Multi-Contact Postures Computation on Manifolds

Stanislas Brossette, Adrien Escande, Abderrahmane Kheddar

Multi-Contact Postures Computation on Manifolds

Stanislas Brossette, Adrien Escande, and Abderrahmane Kheddar, *Senior Member, IEEE*

***Abstract*—We propose a framework to generate static robot configurations satisfying a set of physical and geometrical constraints. This is done by formulating nonlinear constrained optimization problems over non-Euclidean manifolds and solving them. To do so, we present a new sequential quadratic programming (SQP) solver working natively on general manifolds, and propose an interface to easily formulate the problems, with the tedious and error-prone work automated for the user. We also introduce several new types of constraints for having more complex contacts or working on forces/torques. Our approach allows an elegant mathematical description of the constraints and we exemplify it through formulation and computation examples in complex scenarios with humanoid robots.**

***Index Terms*—Robotics, Posture Computation, Contacts, Optimization on Manifolds.**

I. Introduction

COMPUTING static robot configuration that fulfills a set of objectives is needed in many aspects of robotics. This is crucial in humanoid robotics, where it has been used to find feasible multi-contact postures in planning e.g. [1], [2], [3], to provide low-priority reference postures in control [4], to explore feasible head position for object reconstruction [5] or to discover the surroundings [6].

For a given robotic system, we consider the problem of finding a single configuration to achieve a set of objectives T_i. We denote $\mathbf{q}$ the joint configuration (n joints plus a reference base) of the robot, $\xi = \left(\xi_1^T, \dots, \xi_m^T\right)^T$ represents the m contact forces applied on the robot, and $\mathbf{y}$ denotes additional variables. Each objective T_i can be represented by a set of equality and inequality equations:

$$\begin{cases} g_i(\mathbf{q}, \xi, \mathbf{y}) = 0 \\ h_i(\mathbf{q}, \xi, \mathbf{y}) \geq 0 \end{cases} \tag{1}$$

The extra variable $\mathbf{y}$ is useful to express many constraints [7] as explained later (section V). In addition to satisfying the tasks T_i, the configuration we seek for –we call it posture– must represent a feasible state of the robotic system. This translates into the following feasibility problem:

$$\mathcal{Q}: \begin{cases} q_i^- \leq q_i \leq q_i^+ \quad \forall i \in [1, n] & (2) \\ \tau_i^- \leq \tau_i(\mathbf{q}, \xi) \leq \tau_i^+ \quad \forall i \in [1, n] & (3) \\ \epsilon_{ij} \leq d(r_i(\mathbf{q}), r_j(\mathbf{q})), \quad \forall (i,j) \in \mathcal{I}_{\text{auto}} & (4) \\ \epsilon_{ik} \leq d(r_i(\mathbf{q}), O_k), \quad \forall (i,k) \in \mathcal{I}_{\text{coll}} & (5) \\ s(\mathbf{q}, \xi) = 0 & (6) \\ c(\xi_i) \geq 0 \quad i \in [1, m] & (7) \\ g_i(\mathbf{q}, \xi, \mathbf{y}) = 0 & (8) \\ h_i(\mathbf{q}, \xi, \mathbf{y}) \geq 0 & (9) \end{cases}$$

where equations (2) and (3) are the joint and torque limits respectively; eq. (4) disallows auto-collision between a pair of robot bodies $\{r_i, r_j\}$ given by a set $\mathcal{I}_{\text{auto}}$ and where d is the distance function; eq. (5) forbids collisions between a robot body r_i and an object of the environment O_k defined in a set $\mathcal{I}_{\text{coll}}$; ϵ_{ij} and ϵ_{ik} are the acceptable distance thresholds. Eq. (6) ensures the stability of the robot, by requiring the balance of external forces and torques applied to the robot. Finally, the contact forces must lie within the Coulomb friction cones, which is enforced by eq. (7). We illustrate these constraints in Fig. 1.

The set $\mathcal{Q}$, if not empty, usually contains infinitely many feasible postures and we may want to pick-up one according to

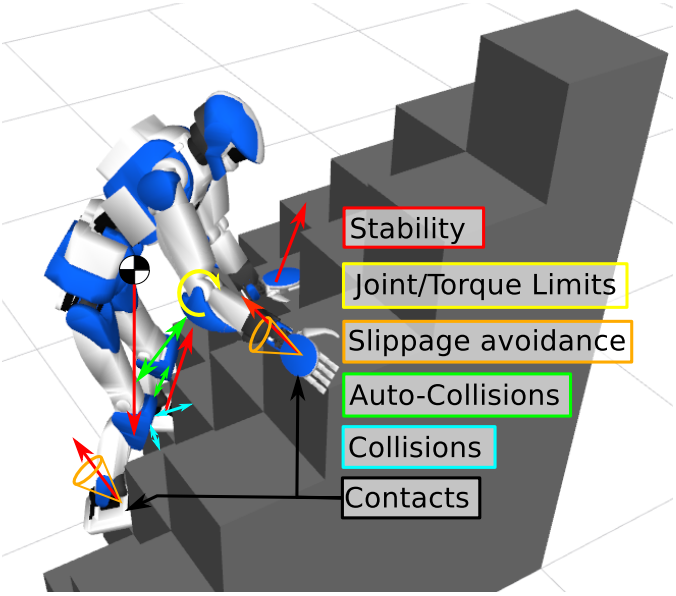


Fig. 1. Illustration of the different constraints involved in the Posture Generation problem.

Manuscript received Xxxxxxx, 2017; revised Xxxxx XX, 2017; accepted Xxxxxx XX, 2018. Date of publication Xxxxxxxx X, 2018; date of current version Xxxxxxxxx X, 2018. This paper was recommended for publication by Associate Editor X. Xxxxxxxx and Editor T. Murphey upon evaluation of the reviewer's comments.

This work was partially supported by the EU H2020 COMANOID project and the CNRS-AIST-Airbus Joint Research Program.

S. Brossette is with Wandercraft, Paris, France

A. Kheddar and A. Escande are, S. Brossette was, with the CNRS-AIST Joint Robotics Laboratory (JRL), UMI3218/RL, Japan

A. Kheddar is, S. Brossette was, also with the CNRS-University of Montpellier LIRMM Interactive Digital Humans team, France

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 00.0000/TRO.201X.0000000

a cost function f. This yield a nonlinear optimization problem that we name *Posture Generation* (PG) problem:

$$\begin{array}{l} \min\limits_{\mathbf{q},\xi,\mathbf{y}} f(\mathbf{q},\xi,\mathbf{y}) \\ \text{s.t. } (\mathbf{q},\xi,\mathbf{y}) \in \mathcal{Q} \end{array} \tag{10}$$

Usually, the PG problem is formulated over a Euclidean space. However, robot variables may be more appropriately expressed in non-Euclidean manifolds. An example of a typical manifold is the rotation space $SO(3)$, which is used to represent the orientation of the robot root, or to parametrize a ball joint. Another example is the 3D unit sphere on S^2 which is used to express the contact with a parametric surface in [8]; in [9] a human shoulder is parametrized on $S^2 \times \mathbb{R}$.

Formulating a non-Euclidean manifold optimization problem over $\mathbb{R}^n$ leads either to singularities or to the addition of constraints and variables to this problem, as detailed in the next section. Since manifolds are locally Euclidean, the properties of distance, derivatives, and more generally all the Euclidean geometry can be used locally. Usual *unconstrained* optimization methods have been adapted to work on non-Euclidean manifolds (e.g. [10], [11], [12], [13]). However, posture generation requires using *constrained* optimization together with manifolds. To the best of our knowledge, there is no method to solve such class of problems with generic manifolds (for specific ones, [14] proposes an extension of a classical approach to the special case $SE(3)$).

Our primary contribution consists in a Sequential Quadratic Programming (SQP) solver working directly on manifolds. We were largely inspired by the seminal work of Absil [13] for unconstrained optimization, that we adapt to the constrained case by borrowing from the existing literature, and in particular from the work of Fletcher [15]. A background motivation is to have our own optimization solver, that we can customize at will for robotic problems, leveraging modeling properties and approximations for a gain in time and robustness. The theoretical development and the sketch of our SQP on manifolds are given in Sec. II, while the different refinements needed to have a fully working solver are explained in Sec. III.

The definition of a PG problem as an optimization problem is often a cumbersome task. Our second contribution is a framework simplifying and automating part of this task so that the user can focus on the mathematical formulation of the constraints (Sec. IV). It makes it as convenient as possible to write custom PG problems and to develop custom constraints and cost functions. We take advantage of it to introduce new constraints for having contact on parametric surfaces and formulating force-based tasks (Sec. V). We also provide an efficient algorithm to compute the derivative of the torques with respect to the problem's variables. Finally, we present some evaluations of both the solver alone and the posture generator, as well as some scenarios illustrating their combined capabilities (Sec. VI).

Part of this paper was presented in [16]. The material of secs. II and IV are extended version of the previous work with more details and a few corrections. The rest of the paper is almost completely new. The example of Fig. 10 is borrowed from [16], the one from Fig. 11 is reworked from [8], with the addition of collision avoidance.

II. Constrained optimization on manifolds

We consider the generic problem

$$\begin{array}{l} \min\limits_{x \in \mathcal{M}} \quad f(x) \\ \text{subject to } \; l \leq c(x) \leq u \end{array} \tag{11}$$

where $\mathcal{M}$ is a n-dimensional smooth manifold and f and c are 1- and m-dimensional real-valued C^1 functions. Equality constraints are formulated by taking $l_i = u_i$.

Except for specific cases, the numerical approach to solve problem (11) for $\mathcal{M} = \mathbb{R}^n$ is iterative: it starts from an initial guess x_0 and performs successive steps $x^{(k+1)} = x^{(k)} + \mathbf{p^{(k)}}$, until convergence. The strategy to compute $\mathbf{p^{(k)}}$ at each iteration depends on the solver. This scheme cannot be readily applied to optimization over non-Euclidean manifolds. In what follows we explain why, and give an adaptation of the SQP approach to tackle general manifolds.

A. Representation problem

To manipulate elements of $\mathcal{M}$ we need a way to represent them in memory. We choose a representation space $\mathbb{E} = \mathbb{R}^r$ (with $r \geq n$) and a map, that is

$$\psi \; : \; \begin{array}{c} x \; \mapsto \mathbf{x} \\ \mathcal{M} \to \mathbb{E} \end{array}$$

According to the *Whitney embedding theorem* [17], one can take ψ as a diffeomorphism onto $\psi(\mathcal{M})$ provided that r is large enough. In the following, we identify abusively $\mathcal{M}$ with $\psi(\mathcal{M}) \subseteq \mathbb{E}$. Once the choice is made, we can also define a projection π from $\mathbb{E}$ to $\psi(\mathcal{M})$.

With this representation, it is tempting to simply transform the problem (11) as an optimization over $\mathbb{R}^r$ with the objective $f \circ \psi^{-1}$ and the constraint $c \circ \psi^{-1}$, to be resolved 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, which can lead to critical convergence problems;
(ii) $r > n$, then most elements of $\mathbb{E}$ do not represent an element of $\mathcal{M}$ and ψ cannot be surjective. To force the solution on $\mathcal{M}$, we need to add constraints equivalent to $\pi(x) = x$. 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 $x^{(k)}$ is an element of $\mathcal{M}$, $x^{(k)} + \mathbf{p^{(k)}}$ 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 $x^{(k)}$, one has to project it on $\psi(\mathcal{M})$ first, effectively computing $f \circ \psi^{-1} \circ \pi$ and $c \circ \psi^{-1} \circ \pi$. The composition by π is an additional burden in programming (see e.g. in [18]).

