *IEEE International Conference on Robotics and Automation*, pages 1697–1702.

[7] Boumal, N., Mishra, B., Absil, P.-A., and Sepulchre, R. (2014). Manopt, a matlab toolbox for optimization on manifolds. *Journal of Machine Learning Research*, 15:1455–1459.

[8] Bouyarmane, K., Escande, A., Lamiraux, F., and Kheddar, A. (2009). Collision-free contacts guide planning prior to non-gaited motion planning for humanoid robots. In *IEEE International Conference on Robotics and Automation*.

[9] Bouyarmane, K. and Kheddar, A. (2011). Using a multi-objective controller to synthesize simulated humanoid robot motion with changing contact configurations. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Fransico, CA.

[10] Bouyarmane, K. and Kheddar, A. (2012a). Humanoid robot locomotion and manipulation step planning. *Advanced Robotics*, 26.

[11] Bouyarmane, K. and Kheddar, A. (2012b). On the dynamics modeling of free-floating-base articulated mechanisms and applications to humanoid whole-body dynamics and control. In *IEEE/RSJ International Conference on Humanoid Robots*.

[12] Bouyarmane, K. and Kheddar, A. (2012c). On the dynamics modeling of free-floating-base articulated mechanisms and applications to humanoid whole-body dynamics and control. In *IEEE-RAS Int. Conf. Humanoid Robot.*, pages 36–42, Osaka, Japan.

[13] Chestnutt, J., Takaoka, Y., Suga, K., Nishiwaki, K., Kuffner, J., and Kagami, S. (2009). Biped navigation in rough environments using on-board sensing. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, (IROS'09)*, pages 3543–3548, Piscataway, NJ, USA. IEEE Press.

[14] Escande, A., Kheddar, A., and Miossec, S. (2013). Planning contact points for humanoid robots. *Robotics and Autonomous Systems*, 61(5):428 – 442.

[15] Escande, A., Kheddar, A., Miossec, S., and Garsault, S. (2008). Planning support contact-points for acyclic motions and experiments on HRP-2. In *International Symposium on Experimental Robotics*, Athens, Greece.

[16] Escande, A., Mansard, N., and Wieber, P.-B. (2010). Fast resolution of hierarchized inverse kinematics with inequality constraints. In *IEEE International Conference on Robotics and Automation*, pages 3733 – 3738, Anchorage, USA.

[17] Fletcher, R. and Leyffer, S. (2000). Nonlinear programming without a penalty function. *Mathematical Programming*, 91:239–269.

[18] Gill, P. E. E., Hammarling, S. J., Murray, W., Saunders, M. A., and Wright, M. H. (1986). User's guide for lssol (version 1.0): a fortran package for constrained linear least-squares and convex quadratic programming. Technical Report 86-1, Standford University, Standord, California 94305.

[19] Golub, G. and Van Loan, C. (1996). *Matrix computations*. John Hopkins University Press, 3rd edition.

[20] Hauser, K., Bretl, T., and Latombe, J.-C. (2005). Non-gaited humanoid locomotion planning. In *IEEE/RSJ International Conference on Humanoid Robots*, pages 7–12.

[21] Hauser, K., Bretl, T., Latombe, J.-C., Harada, K., and Wilcox, B. (2008). Motion planning for legged robots on varied terrain. In *Intl. J. of Robotics Research 27(11-12):1325-1349*.

[22] Lawrence, C., Zhou, J. L., and Tits, A. L. (1997). User's guide for CFSQP version 2.5: A C code for solving (large scale) constrained nonlinear (minimax) optimization problems, generating iterates satisfying all inequality constraints.

[23] Lengagne, S., Vaillant, J., Yoshida, E., and Kheddar, A. (2013). Generation of whole-body optimal dynamic multi-contact motions. *I. J. Robotic Res.*, 32(9-10):1104–1119.

[24] Lutz, C., Atmanspacher, F., Hornung, A., and Bennewitz, M. (2012). Nao walking down a ramp autonomously. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5169–5170.

[25] Maier, D., Hornung, A., and Bennewitz, M. (2012). Real-time navigation in 3D environments based on depth camera data. In *Humanoids'12: 12th IEEE-RAS International Conference on Humanoid Robots*, Osaka, Japan.

[26] Mordatch, I., Todorov, E., and Popović, Z. (2012). Discovery of complex behaviors through contact-invariant optimization. *ACM Trans. Graph.*, 31(4):1–8.

[27] Moulard, T., Lamiraux, F., Bouyarmane, K., and Yoshida, E. (2013). Roboptim: an optimization framework for robotics. In *Japan Society for Mechanical Engineers: Robotics and Mechatronics Conference*.

[28] Nakhaei, A. and Lamiraux, F. (2008). Motion planning for humanoid robots in environments modeled by vision. In *Humanoids'08: 8th IEEE-RAS International Conference on Humanoid Robots*, pages 197–204.

[29] Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer, 2nd edition.

[30] Nocedal, J. and Yuan, Y.-x. (1993). Analysis of a self-scaling quasi-newton method. *Mathematical Programming*, 61(1-3):19–37.

[31] Osswald, S., Gorog, A., Hornung, A., and Bennewitz, M. (2011). Autonomous climbing of spiral staircases with humanoids. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 4844–4849.

[32] Oßwald, S., Hornung, A., and Bennewitz, M. (2012). Improved proposals for highly accurate localization using range and vision data. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1809–1814, Vilamoura, Portugal.

[33] Poppinga, J., Vaskevicius, N., Birk, A., and Pathak, K. (2008). Fast plane detection and polygonalization in noisy 3d range images. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3378–3383.

[34] Rusu, R. B. and Cousins, S. (2011). 3d is here: Point cloud library (pcl). In *IEEE International Conference on Robotics and Automation*, Shanghai, China.

[35] Schulman, J., Duan, Y., Ho, J., Lee, a., Awwal, I., Bradlow, H., Pan, J., Patil, S., Goldberg, K., and Abbeel, P. (2014). Motion planning with sequential convex optimization and convex collision checking. *Int. J. Rob. Res.*

[36] Sentis, L. and Khatib, O. (2010). Compliant control of multicontact and center-of-mass behaviors in humanoid robots. *IEEE Trans. Robot.*, 26(3):483–501.

[37] Vaillant, J., Kheddar, A., Audren, H., Keith, F., Brossette, S., Kaneko, K., Morisawa, M., Yoshida, E., and Kanehiro, F. (2014). Vertical Ladder Climbing by HRP-2 Humanoid Robot. In *IEEE-RAS Int. Conf. Humanoid Robot.*, pages 671–676, Madrid, Spain.

[38] Wächter, A. and Biegleri, L. (2006). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57.

[39] Whitty, M., Cossell, S., Dang, K., Guivant, J., and Katupitiya, J. (2010). Autonomous navigation using a real-time 3d point cloud. In *ACRA'10: Australasian Conference on Robotics & Automation*, Brisbane.

[40] Zhang, Y., Luo, J., Hauser, K., Ellenberg, R., Oh, P., Park, H., Paldhe, M., and Lee, C. (2013). Motion planning of ladder climbing for humanoid robots. In *IEEE Conf. on Technologies for Practical Robot Applications*.