As an example, the set of 3D-rotations $SO(3)$ is a manifold of dimension 3. The following usual 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.

- Spatial twist or angle-axis ($\mathbb{E} = \mathbb{R}^3$), singularities when the rotation is $2k\pi$, $k \neq 0$.

For the last two cases, the convergence in the vicinity of a singularity can be disturbed because two infinitesimally close rotations can have very different representations (see example in Appendix A).

B. Local parametrization

Given $x \in \mathcal{M}$, we note $T_x\mathcal{M}$ the tangent space of $\mathcal{M}$ at x. It is a linear space, identifiable with $\mathbb{R}^n$. By definition of a smooth manifold, there exists a map φ_x from an open set U of $T_x\mathcal{M}$ to an open set of $\mathcal{M}$ such that $\varphi_x(0) = x$. φ_x provides a local parametrization for $\mathcal{M}$. Should the solution of (11) lies in $\varphi_x(U)$ for a given x, we could transform the objective and constraints into $f \circ \varphi_x$ and $l \leq c \circ \varphi_x(\mathbf{z}) \leq b$ and simply solve a problem over $\mathbb{R}^n$ with an off-the-shelf solver. However, finding a point x close enough to the solution is a problem in itself. A more systematic approach, and the driving idea of optimization on manifold, is to change the parametrization at each iteration: for each iterate $x^{(k)}$, we reformulate the problem as

$$\begin{aligned} \min_{\mathbf{z} \in T_{x^{(k)}}\mathcal{M}} \quad & f \circ \varphi_{x^{(k)}}(\mathbf{z}) \\ \text{subject to} \quad & l \leq c \circ \varphi_{x^{(k)}}(\mathbf{z}) \leq b \end{aligned} \tag{12}$$

which is a classical optimization problem over $\mathbb{R}^n$. Starting from $\mathbf{z} = 0$ (recall that $\varphi_{x^{(k)}}(0) = x^{(k)}$), we can perform one iteration to get $\mathbf{z^{(1)}}$, set $x_{k+1} = \varphi_{x^{(k)}}(\mathbf{z^{(1)}})$ and re-parametrize. The process is repeated until convergence is achieved. In the computation of the step $\mathbf{z^{(1)}}$, one needs to be careful about the local validity of the map: if $\varphi_{x^{(k)}}$ is defined on $U_k \subset T_{x^{(k)}}\mathcal{M}$, then $\mathbf{z^{(1)}}$ must be restricted to U_k.

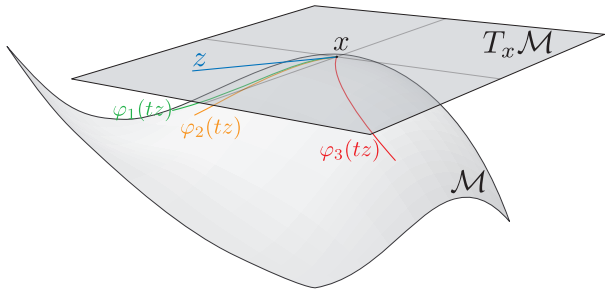


Fig. 2. There are many possible choices for φ_x but not all yield to a curve $\varphi_x(t\mathbf{z})$ which is going in the same direction as $\mathbf{z}$: φ_1 and φ_2 are correct choices, φ_3 is not.

At any point $x^{(k)}$, there is an infinite number of possible maps $\varphi_{x^{(k)}}$ (see Fig. 2), most of which are unsuitable to the optimization process. A suitable map needs to translate the direction and magnitude of $\mathbf{z}$ onto $\mathcal{M}$: the tangent vector at $t = 0$ of the curve $t \mapsto \varphi_{x_i}(t\mathbf{z})$ must be $\mathbf{z}$. This is known as the *local rigidity condition*. This led to the concept of *retraction* [13], a smooth mapping $\varphi : (x, \mathbf{z}) \mapsto \varphi_{\mathbf{x}}(\mathbf{z})$ such that for every x

1) $\varphi_x(0) = x$
2) $\forall \mathbf{z} \in T_x\mathcal{M}$, $\left.\dfrac{d\varphi_x(t\mathbf{z})}{dt}\right|_{t=0} = \mathbf{z}$

Even when restricting to retractions, there are still many choices of (family of) maps. The exponential map is an obvious candidate, but in the general case it is very expensive to compute, because it amounts to solving a differential equation. The literature offers cheaper alternatives for many classical manifolds (see e.g. [13], [19]).

Using a retraction φ, the iterative process uses the minimum number of variables and avoids additional constraints and parametrization-related singularities. Yet, we need to keep track of the iterates $x^{(k)}$ globally. Since the $x^{(k)}$ are guaranteed to be on $\mathcal{M}$, we can use a space $\mathbb{E} = \mathbb{R}^r$ and a map ψ, with $r > n$, without any drawback. The user can write the functions $f' = f \circ \psi^{-1}$ and $c' = c \circ \psi^{-1}$ as if they were functions from $\mathbb{E}$ without using a projection π beforehand. When using a new manifold, one needs to program once and for all $\psi \circ \varphi$. The Fig. 3 sums-up the different spaces and mappings.

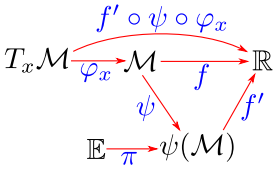


Fig. 3. Given a manifold $\mathcal{M}$, we use a global map ψ from an embedding space $\mathbb{E}$ to track its elements. π projects a generic element of $\mathbb{E}$ on $\psi(\mathcal{M})$. Given $x \in \mathcal{M}$, we also define a local map φ_x with which we can perform the computations. The functions actually implemented are f' and $\psi \circ \varphi$, which are functions between Euclidean spaces.

C. A basic SQP on manifold

There are several ways to compute a step $z^{(1)}$. We adopt the SQP approach which consists in making a quadratic-linear approximation of the problem (12) around the current iterate, corresponding to $\mathbf{z} = \mathbf{0}$, and solving it.

Let us first define the Lagrangian function

$$\mathcal{L}_x = f \circ \varphi_x - \lambda_-^T (c \circ \varphi_x - l) + \lambda_+^T (c \circ \varphi_x - u) \tag{13}$$

where $\lambda_-, \lambda_+ \in \mathbb{R}^m$ are the Lagrange multipliers associated with the lower and upper bounds respectively[1]. Let $H^{(k)}$ be the Hessian matrix $\nabla^2_{zz}\mathcal{L}_{x_k}$. Then the approximation writes as a Quadratic Program (QP)

$$\begin{aligned} \min_{\mathbf{z} \in \mathbb{R}^n} \quad & \frac{\partial f \circ \varphi_{x^{(k)}}}{\partial \mathbf{z}}(0)^T \mathbf{z} + \frac{1}{2}\mathbf{z}^T H^{(k)} \mathbf{z} \\ \text{subject to} \quad & 1 \leq c \circ \varphi_{x^{(k)}}(0) + \frac{\partial c \circ \varphi_{x^{(k)}}}{\partial \mathbf{z}}(0)\mathbf{z} \leq \mathrm{u} \end{aligned} \tag{14}$$

The basic SQP approach adapted to manifolds is as follows

1) set $k = 0$, and set $x^{(k)}$ to the initial value
2) solve the QP (14) for current $x^{(k)}$ to get $\mathbf{z}$ as well as the QP's Lagrange multipliers λ_-^{QP} and λ_+^{QP}
3) set $x^{(k+1)} = \varphi_{x^{(k)}}(\mathbf{z})$, $\lambda_-^{(k+1)} = \lambda_-^{\mathrm{QP}}$, $\lambda_+^{(k+1)} = \lambda_+^{\mathrm{QP}}$ and $k = k + 1$
4) if the convergence is not yet achieved go to step 2

[1] In practice, we use only the derivatives of the Lagrange function, so that λ_- and λ_+ are irrelevant, and we work with $\lambda = \lambda_+ - \lambda_-$. Since for $i = 1 \cdots m$, $(\lambda_-)_i = 0$ or $(\lambda_+)_i = 0$ depending on whether the upper or lower bound of the i-th constraint is active, λ_- and λ_+ are obtained from λ.

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

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

are functions between Euclidean spaces.

The gradient of $f \circ \varphi_x$ is

$$\frac{\partial f \circ \varphi_x}{\partial \mathbf{z}} = \frac{\partial f'}{\partial y}(\psi \circ \varphi_x) \times \frac{\partial (\psi \circ \varphi_x)}{\partial \mathbf{z}} \tag{15}$$

Because the problem is reparametrized at each iteration, the derivative of $\psi \circ \varphi_x$ is always computed at $\mathbf{z} = \mathbf{0}$, which usually allows for some fast specialized code.

The approach taken here is in theory applicable to any smooth manifold (including products of such manifolds), and is usable in practice whenever *tractable* retractions φ and vector transports (see below) can be implemented.

III. Practical implementation

The algorithm given in Sec. II-C is conceptual and works correctly when the initial iterate is close enough to the solution. In practice, several refinements need to be implemented to avoid divergence, deal with possible infeasibility in the QP approximation, alleviate the computations and speed-up the process. In this section, we detail these refinements. For each of these issues, several solutions exist, sometimes interdependently. We mostly follow the approach of Fletcher [15], that we adapt to work with manifolds, with some choices motivated by experiments on robotics problems, such as those described in Section VI. We provide a working and coherent set of refinements, carefully chosen and adapted to the context of non-Euclidean optimization from the wealth of the literature.

A. Convergence criterion

The convergence is considered to be achieved when the KKT conditions are satisfied[2] (see [20]). With the sign convention taken in definition (13), the KKT conditions for (11) are (with quantities taken at $z = 0$):

$$\begin{aligned} &\nabla_z \mathcal{L}_x = \nabla_z f \circ \varphi_x + (\nabla_z c \circ \varphi_x)^T (\lambda_+ - \lambda_-) = 0 \\ &l \leq c \circ \varphi_x \leq u \\ &\lambda_- \geq 0, \quad \lambda_-^T (c \circ \varphi_x - l) = 0 \\ &\lambda_+ \geq 0, \quad \lambda_+^T (c \circ \varphi_x - u) = 0 \end{aligned}$$

In practice, we account for floating point computations using tolerances that we scale to the problem. Given two user defined tolerance parameters τ_P and τ_D, and following [21], we define $\tau_x = \tau_P(1 + \|x\|_\infty)$ and $\tau_\lambda = \tau_D(1 + \|\lambda\|_\infty)$, where $\lambda = \lambda_+ - \lambda_-$. Our stopping criterion is

$$\|\nabla_z \mathcal{L}_x\| \leq \tau_\lambda \tag{16}$$

$$\forall i \begin{cases} & |c_i(x) - l_i| \leq \tau_x \quad \text{and } \lambda_i \leq -\tau_\lambda \\ \text{or} & l_i + \tau_x \leq c_i(x) \leq u_i - \tau_x \quad \text{and } |\lambda_i| \leq \tau_\lambda \\ \text{or} & |c_i(x) - u_i| \leq \tau_x \quad \text{and } \lambda_i \geq \tau_\lambda \end{cases} \tag{17}$$

[2]These conditions only ensure that we reached a critical point, not necessarily a minimum, much less the global minimum.

B. Globalization

The approximation (14) might be accurate only in a small neighborhood of $x^{(k)}$, so that the step found is not correct. The basic SQP algorithm might not converge, or even diverge rapidly. Adapting the algorithm to enforce its convergence (at least theoretically and not necessarily to the solution) is called globalization. It consists of two components: (i) a criterion to assess the correctness of the step, and (ii) a method to modify it if needed. In constrained optimization, the most classical criterion is to take a penalty function $\Phi(x, p)$ aggregating the objective and the violation of the constraints, where p is a penalty parameter balancing between objective and violation. A step is correct if $\Phi(x^{(k+1)}, p) < \Phi(x^{(k)}, p)$. For the method to converge to the solution, the parameter p needs to be adapted along the iterations and here lies the difficulty of this approach. In our early tests, we had troubles obtaining convergence on robotics problems. Instead, we opted for a quite recent criterion (see [15]) based on filters.

Let $v_i(x^{(k)})$, $i = 1 \cdots m$ be defined as $l_i - c_i(x^{(k)})$ if $c_i(x^{(k)}) < l_i$, $c_i(x^{(k)}) - u_i$ if $c_i(x^{(k)}) > u_i$ and 0 otherwise. It represents the violation of the i-th constraint. Noting $v = \sum v_i$, the optimization process can be seen as trying to satisfy two possibly conflicting objectives: decrease f and decrease v. The idea of the filter approach is to accept a step whenever the new iterate does not do worse than any previous one on both objectives, and does marginally better for one objective than at least one previous iterate. More precisely, we say that x_i *dominates* x_j if $f(x_i) \leq f(x_j) - \gamma v(x_j)$ or $v(x_i) \leq (1 - \gamma) v(x_j)$, where γ is a small number (typically 10^{-5}) and the terms in γ translates the "marginally better" concept. A *filter* is a set of values x_i where no element dominates another. A filter dominates an iterate x if any of its elements dominates x. We start with an empty filter F; at iteration k the step $\mathbf{z}$ is accepted if the new iterate $x^{(k+1)} = \varphi_{x^{(k)}}(\mathbf{z})$ is not dominated by F. In this case, $x^{(k+1)}$ is added to F, and all the elements of F it dominates are removed from the filter. If $x^{(k+1)}$ is dominated by F, the step is rejected and has to be modified.

There are two big classes of modifying methods: line-search and trust-region. In our case, we already have a notion of trust-region as we need the step $\mathbf{z}$ to remain within the set U_k on which $\varphi_{x^{(k)}}$ is defined, so we chose this method. The trust-region approach consists in restricting $\mathbf{z}$ to be in a region where the QP approximation (14) is "good enough". To simplify the approach and retain a QP formulation, we consider a rectangular trust region: given a user-defined vector $\mathbf{z_t} \in \mathbb{R}^\mathbf{n}$ with $\mathbf{z_t} \geq 0$, we want $-\rho \mathbf{z_t} \leq \mathbf{z} \leq \rho \mathbf{z_t}$, where ρ is a scalar accounting for the confidence we have in the QP approximation. When a step is rejected by the filter, we modify the trust region by taking $\rho = \max(\rho/2, \rho_{\min})$. If a step is accepted, and it was constrained by the trust region (i.e. $\mathbf{z_i} = \pm \rho \mathbf{z_{t,i}}$ for some i), we take $\rho = \min(2\rho, \rho_{\max})$. $\rho_{\min}$, $\rho_{\max}$ and the initial ρ are user-defined. Typical default value are 10^{-8}, 2 and 1. $\mathbf{z_t}$ encodes the typical step size of each variables. It allows to consider variables of different scales, by having an anisotropic trust-region, without having to write the functions f and c to take into account those scale differences.

For example, on a 60kg humanoid robots with joint and force variables, we typically take 0.1 for joints and 10 for forces.

The region U_k is also considered as rectangular: $\mathbf{z_u^-} \leq \mathbf{z} \leq \mathbf{z_u^+}$. We then defined $\mathbf{z}^- = \max(\mathbf{z_u^-}, -\rho\mathbf{z_t})$ and $\mathbf{z}^+ = \min(\mathbf{z_u^+}, \rho\mathbf{z_t})$, and QP (14) with a trust-region is given by:

$$\min_{\mathbf{z}\in\mathbb{R}^\mathbf{n}} \quad \frac{\partial f \circ \varphi_{x^{(k)}}}{\partial \mathbf{z}}(0)^T\mathbf{z} + \frac{1}{2}\mathbf{z}^T H^{(k)}\mathbf{z} \tag{18}$$

$$\text{subject to} \quad l \leq c \circ \varphi_{x^{(k)}}(0) + \frac{\partial c \circ \varphi_{x^{(k)}}}{\partial \mathbf{z}}(0)\mathbf{z} \leq u \tag{19}$$

$$\mathbf{z}^- \leq \mathbf{z} \leq \mathbf{z}^+$$

C. Restoration and second order correction

QP (18) can be infeasible because some constraints in (19) are incompatible or, more often, because the feasible points for (19) are outside of the trust region. When this is detected, we trigger a *restoration phase* whose goal is to reduce the violation of the constraints $l \leq c(x) \leq u$ until QP (18) becomes feasible again. We follow the approach in [15].

Infeasibility is detected by the QP solver when attempting to solve QP (18). From the solver output, we can define 2 lists of indexes $\mathcal{I}_l$ and $\mathcal{I}_u$ corresponding to constraints whose lower or upper bound cannot be satisfied, and a list $\mathcal{F}$ corresponding to feasible constraints. The idea of the restoration phase is to begin to solve the problem

$$\min_{x\in\mathcal{M}} \quad \sum_{i\in\mathcal{I}_l}(l_i - c_i(x)) + \sum_{i\in\mathcal{I}_u}(c_i(x) - u_i) \tag{20}$$

$$\text{subject to} \quad l^{\text{rest}} \leq c(x) \leq u^{\text{rest}} \tag{21}$$

with $l_i^{\text{rest}} = -\infty$ if $i \in \mathcal{I}_l$ and l_i otherwise, and likewise $l_i^{\text{rest}} = +\infty$ if $i \in \mathcal{I}_u$ and l_i otherwise. This corresponds to minimizing the violation of the infeasible constraints while enforcing the feasible ones.

The above problem is nonlinear, and we solve it with the same SQP. The restoration phase has its own filter F^{rest} and its own Hessian approximation (see next subsection), but shares the trust region parameter with the main phase. The difference with the main phase is that we stop to iterate as soon as we detect that QP (18) is feasible again. Also, at each iteration the sets $\mathcal{I}_l$, $\mathcal{I}_u$ and $\mathcal{F}$ can be updated to account for (linearized) constraints becoming feasible or infeasible in (18). For more details, the interested reader can refer to [22].

In restoration, if a step $\mathbf{z}$ is rejected by the filter F^{rest}, we attempt first to solve a modified QP problem and reduce the radius of the trust region only if the new step is rejected. In this modified QP, a linear term is added to the constraints (19) which approximates the second order in the Taylor development of $c \circ \varphi_x$. This is known as the second-order correction (SOC) [15]. SOC can also be used in the main phase, but, in our tests, we didn't find a particular advantage to use it.

We exit the restoration with an iterate $x^{(k)}$, that we add to the main filter F. In case $x^{(k)}$ is dominated by some elements of F, we remove them, what we refer to as *forcing* the filter.

D. Hessian computation

The computation of the exact Hessian matrices $H^{(k)}$ or $H_j^{(k)}$ (for each constraint and the cost function) is costly. It is common to only approximate them based on the previously computed steps and gradients. This is known as the quasi-Newton approach. In usual Euclidean settings, we define $s^{(k)} = x^{(k+1)} - x^{(k)}$ and $y^{(k)} = \nabla\mathcal{L}_{x^{(k+1)}} - \nabla\mathcal{L}_{x^{(k)}}$, and, given an approximation $\hat{H}^{(k)}$ of $H^{(k)}$, the approximation of $H^{(k+1)}$ is given by $\hat{H}^{(k+1)} = \Omega(\hat{H}^{(k)}, s^{(k)}, y^{(k)})$ where the function Ω is light to compute (it is typically a rank-1 or rank-2 update of $\hat{H}^{(k)}$) and depends on the method.

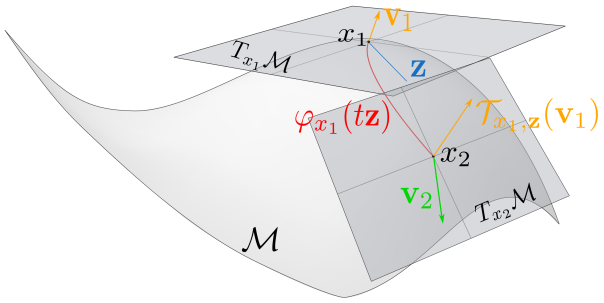


Fig. 4. The vector $v_1 \in T_{x_1}\mathcal{M}$ is transported along z. $\mathcal{T}_{x_1,z}(v_1)$ is an element of $T_{x_2}\mathcal{M}$ and can be compared to other elements of this tangent space such as v_2.

With manifolds, both definitions of $s^{(k)}$ and $y^{(k)}$ are usually not valid, the former because the subtraction might not be defined, the latter because the gradients live in different tangent spaces so that the subtraction is not meaningful. We extend the idea in [13] to the case of constrained optimization: with $z \in T_{x^{(k)}}\mathcal{M}$, the step from $x^{(k)}$ to $x^{(k+1)}$, we define

$$s^{(k)} = \mathcal{T}_z z \tag{22}$$

$$y^{(k)} = \nabla_z\mathcal{L}^{(k+1)}(0, \lambda^{(k+1)}) - \mathcal{T}_z\left(\nabla_z\mathcal{L}^{(k)}(0, \lambda^{(k)})\right) \tag{23}$$

where $\mathcal{T}_z$ is a vector transport along z, mapping a vector of $T_{x^{(k)}}\mathcal{M}$ to a vector of $T_{x^{(k+1)}}\mathcal{M}$ (see Fig. 4), see [13] for more details. This way, all quantities live in $T_{x^{(k+1)}}\mathcal{M}$ and we can perform meaningful operations. The resulting $\hat{H}^{(k+1)}$ can be viewed as the matrix of a quadratic form from $T_{x^{(k+1)}}\mathcal{M} \times T_{x^{(k+1)}}\mathcal{M}$ to $\mathbb{R}$.

We also use the usual damped update to account for the fact that we have a constrained optimization, and define $r^{(k)}$ as a modified $y^{(k)}$:

$$\theta^{(k)} = \begin{cases} 1 & \text{if } s^{(k)T}y^{(k)} \geq 0.2s^{(k)T}\tilde{H}^{(k)}s^{(k)} \\ \frac{0.8s^{(k)T}\tilde{H}^{(k)}s^{(k)}}{s^{(k)T}\tilde{H}^{(k)}s^{(k)} - s^{(k)T}y^{(k)}} & \text{otherwise} \end{cases}$$

$$r^{(k)} = \theta^{(k)}y^{(k)} + \left(1 - \theta^{(k)}\right)\tilde{H}^{(k)}s^{(k)}$$

with $\tilde{H}^{(k)}$ such that for $u \in T_{x^{(k+1)}}\mathcal{M}$, $\tilde{H}^{(k)}u = \mathcal{T}_z(\hat{H}\mathcal{T}_z^{-1}u)$.

We tried several update schemes: the BFGS formula

$$\hat{H}^{(k+1)} = \tau^{(k)}\left(\tilde{H}^{(k)} - \frac{\tilde{H}^{(k)}s^{(k)}s^{(k)T}\tilde{H}^{(k)}}{s^{(k)T}\tilde{H}^{(k)}s^{(k)}}\right) + \frac{r^{(k)}r^{(k)T}}{s^{(k)T}r^{(k)}}$$

with $\tau^{(k)} = 1$ for the original formula [20], or $\tau^{(k)} = \min\left(1, \frac{s^{(k)T}r^{(k)}}{s^{(k)T}\tilde{H}^{(k)}s^{(k)}}\right)$ for the so-called self-scaled version [23]; and the SR1 formula [20]

$$\hat{H}^{(k+1)} = \tilde{H}^{(k)} - \frac{(y^{(k)} - \tilde{H}^{(k)}s^{(k)})(y^{(k)} - \tilde{H}^{(k)}s^{(k)})^T}{(y^{(k)} - \tilde{H}^{(k)}s^{(k)})^T s^{(k)}}$$

Finally, we tried a mixed BFGS-SR1 scheme [24], but we did not get good results with it.

The above formulas approximate directly the Hessian of $\mathcal{L}$. Another approach is to use them to approximate individually the Hessian matrices of the cost function f and of every constraints c_i, before using the fact that $\nabla^2_{xx}\mathcal{L} = \nabla^2_{xx}f + \sum_i \lambda_i \nabla^2_{xx} c_i$, as proposed in [25]. We refer to this as the *individual update*, and coin the first approach *grouped update*. The different update options are evaluated on robotics problems in Section VI.

For most of the update schemes, the Hessian approximation $\hat{H}$ need not be positive definite. We regularize it as follows to obtain a positive definite approximation $\hat{H}'$: we perform first a Bunch-Kaufman factorization $P\hat{H}P^T = LBL^T$ where P is a permutation matrix, L is unit lower triangular and B is block diagonal with blocks of size 1×1 or 2×2[3], see [26]. We can then compute (in $\mathcal{O}(n)$) the eigenvalue decomposition $B = QDQ^T$, with Q a block diagonal orthogonal matrix with the same structure as B, and we define the diagonal matrix D' with diagonal elements $d'_{ii} = \max(d_{ii}, \mu_{\min})$ where the d_{ii} are diagonal elements of d and $\mu_{\min} > 0$ is user-defined. We then define $L' = LQD'^{\frac{1}{2}}$, and $\hat{H}' = P^T L' L'^T P$. $\hat{H}'$ is positive definite because it is similar to D' whose eigenvalue are all strictly positive (Sylvester's law of inertia). Computing L' is cheap ($\mathcal{O}(n^2)$) given the sparsity of D' and Q.

It is not uncommon for QP solvers to accept directly the Cholesky factorization (P, L'), as they would otherwise perform it internally. This is the case for the solver `LSSOL` [27] we use in this work. As a consequence, the overhead cost of the regularization is barely noticeable.

E. Overall algorithm

The full SQP algorithm is depicted in Fig. 5.

Proving formally the convergence of the algorithm is out of the scope of this paper. Intuitively, it behaves locally like an SQP. In particular, when close enough to the solution, the algorithm will take steps equal or very close to those taken by a classical SQP with a fixed local parametrization centered at the solution (i.e. problem (12) formulated at the solution x^*), because the effects of the change of local parametrization become negligible. Thus the algorithm should have the same convergence properties as a classical SQP.

IV. Posture Generator

Writing a PG can easily become cumbersome without the appropriate tools: many data and functions need to be specified and there are many sources of errors. 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's manifold. A fair amount of bookkeeping is always necessary, which should not be the charge of the user. We propose a PG architecture automating most of the problematic tasks so that the user can focus on the mathematical formulation of the problem.

[3]obtaining B as a diagonal matrix is not numerically stable for Cholesky-like decomposition of indefinite matrices

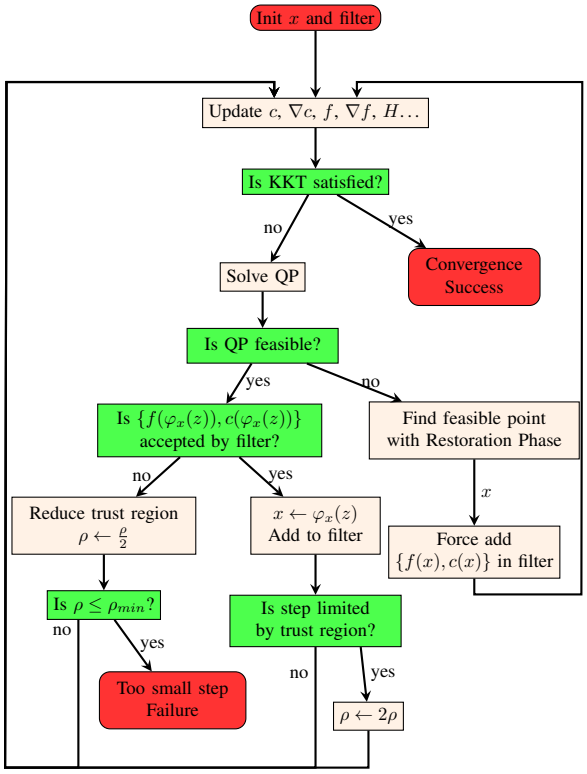


Fig. 5. Main SQP Loop

A. Geometric Expressions

The description of a task (i.e. a robotic constraint) is a mathematical object called *expression*. An expression is the result of a series of elementary operations on or between geometric and physical objects, called *features*, such as vectors, points, wrenches or frames. Provided that the elementary operations, features, and expressions are well-defined, and that the structure of the series of operations is registered in a system of expression graph, one can systematically compute the resulting expressions' values and derivatives by chain rule. The main idea behind such a framework is to separate the purely mathematical logic layer from the geometric one, in order to automate the evaluations and differentiations of complex operations between the geometric features.

The evaluation of a given feature in a given frame is represented by a mathematical expression. We consider 5 types of expressions which can be either variable or constant:

- Scalar, a 1-dimensional element of $\mathbb{R}$
- Coordinate, 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

These expressions can be combined with each other through mathematical operations to create other expressions. Meaningful unary (norm, inverse, opposite...) and binary (addition, subtraction, dot product, cross product...) operations are implemented, along with their derivatives by chain rule. In addition, we have a Function class for more complex and custom expressions. In particular, we use a function operation

expressing $\mathbf{q} \mapsto T_i(\mathbf{q})$ where T_i is the transformation between the reference frame of the robot and the frame of its i-th body to represent the location of each body of a robot[4]. We also use the combinations of elementary operations to define the computation graph used for evaluations.

The expressions and operations constitute the mathematical layer, on top of which the geometric layer is built. The geometric layer is composed of geometric and physical objects used in PG problems, which can be represented by elementary geometric quantities, or features. It aims at abstracting the mathematical layer so that the user can focus on the physical formulation of the problem. Any feature is defined in a reference frame by a mathematical expression but otherwise exists independently of its expression in a given frame.

The geometric layer of our framework in composed of 4 features, all defined in their respective reference frames:

- A Frame, defined by a Transformation expression;
- A Point, defined by a Coordinate expression;
- A Vector, defined by a Coordinate expression;
- A Wrench, defined by a pair of Coordinate expression.

A special World Frame object serves as the starting reference frame. Since each frame is defined with respect to another reference frame, the geometric layer intrinsically constructs a tree of interconnected frames. This structure allows computing the numerical evaluation of any geometric object in any given frame. It allows the geometric layer to ensure that all the features are expressed in the correct frames before the mathematical layer performs the corresponding operations on their associated expressions. 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.

Let us consider a robot that has to look at a specific point P_e of the environment with its camera, defined by a point P_r and a vector V_r. Using the elementary operations of subtraction between points (returns a vector), and the dot product between vectors (returns a scalar), the constraint writes as $(P_e - P_r).V_r = 0$. The user needs only to create those geometric objects, attached to the frames of the robot or of the environment, and write the code `(Pe-Pr).dot(Vr)` to generate the function. Given any variable object `v`, `(Pe-Pr).dot(Vr).diff(v)` returns the differential of the expression w.r.t. `v`. This makes the writing of constraints very easy.

B. Automatic mapping

The complete manifold $\mathcal{M}$ on which the optimization takes place is the Cartesian product of all the sub-manifolds involved in all the functions describing the optimization problem, and so is its representation space $\mathbb{E}$ and the optimization vector $x \in \mathbb{E}$:

$$\begin{aligned} \mathcal{M} &= \mathcal{M}_1 \times \mathcal{M}_2 \times \mathcal{M}_3 \times \dots \\ \mathbb{E} &= \mathbb{E}_1 \times \mathbb{E}_2 \times \mathbb{E}_3 \times \dots \\ x^T &= [x_1^T, x_2^T, x_3^T \dots] \end{aligned} \tag{24}$$

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

The solver does not 'see' the different sub-manifolds. It considers all functions as defined on the complete manifold $\mathcal{M}$, represented by $\mathbb{E}$. Conversely, for the developer, it is quite inconvenient and a potential source of error to write a function on the complete manifold because of the need to manage indexes in order to feed the function with the correct data from the complete optimization vector x, and also because the complete manifold $\mathcal{M}$ can only be formed once the PG problem has been completely defined. Thus, when the function is implemented, the complete $\mathbb{E}$ is not known. A user-written function f is usually defined on a subspace of $\mathbb{E}$, say $\mathbb{E}_I = \mathbb{E}_i \times \mathbb{E}_j \times \mathbb{E}_k \dots$ which is minimalist for that function, and should not account for unrelated manifolds. Our automatic mapping tool keeps track of all the necessary mappings for each function added and upon the instantiation of the problem, generates the correct projection functions $\pi_I : \mathbb{E} \to \mathbb{E}_I$ 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. 6

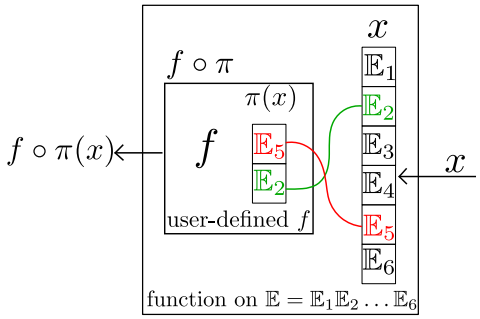


Fig. 6. Automatic variable mapping: example of a function f taking values from $\mathbb{E}_5 \times \mathbb{E}_2$ and written as such, which is mapped into a function from $\mathbb{E}_1 \times \dots \times \mathbb{E}_6$.

C. Problem Generator

The problem generator is in charge of building the optimization problem for each PG problem to solve. It registers all the variables and constraints of the problem as well as the cost function. Each function is likely to bring additional variables with it. For each contact contributing to the static equilibrium, a variable defined on (a subspace of) $\mathbb{R}^6$ representing its wrench is added to the problem. An additional function on that variable can be added for the slippage avoidance constraint. Once the registration is finished, the complete manifold on which the optimization takes place is generated as the Cartesian product of all the unique sub-manifolds on which the functions' variables are defined. The information of the automatic mapping is then used to 'plug' each function with the correct sub-manifold. Hence, 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[5].

The previous tools simplify the development of PG problems and help to avoid implementation mistakes, greatly facilitating the implementation of new or more complex constraints.

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

V. Extensions to new constraints

In this section, we first leverage the variable management capabilities of our PG and propose a constraint formulation that let the the optimization solver chose the contact location on a surface by using of additional variables. We also discuss the formulation of constraints and cost functions involving forces and torques and their implication: the need to compute the derivative of the joint torques, for which we propose an algorithm that does not rely on the computationally expensive differentiation of point's Jacobians.

A. Contact location parametrization

The most common type of contact constraint is between two surfaces S_1 and S_2. Let $F_i = \{O_i, (x_i, y_i, z_i)\}$ be a frame defined on each surface S_i, with O_i on the surface and z_i its outbound normal. Writing relations between geometric quantities defined in frames allows describing a large variety of contacts, such as planar or fixed contacts. For example, a planar contact between S_1 and S_2 can be expressed as:

$$\begin{cases} \overrightarrow{O_1O_2} \cdot \vec{z_1} = 0 \\ \vec{z_2} \cdot \vec{z_1} \leq 0 \\ \vec{z_2} \cdot \vec{x_1} = 0 \\ \vec{z_2} \cdot \vec{y_1} = 0 \end{cases} \tag{25}$$

This formulation is valid if the physical entities in contact are either a fixed point or a flat surface, because the quantities in eq. 25 are invariant w.r.t the location of the contact points. With non-flat surfaces, the locations of the frames used in the formulation matter if the contact points are not fixed. Taking advantage of our framework, we propose to parameterize the location of the contact points with an additional variable $u_S \in \mathcal{M}_S$, whose manifold will automatically be added to the optimization problem. This gives our solver the ability to autonomously choose an optimal contact location on a non-flat surface as part of its resolution process. Depending on its shape, the manifold $\mathcal{M}_S$ on which the surface is parametrized can be $\mathbb{R}$, $\mathbb{R}^2$, S^2. Then we can write the contact constraint on a frame parametrized as follows:

$$\mathcal{F}(u_S \in \mathcal{M}_S) = \{O(u_S), (x(u_S), y(u_S), z(u_S))\} \tag{26}$$

This approach can readily formulate contacts with surfaces represented by closed-form equations, such as spheres, superellipsoids... as presented in [16] and in section VI-C1.

Additionally, in [8] we present a method using a Catmull-Clark subdivision surfaces along with a custom raycasting algorithm to systematically generate a smooth parameterization of the surface of any object described by a star-convex mesh. This opens the way to a large range of applications of contacts in robotics. We illustrate it briefly in Section VI-C2.

B. Force Constraints

If we command an overall posture in which a robot link (e.g. tool used) is asked to apply a threshold of force in a given direction, it must be computed to meet such a requirement. One may also want to limit a force on a given contact or figuring if a robot is able to carry an object of a given weight (see Section VI-D1). Forces can be defined by using a wrench element in our expression framework. In many cases, satisfying the Newton-Euler equation is considered to be enough to have a static equilibrium of a posture, but when constraints on forces are involved, it becomes necessary to check the torque limits. Indeed, without torque limits, if we consider a robot pushing on two opposite walls, a configuration where the forces applied on each wall are opposite and arbitrarily large can be a solution, although not feasible in practice.

Having constraints on forces in our PG makes it necessary to compute the joints' torques, as well as their derivatives.

C. Joint Torque Jacobian Matrix Computation

The computation of the joint torque's Jacobian matrices is often approximated using finite differences, automatic differentiation, or some complex algorithms computing the derivatives of point's Jacobian matrices explicitly. We propose an algorithm that computes the exact torque Jacobian matrices.

For a given configuration $\mathbf{q}$ and a set of external wrenches w_i^{ext}, the joint torques can be computed recursively by using a modified version of the *recursive Newton-Euler algorithm* presented in [28], where the time derivative terms are ignored.

Let us introduce some notations:

- n_J and n_B: respectively the numbers of joints and bodies
- B_i and J_i: respectively the body and the joint of index i
- pred(i) and succ(i): respectively the index of the predecessor and successor bodies of J_i
- ${}^jX_i$, ${}^jR_i$ and ${}^jt_i$: respectively the transformation from the reference frame of B_i to body B_j, its rotation part, and its translation part
- $\wedge$: the cross-product operator
- For any 3D vector v: $\hat{v}$, the 3×3 skew-symmetric matrix such that $\forall u \in \mathbb{R}^3,\ \hat{v}u = v \wedge u$
- I_i: the inertia matrix of B_i
- S_i: the motion subspace of J_i
- $a_g = \begin{bmatrix} \mathbf{a_c} \\ \mathbf{a_f} \end{bmatrix}$: the acceleration field, with $\mathbf{a_c}$ its rotation part, and $\mathbf{a_f}$ its translation part
- $w_i^{\text{ext}} = \begin{bmatrix} \mathbf{m_i^{\text{ext}}} \\ \xi^{\text{ext}} \end{bmatrix}$: the external wrench applied on B_i, with the moment part $\mathbf{m_i^{\text{ext}}}$, and the resultant part ξ^{ext}
- $\dim(.)$: the dimension of .

For any joint J_i, the transformation from $B_{\text{pred}(i)}$ to $B_{\text{succ}(i)}$ denoted $X_i^{\text{PtS}} = {}^{\text{succ}(i)}X_{\text{pred}(i)}$ can be decomposed into a static transformation X_i^x from the reference frame of the predecessor body to the one of the joint, and a variable transformation $X_i^J(q)$, q being the joint parameter, such that $X_i^{\text{PtS}} = X_i^J(q)X_i^x$.

Let A and B be the Cartesian frames with origins O_A and O_B respectively, $\mathbf{t}$ be the coordinate vector expressing $\overrightarrow{O_AO_B}$ in A, $\mathbf{R}$ be the rotation matrix that transforms 3D vectors from A to B coordinates. The transformation from A to B for a motion vector and its inverse are defined as follows:

$${}^BX_A = \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ -\mathbf{R}\hat{\mathbf{t}} & \mathbf{R} \end{bmatrix};\ {}^BX_A^{-1} = {}^AX_B = \begin{bmatrix} \mathbf{R}^T & \mathbf{0} \\ \hat{\mathbf{t}}\mathbf{R}^T & \mathbf{R}^T \end{bmatrix} \tag{27}$$

The transformation from A to B for a force vector, and its inverse are defined as follows:

$$ {}^{B}X_{A}^{*} = \begin{bmatrix} \mathbf{R} & -\mathbf{R}\hat{\mathbf{t}} \\ \mathbf{0} & \mathbf{R} \end{bmatrix}; \quad {}^{B}X_{A}^{-*} = \begin{bmatrix} \mathbf{R}^T & \hat{\mathbf{t}}\mathbf{R}^T \\ \mathbf{0} & \mathbf{R}^T \end{bmatrix} \tag{28} $$

The torque computation is done in two steps: (i) the generalized forces on all bodies are computed, then (ii) the effect of each body on its parent is propagated through the joints and the torques are computed by Alg. 1. We differentiate this algorithm w.r.t. $\mathbf{q}$ and any other variables of the problem that we denote generically $\mathbf{y}$ (e.g. force variables, parametrization of non-flat surfaces, configuration of another robot, etc). We assume that the derivatives of $\mathbf{m}_i^{\text{ext}}$ and ξ_i^{ext}: $\frac{\partial \mathbf{m}_i^{\text{ext}}}{\partial \mathbf{q}}$, $\frac{\partial \mathbf{m}_i^{\text{ext}}}{\partial \mathbf{y}}$, $\frac{\partial \xi_i^{\text{ext}}}{\partial \mathbf{q}}$, and $\frac{\partial \xi_i^{\text{ext}}}{\partial \mathbf{y}}$ are given. Besides $\mathbf{m}_i^{\text{ext}}$ and ξ_i^{ext}, all the quantities of the algorithm depend only on $\mathbf{q}$.

Algorithm 1 Inverse Static Algorithm

$\forall i \in [0, n_B],\ \xi_i^G = I_i {}^{i}X_0 a_g - {}^{i}X_0^* w_i^{\text{ext}}$
for $i = n_J - 1 : 0$ **do**
  $\tau_i = {\xi_i^G}^T S_i$
  $\xi_{\text{pred}(i)}^G = \xi_{\text{pred}(i)}^G + X_i^{\text{PtS}}(\mathbf{q})^{-*} \xi_i^G$
end for

The column j of the Jacobian matrix of a body i represents the derivative of its rotation ${}^{i}\mathbf{R}_W$ and translation ${}^{i}\mathbf{t}_W$ w.r.t $\mathbf{q}_j$, that we denote respectively $\omega_{i,j}$ and $v_{i,j}$.

$$ \mathbf{Jac}_i^W = \begin{bmatrix} \omega_{i,0} & \cdots & \omega_{i,j} & \cdots & \omega_{i,\text{dof}} \\ v_{i,0} & \cdots & v_{i,j} & \cdots & v_{i,\text{dof}} \end{bmatrix} \tag{29} $$

To differentiate the Inverse Static algorithm, we first expand all the matrix products of its equations, and then differentiate each term. The following relations (30) are useful when differentiating Alg. 1, into Alg. 2.

$$ \begin{aligned} \frac{\partial {}^{i}\mathbf{R}_W \mathbf{u}}{\partial q_j} &= {}^{i}\mathbf{R}_W \mathbf{u} \wedge \omega_{i,j} = {}^{i}\mathbf{R}_W \widehat{\mathbf{u}} \omega_{i,j} \\ \frac{\partial {}^{i}\mathbf{R}_W {}^{i}\mathbf{t}_W \wedge \mathbf{u}}{\partial q_j} &= {}^{i}\mathbf{R}_W \mathbf{v}_{i,j} \wedge \mathbf{u} + {}^{i}\mathbf{R}_W \left({}^{i}\mathbf{t}_W \wedge \mathbf{u} \right) \wedge \omega_{i,j} \\ &= -{}^{i}\mathbf{R}_W \widehat{\mathbf{u}} \mathbf{v}_{i,j} + {}^{i}\mathbf{R}_W \widehat{\left(\widehat{{}^{i}\mathbf{t}_W} \mathbf{u} \right)} \omega_{i,j} \end{aligned} \tag{30} $$

Alg. 2 allows to compute the exact Jacobian of the torques in a robot without having to compute the derivatives of the forces application points.

Algorithm 2 Torque Jacobian Matrix Computation

Compute the Jacobian matrix of generalized forces
for $i = 0 : n_B$ **do**
  $\xi_i^G = I_i {}^{i}X_0 a_g - {}^{i}X_0^* w_i^{\text{ext}}$
  $M = \begin{bmatrix} {}^{i}\mathbf{R}_W \widehat{\mathbf{a_c}} & 0 \\ -{}^{i}\mathbf{R}_W \widehat{\left(\widehat{{}^{i}\mathbf{t}_W} \mathbf{a_c} \right)} + {}^{i}\mathbf{R}_W \widehat{\mathbf{a_f}} & {}^{i}\mathbf{R}_W \widehat{\mathbf{a_c}} \end{bmatrix}$
  $N = \begin{bmatrix} -{}^{i}\mathbf{R}_W \widehat{\left(\widehat{{}^{i}\mathbf{t}_W} \xi_{\mathbf{i}}^{\text{ext}} \right)} + {}^{i}\mathbf{R}_W \widehat{\mathbf{m_i^{ext}}} & {}^{i}\mathbf{R}_W \widehat{\xi_{\mathbf{i}}^{\text{ext}}} \\ {}^{i}\mathbf{R}_W \widehat{\xi_{\mathbf{i}}^{\text{ext}}} & 0 \end{bmatrix}$
  $\text{fJac}_i^W = \begin{bmatrix} \text{Jac}_i^W & \mathbf{0}_{6 \times \dim(y)} \end{bmatrix}$
  $\frac{\partial \xi_i^G}{\partial x} = (I^W M - N) \text{fJac}_i^W + {}^{i}X_0^* \frac{\partial w_i^{\text{ext}}}{\partial x}$
end for

Compute the Jacobian of torques
for $i = n_J - 1 : 0$ **do**
  $\text{fS}_i = \begin{bmatrix} S_i & \mathbf{0}_{6 \times \dim(y)} \end{bmatrix}$
  $\frac{\partial \tau_i}{\partial x} = \text{fS}_i^T \frac{\partial \xi_i^G}{\partial x}$
  $h_i = \left(\mathbf{R}_i^{J^T} \xi_i^G \right)$
  $K = \begin{bmatrix} \widehat{(\mathbf{R}_i^{J^T} \mathbf{m}_i^G)} + h_i \cdot \mathbf{t}_i^{J^T} - \left({h_i}^T \cdot \mathbf{t}_i^J \right) \mathbf{I}_3 & -\widehat{h_i} \\ \widehat{h_i} & 0 \end{bmatrix}$
  $\xi_{\text{pred}(i)}^G \leftarrow \xi_{\text{pred}(i)}^G + X_i^{PtS}(\mathbf{q})^{-*} \xi_i^G$
  $\frac{\partial \xi_{\text{pred}(i)}^G}{\partial x} \leftarrow \frac{\partial \xi_{\text{pred}(i)}^G}{\partial x} - X_i^x\ K\ \text{fS}_i$
end for

VI. Evaluation and Experimentation

A. Solver's evaluation: the cubes stacking problem

In a typical robotics PG problem, the only non-Euclidean manifold encountered is the instance of $SO(3)$ representing the 3D rotation of each robot's free-flyer. The other variables (joint angles and contact forces) can be represented in real-spaces. Because of the predominance of Euclidean manifold, and the complexity of the problem's equations, it is unlikely for a formulation on manifolds to converge to the same local minimum as a formulation on real-spaces, and if they do, the influence of the formulation should be minor. Thus, in order to compare those two formulations, we consider a toy problem involving many non-Euclidean manifolds and simple equations: stacking a set of unit cubes in a box defined by a set of planes, with no interpenetration. Each cube is defined on $\mathbb{R}^3 \times SO(3)$. For any cube C_i, we denote $V_i = \{v_0, v_1, \ldots, v_7\}$ the set of all its vertices, $\vec{t_i} \in \mathbb{R}^3$ its translation and $R_i \in SO(3)$ its rotation w.r.t the world frame. A plane $P_j = \{d_j, \vec{n}_j\}$ is described by its normal $\vec{n}_j \in S^2$, and by $d_j \in \mathbb{R}$, its signed distance to the origin along $\vec{n}_j$. The constraint for a cube C_i to be 'above' a plane P_j is of dimension 8 and can be written as:

$$ \forall v \in V_i,\ (\vec{t_i} + R_i v) \cdot \vec{n}_j \geq d_j \tag{31} $$

For the cubes to be inside a box defined by a set of planes $\mathcal{P}$, we constrain each cube's vertex to be above each plane of $\mathcal{P}$.

To avoid interpenetration, we could use classical collision avoidance constraints, but the use of the exact mesh of the cubes would generate gradient discontinuities of these constraints. Approximating the mesh with STP-BV [29] would allow avoiding those discontinuities. Yet, the closer to the exact mesh is the STP-BV approximation, the closer to discontinuity is the gradient. Instead, we propose another approach that uses non-Euclidean manifolds: for each pair of cubes (C_i, C_j), we require a plane $P_{ij} = \{d_{ij}, \vec{n}_{ij}\}$ to separate them. To do so, we constrain C_i to be above P_{ij}, and C_j to be above $P_{ij}^- = \{-d_{ij}, -\vec{n}_{ij}\}$, both using constraint (31).

To simulate gravity, we minimize the potential energy of all the cubes (simplified by a factor mass times gravity):

$$ f = \sum_i \vec{t_i} \cdot \vec{z} \tag{32} $$

We consider the problem of stacking n cubes in an open-top box composed of 5 planes (the ground and 4 walls). In Fig. 7, we illustrate the case where $n = 3$ cubes.

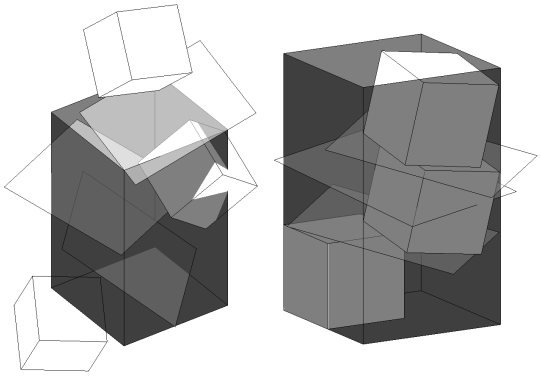


Fig. 7. Stacking 3 cubes in a box, separating each pair of cubes by a virtual plane. Initial (left) and final (right) configurations.

There is $n(n-1)/2$ separating planes, one per pair of cubes. Thus the search manifold is:

$$\mathcal{M} = \left(\mathbb{R}^3 \times SO(3)\right)^n \times \left(\mathbb{R} \times S^2\right)^{\frac{n(n-1)}{2}}$$

The problem contains 5 constraints of dimension 8 per cube to fit them in the box and $n(n-1)/2$ constraints of dimension 16 to avoid interpenetrations. We have a problem of dimension $4.5n + 1.5n^2$ with $32n + 8n^2$ constraints.

We also formulate this problem over $\mathbb{R}^n$ to compare the resolution with and without the use of manifolds. Each variable on $SO(3)$ is replaced by a variable on $\mathbb{R}^4$, while each variable on S^2 is replaced by one on $\mathbb{R}^3$. In both cases, a norm-1 constraint on the variable is added to the problem to force those variables on the manifolds. This results in a problem on

$$\mathcal{M} = \left(\mathbb{R}^3 \times \mathbb{R}^4\right)^n \times \left(\mathbb{R} \times \mathbb{R}^3\right)^{\frac{n(n-1)}{2}} = \mathbb{R}^{5n+2n^2}$$

that is a problem of dimension $5n + 2n^2$ with $32.5n + 8.5n^2$ constraints, which is $\frac{n(n+1)}{2}$ more variables and constraints relatively to the manifold formulation.

We solve these two problems for different numbers of cubes and compare the results in terms of number of iterations before convergence, convergence time, and time spent per iterations. In each resolution, both problems (Real space and Manifold formulations) start with the same initial guess. The resolutions are carried out with our solver, `PGSolver`, for both formulations with the same set of parameters, and with an off-the-shelf SQP solver, `CFSQP` [30], for the Real-space formulation only, using `CFSQP` default parameters. The initial positions of the cubes are chosen randomly, and each plane is initialized at a position between the two cubes it separates.

The results of these tests are plotted in Fig. 8. With 300 resolutions per case, about 98% converged when using manifolds versus 99.5% for non-manifold using `PGSolver`. The success rates of `CFSQP` drop incrementally from 100% for 2 cubes to 70% for 7 cubes. Concerning the resolutions with our `PGSolver`, we observe that the numbers of iterations are sensibly similar for the two types of resolutions. Yet, the time spent per iteration is consistently smaller in the case

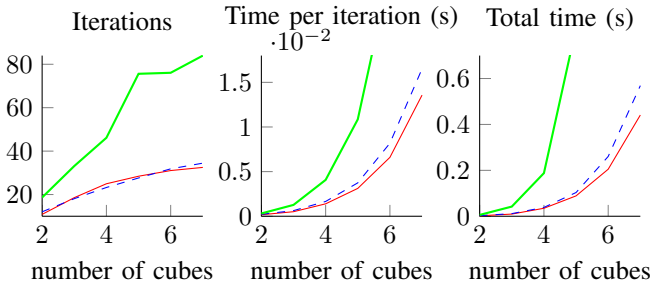


Fig. 8. Comparison of resolutions with and without using manifolds: red curves are the results with `PGSolver` on manifolds; blue dashed curves are those of the Real space; the green thick curves are with `CFSQP` on Real space.

of manifolds, which is in agreement with our expectations. Subsequently, the convergence time is consistently shorter for the formulation with manifolds. The resolutions with `CFSQP` take 4 times more iterations and each iteration is on average 3 times longer than with our `PGSolver`. Resolutions with `PGSolver` on manifolds are on average 7 times faster than with `CFSQP` on Real-space for this specific type of problem.

This study shows that on a toy example, optimization on manifolds with our `PGSolver` not only allows the user to benefit from a simple and more intuitive formulation of the problem, but it also outperforms the classical approach (using `CFSQP`) in terms of success rates and convergence speed.

B. Posture Generator's evaluation

We tested our PG in different types of feasible problems, and compare the success rates and computation times of different resolution approaches. To generate feasible problems, we compute locations reached by the robot's end effectors (feet and grippers) for a random configuration q_{rand} within its joint limits. We then compute different PG problems to find configurations with 2 (feet), 3 (feet and right gripper) and 4 (feet and grippers) contacts at the aforementioned locations. In particular, the contacts on the foot are fixed, unilateral, and with friction constraints (i.e. contact forces must be within friction cones), whereas the contacts on the grippers are fixed bilateral contacts (i.e. forces can be applied in any direction). Those contact constraints are formulated with the equations (25), whose quantities are all functions of the robot's variables $q \in \mathcal{M}$. If we were using a framework with optimization on Euclidean spaces, those functions would need to be composed with a projection on $\psi(M)$ and additional constraints would need to be added to the problem, as explained in II-A. We attempt a resolution for each PG problem, using the q_{rand} as initial guesses. If a solution q^* is found (the problem is feasible), it is saved to be used in the test.

Yet, using q_{rand} to generate the problem results in very twisted configurations of the robot that do not reflect 'useful' postures. We computed another set of feasible problems in which the value of q_{rand} is taken closer to the half-sitting posture: $q_{\text{rand}} \leftarrow \frac{1}{2}\left(q_{\text{rand}} + q_{\text{half-sitting}}\right)$. This resulted in better success rates and postures, which implies that the initial guess and its distance from the solution, that we name the *initial distance*, impact the resolution of PG problems. We define the

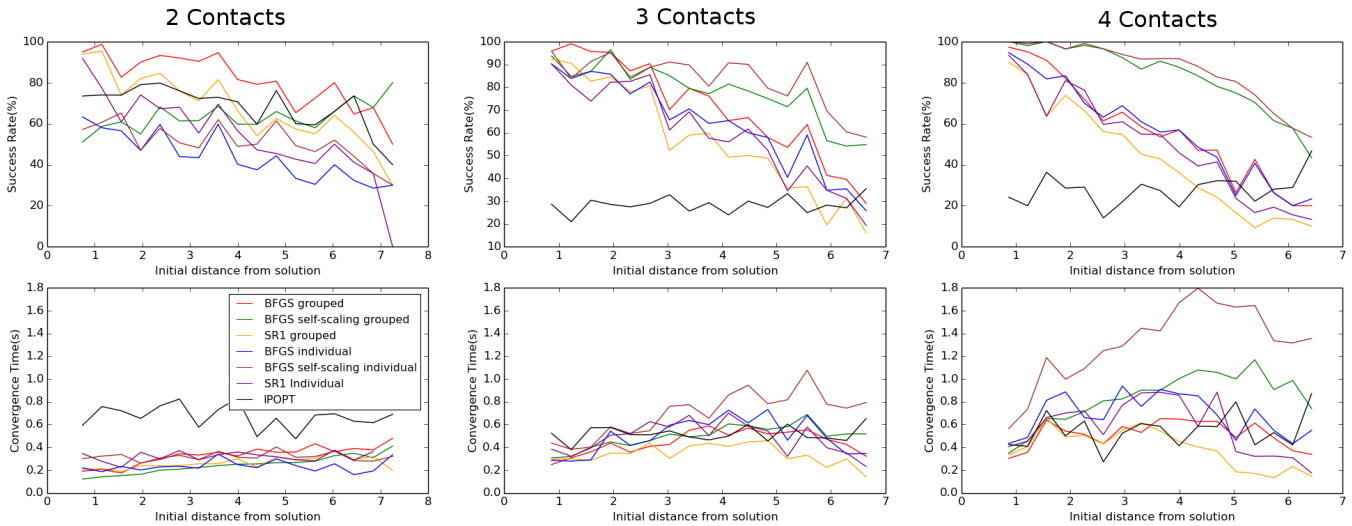


Fig. 9. Comparison of success rates and convergence times of the PG resolution w.r.t the distance between the initial guess and the solution for sets of feasible problems with 2, 3 and 4 contact constraints and for different choices of Hessian update methods in our solver, and also with the IPOPT solver [31].

initial distance for a problem as $d = \|q_0 - q^*\|^2$, where q_0 is the initial guess and q^* is a solution.

In order to highlight the effect of the initial distance on the success rate and convergence time of our PG, we solve the feasible problems several times, with an initial guess iteratively closer to a solution. For a random initial guess q_0, we initialize the problem with $q_0^{n\%} = nq_0 + (1-n)q^*$, with n taking successively the values 1.0, 0.8, 0.6, 0.4, and 0.2.

We compare the results obtained with three types of Hessian updates, namely BFGS, BFGS with self-scaling and SR1; for each, we use grouped and individual updates in restoration. In addition, we solved those problems with the framework proposed in [32], that formulates the same problems on real-spaces and uses the IPOPT solver [31]. We gather the results after solving 250 different PG problems, 5 times each, starting increasingly close to the solution. We sample the range of initial distances into 20 segments of same length and report the average results on each segment as a data point in Fig. 9.

Although no Hessian update method clearly dominates the others, we can observe that for the most constrained problems, with 3 and 4 contacts, the BFGS with self-scaling methods present the best success rates but are slower than other methods, whereas for problems with only 2, the BFGS method outperforms the others. Running such experiments allows the user to devise strategies to tune the solver optimally for a given family of problems. Here, it suggests that for solving PG problems in which only the feet of the robot are used for contact, an update of type BFGS should be used, while for problems in wich hands also play a role in contacts, a BFGS with self-scaling is preferable. In no cases did the rank-1 update SR1 provide the best results.

Our framework arguably performs better in all cases w.r.t the framework of [32] that we denote IPOPT. With 2 contacts, the success rates are similar, but the convergence times of IPOPT are much longer. With 3 and 4 contacts, the convergence times are similar, but the success rates of IPOPT are much lower, averaging around 25%. This shows that our PG framework is more efficient and robust w.r.t the state-of-the-art.

These results also show that with our solver, the closer the initial guess is to the solution, the more likely a solution is to be found, and it gives us a quantitative estimate of the expected success rate and convergence times with respect to the initial distance. Interestingly, this observation cannot be made for resolution with the IPOPT framework.

Although all those computation times are long, and make it inconvenient to compute large numbers of postures, as is often done in contact planning, they remain of the same order of magnitude as the ones reported in [33] (3-4 seconds per posture), [1] (0.2 seconds per posture with only inverse kinematics). The most relevant comparison is with [32] that we presented earlier. Previous approaches are based on off-the-shelf solvers, while we use an open solver that we developed and that can be further customized to become faster and more robust with further developments.

In order to have a better understanding of its most common failure cases, we compiled the termination status of our solver that we report in Table I. This allows us to evaluate the weaknesses of our solver and will guide our future developments toward improving its robustness. Having most of the failure cases related to reaching the maximum number of iteration is not abnormal, but some cases are linked to a slow convergence due to poor Hessian approximation. The failure of the SOC phase should not happen so often, thus we will need to investigate that aspect of the solver first.

TABLE I
FREQUENCY OF FAILURE REASONS FOR UNSUCCESSFUL OPTIMIZATIONS

max iteration in main SQP	39%
max iteration in restoration	30%
SOC failure	22%
Restoration failure	7%
QP failure	2%

C. Postures with contacts on parametrized surfaces

We present some use cases for parameterizing the location of a contact point on a given surface by adding variables to the PG optimization problem, Section V-A.

1) Contact with an object defined by a parametric equation: We want the HRP-4 robot to carry a cube in its two hands. Instead of running combinatorial cases of hands/faces contacts, we approximate the cube with a superellipsoid parametrized on S^2. Its implicit equation is $S(x, y, z) = 0$, with

$$S(x,y,z) = \left(\left|\frac{x}{A}\right|^r + \left|\frac{y}{B}\right|^r \right)^{\frac{t}{r}} + \left|\frac{z}{C}\right|^t - 1 \qquad (33)$$

A point in S^2 is represented by a vector $v = (x, y, z)$ in $\mathbb{E} = \mathbb{R}^3$. To a given unit 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 $\frac{\nabla S(v)}{\|\nabla S(v)\|}$. Given this parametrization, we can write a contact constraint between given robot hand's frames and the point and normal on the parametrized superellipsoid's surface.

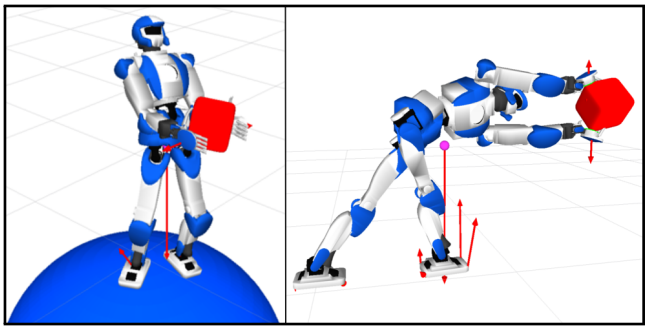


Fig. 10. HRP-4 carrying a 2kg cube. Left: feet on a sphere, objective function is to maintain the cube at a given position. Right: right foot free to move on the floor, objective is to put the cube as far as possible in a given direction.

The Fig. 10 presents some results for PG problems where the HRP-4 robot is required to grasp a cube modeled as a superellipsoid. On the left side, the feet are free to move on a sphere, on the right side, the left foot's position is fixed and the right foot is free to move on the ground. In both cases, contacts must occur between predefined points on the hands of the HRP-4 and points free to move on the surface of the cube. The cube is free to move (parametrized on $\mathbb{R}^3 \times SO(3)$) and all the contact forces must ensure grasp and posture equilibrium.

This approach extends to any other 3D shapes that can be represented by parametric or implicit equations.

2) Contact with an object defined by a mesh: We now show an example of a PG using the 3D modeling technique proposed in [8] to generate contacts with complex objects described by a 3D mesh. The Fig. 11 illustrates the HRP-4 robot climbing on a stack of cubes using both its hands and feet under constraints of static equilibrium, torque limits, etc.

The stack of cubes is modeled as a single object and the location of the 4 contact points are predefined on the robot but are free to be anywhere on the surface: the optimization algorithm determines the contact location as part of the solution. This is a very attractive feature of our approach as it allows to include discrete choices directly into our PG. This can be used to alleviate the work to be done by the user, be it a human or a planning algorithm: the user still has to specify the bodies in contact, but part of the combinatorics relative to the matching of a body with a particular surface or object is handled directly by the PG. Since the stack is a non-convex object, it cannot be used 'as is' in our collision detection algorithms. Thus, for collision constraints, we model it with a set of 21 slightly smaller cubes, see Fig. 11 (Right). The optimization takes around 115 iteration to converge to a solution.

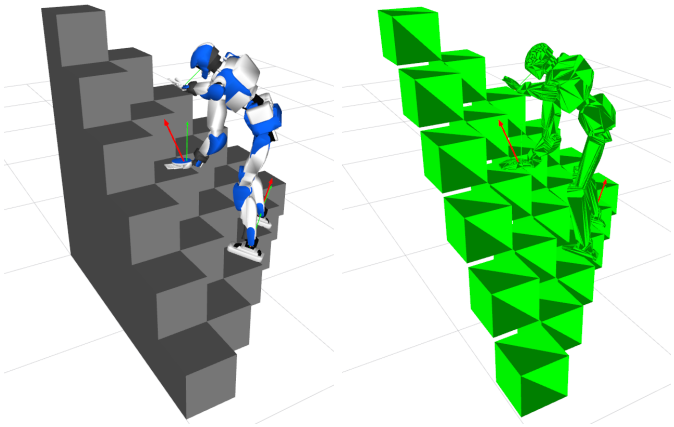


Fig. 11. HRP-4 climbing a stack of cubes modeled as a single object with a single surface (left). The convex shapes used for collision avoidance (right).

D. Postures with force constraints

Tasks where the robot needs to apply a given set of forces or torques on the environment are necessary for many applications. Therefore, we need to complete our PG with 'force-based' constraints to compute subsequent robot configurations.

1) Applying a desired force: To illustrate this feature, let's consider the example of a humanoid robot asked to apply a desired force on a given point of an airplane structure (printing or gluing a bracket, preparation for drilling, etc.). We denote ξ_d the desired value of the force in direction d, and ξ_c the actual force at the contact point. We can implement the following function $f_{\text{target}}(\xi_c)$ to minimize (hence a cost):

$$f_{\text{target}}(\xi_c) = (\xi_c \cdot d - \xi_d)^2 \qquad (34)$$

In addition to that cost function, the robot needs to keep its foot on the ground, its left hand is used to lean on a beam of the structure to allow a wider reach with the right hand. Those constraints must be fulfilled under the joint and torque limits of the robot, maintaining balance and avoiding auto-collisions. The result generated by our `PGSolver` is depicted in Fig. 12, where the robot applies a force of $100N$ in the desired direction on the airplane structure[6].

We assessed the validity of postures generated this way by solving a similar problem with an HRP-2 robot, and executing it on the real robot in a mockup of the airplane structure. The robot had to apply $100N$ and $200N$ on the environment. The posture and contact locations computed by our PG are not enough to generate the desired effort on the right hand of

[6]www.comanoid.eu

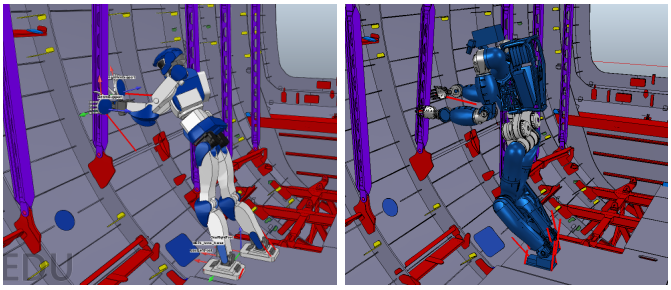


Fig. 12. HRP-4 / TORO applying a $100N$ desired force on a contact point with its right hand.

the robot. Indeed, for a given posture, an infinite number of contact forces can be generated by modifying the joint torques, so a specific control strategy needs to be implemented: we controlled the right gripper in admittance, while the other contacts were simply controlled in position. The robot was able to reach those forces ($100N$ and $200N$) without falling or slipping and while maintaining a posture very close to the one computed by our PG (less than 2% difference per joint w.r.t the joint range).

2) Applying a desired torque: We can also compute a posture where a desired torque is applied, for example in a torquing scenario pulling a lever by applying a minimum torque on its revolute joint. We consider a situation where a lever is attached to a wall of the environment by a revolute joint, the system $\{\text{wall} + \text{lever}\}$ is considered as a robot with a single degree of freedom. We write constraints allowing the hands of the robot to be in contact with the lever, rotating and translating freely along it (the translations being limited by the length of the lever). Additionally, those constraints allow the generation of contact forces in the plane orthogonal to the lever's axis passing through the contact point.

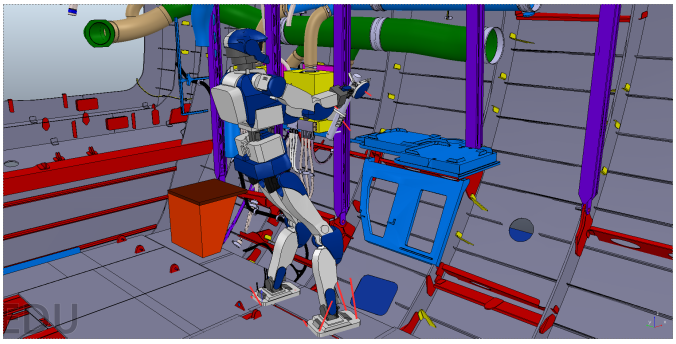


Fig. 13. HRP-4 applying a 20N.m desired torque on a lever.

To enforce the value of the torque to generate on the revolute joint of the lever, the torque limits of the lever robot can simply be overwritten. This formulation, augmented by the usual constraints on this system, allows computing postures where a desired torque is applied on the lever. The Fig. 13 depicts a PG solution to such a case; the location of the hands and the contact forces are autonomously chosen by the PG to generate a torque of 20N.m on the revolute joint at the lever basis.

Note that inequalities can also apply to contact forces or torques to limit or threshold them if necessary.

VII. Conclusion

Computing feasible postures using nonlinear optimization on manifolds proved to be elegant in terms of problems specifications while keeping good performances w.r.t Real-space formulation of the same problem. We devised our own SQP solver on manifolds to deal more efficiently with robotic problems. Our posture generator software includes many features not found in existing inverse kinematics (even generalized) ones. For example, we added the force dimensionality (specifying task desired force/torque thresholds, directions, limits, etc.) and static equilibrium constraints (grasp and whole-body positioning). We also show that we treat surfaces defined by closed-form equations or star-meshes to leave the decision of the contacts localization to the solver on manifolds. We also conducted assessment test on toy examples and illustrate the scalability and applicability to humanoid robots[7].

There are still things to improve, most of which are rather technical and require engineering efforts in software development (e.g. a user-friendly interactive API). On the research part, we will work on deeper specialization of the solver to the PG problems. It would be interesting to run more tests to study the influence of different solver options and find optimal strategies to set the solver's option to solve PG problems, maybe using learning techniques. Perhaps choosing automatically the update method and other options based on the structure of the problem at hand would allow increasing the general performance.

Appendix
$SO(3)$ example of non-convergence

Given x the angle-axis representation of a rotation (rotation of angle $\|x\|$ and axis x), let $R(x)$ be the corresponding rotation matrix. Given a rotation R_t, let us consider

$$\min_x . -\text{trace}(R_t^T R(x))$$

whose solution is attained for x such that $R(x) = R_t$. Taking the starting point $x^{(0)} = \begin{bmatrix}2\pi - \epsilon & 0 & 0\end{bmatrix}^T$ and $R_t = R(\begin{bmatrix}0 & 0 & \epsilon\end{bmatrix}^T)$ state-of-the-art solvers do not converge for ϵ small enough (*e.g* $\epsilon = 0.01$ with Matlab's `fminunc`). This is because $R(x^{(0)})$ is very close to R_t so that the gradient of the objective is small but $x^{(0)}$ is far from $\begin{bmatrix}0 & 0 & \epsilon\end{bmatrix}^T$.

References

[1] A. Escande, A. Kheddar, and S. Miossec, "Planning support contact-points for humanoid robots and experiments on HRP-2," in *IEEE/RSJ International Conference on Robots and Intelligent Systems*, Beijing, China, 9-15 October 2006, pp. 2974–2979.

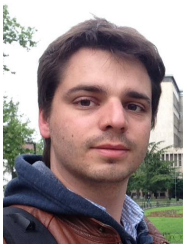
[2] K. Hauser, "Motion planning for legged and humanoid robots," Ph.D. dissertation, Stanford, 2008.

[3] A. Escande, A. Kheddar, and S. Miossec, "Planning contact points for humanoid robots," *Robotics and Autonomous Systems*, vol. 61, no. 5, pp. 428 – 442, 2013.

[4] J. Vaillant, A. Kheddar, H. Audren, F. Keith, S. Brossette, A. Escande, K. Bouyarmane, K. Kaneko, M. Morisawa, P. Gergondet, E. Yoshida, S. Kajita, and F. Kanehiro, "Multi-contact vertical ladder climbing with an HRP-2 humanoid," *Autonomous Robots*, vol. 40, no. 3, pp. 561–580, 2016.

[7]Software available at https://github.com/stanislas-brossette/manifolds and https://gite.lirmm.fr/posture_generator/problem-generator

[5] O. Stasse, D. Larlus, B. Lagarde, A. Escande, F. Saidi, A. Kheddar, K. Yokoi, and F. Jurie, "Towards autonomous object reconstruction for visual search by the humanoid robot HRP-2," in *IEEE-RAS Conference on Humanoids Robots*, Pittsburg, USA, 30 Nov. - 2 Dec. 2007.
[6] T. Foissotte, O. Stasse, P.-B. Wieber, A. Escande, and A. Kheddar, "Autonomous 3d object modeling by a humanoid using an optimization-driven next-best-view formulation," *International Journal on Humanoid Robotics*, vol. 7, no. 3, pp. 407–428, 2010.
[7] S. Brossette, A. Escande, J. Vaillant, F. Keith, T. Moulard, and A. Kheddar, "Integration of non-inclusive contacts in posture generation," in *IEEE/RSJ International Conference on Robots and Intelligent Systems*, Chicago, USA, Sep. 2014.
[8] A. Escande, S. Brossette, and A. Kheddar, "Parametrization of catmull-clark subdivision surfaces for posture generation," in *IEEE International Conference on Robotics and Automation*, Stockholm, Sweden, Feb. 2016.
[9] P. Baerlocher and R. Boulic, "Parametrization and range of motion of the ball-and-socket joint," in *Deform. Avatars*, 2001, pp. 180–190.
[10] D. G. Luenberger, "The gradient projection method along geodesics," *Management Science*, vol. 18, no. 11, pp. 620–631, 1972.
[11] A. Stuart and A. R. Humphries, *Dynamical systems and numerical analysis*. Cambridge University Press, 1998, vol. 2.
[12] D. Gabay, "Minimizing a differentiable function over a differential manifold," *Journal of Optimization Theory and Applications*, vol. 37, no. 2, pp. 177–219, 1982.
[13] P.-A. Absil, R. Mahony, and R. Sepulchre, *Optimization Algorithms on Matrix Manifolds*. Princeton University Press, 2008.
[14] J. Schulman, Y. Duan, J. Ho, a. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel, "Motion planning with sequential convex optimization and convex collision checking," *International Journal of Robotic Research*, vol. 33, no. 9, pp. 1251–1270, 2014.
[15] R. Fletcher and S. Leyffer, "Nonlinear programming without a penalty function," *Mathematical Programming*, vol. 91, pp. 239–269, 2000.
[16] S. Brossette, A. Escande, G. Duchemin, B. Chretien, and A. Kheddar, "Humanoid posture generation on non-Euclidean manifolds," in *IEEE-RAS International Conference on Humanoid Robots*, 2015, pp. 352–358.
[17] J. M. Lee, *Introduction to smooth manifolds*, 2nd ed. Springer New York, 2012.
[18] K. Bouyarmane and A. Kheddar, "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.*, Osaka, Japan, Nov. 29 - Dec. 1 2012, pp. 36–42.
[19] N. Boumal, B. Mishra, P.-A. Absil, and R. Sepulchre, "Manopt, a matlab toolbox for optimization on manifolds," *Journal of Machine Learning Research*, vol. 15, pp. 1455–1459, 2014.
[20] J. Nocedal and S. J. Wright, *Numerical Optimization*. Springer, 2006.
[21] P. E. Gill, W. Murray, and M. A. Saunders, "Snopt: An sqp algorithm for large-scale constrained optimization," *SIAM review*, vol. 47, no. 1, pp. 99–131, 2005.
[22] R. Fletcher, "The sequential quadratic programming method," in *Nonlinear optimization*. Springer, 2010, pp. 165–214.
[23] J. Nocedal and Y.-x. Yuan, "Analysis of a self-scaling quasi-newton method," *Mathematical Programming*, vol. 61, no. 1-3, pp. 19–37, 1993.
[24] R. Fletcher, *A new low rank quasi-Newton update scheme for nonlinear programming*. IFIP, 2006, no. August.
[25] M. J. Goldsmith, "Sequential quadratic programming methods based on indefinite hessian approximations," Ph.D. dissertation, Stanford University, March 1999.
[26] G. Golub and C. Van Loan, *Matrix computations*, 3rd ed. John Hopkins University Press, 1996.
[27] P. E. E. Gill, S. J. Hammarling, W. Murray, M. A. Saunders, and M. H. Wright, "User's guide for lssol (version 1.0): a fortran package for constrained linear least-squares and convex quadratic programming," Standford University, Standord, California 94305, Tech. Rep. 86-1, January 1986.
[28] R. Featherstone, *Rigid Body Dynamics Algorithms*. Springer, 2007.
[29] A. Escande, S. Miossec, M. Benallegue, and A. Kheddar, "A strictly convex hull for computing proximity distances with continuous gradients," *IEEE Transactions on Robotics*, vol. 30, no. 3, pp. 666–678, 2014.
[31] A. Wächter and L. T. Biegler, "On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2006.
[30] C. Lawrence, J. L. Zhou, and A. L. Tits, "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," 1997.
[32] K. Bouyarmane, "On autonomous humanoid robots: Contact planning for locomotion and manipulation," Ph.D. dissertation, Université Montpellier II-Sciences et Techniques du Languedoc, 2011.
[33] S. Brossette, J. Vaillant, F. Keith, A. Escande, and A. Kheddar, "Point-Cloud Multi-Contact Planning for Humanoids: Preliminary Results," in *IEEE Cybernetics and Intelligent Systems Robotics, Automation and Mechatronics*, Manila, Philippines, Nov. 2013.



Stanislas Brossette Stanislas Brossette received the BS degree of Mechanical Engineering from the University of Pierre and Marie Curie, Paris 6, in 2009, and the MS degree of Computational Mechanics from the École Normale Supérieure de Cachan, France in 2011. He then obtained the Ph.D. degree in Robotics in 2016 from the Université de Montpellier, France after spending four years with the Laboratory of Informatics, Robotics, and Microelectronics, Montpellier, France (LIRMM) and the CNRS-AIST Joint Robotics Laboratory (JRL), UMI3218/RL, Tsukuba, Japan. He then spent one year working as a post-doctoral fellow on the topic of bipedal walking at INRIA Grenoble, France. He currently works at Wandercraft in Paris as a Control-Command R&D Engineer where he develops walking algorithm for the Atalante autonomous exosqueleton. His research interests include multi-contact whole-body posture generation, numerical optimization and perception for robotics.

Adrien Escande Adrien Escande received the MS degree in 2005 from École des Mines de Paris, France, and the Ph.D. degree in 2008 in robotics from Université d'Évry Val-d'Essonne, France after spending three years in the CNRS-AIST Joint Robotics Laboratory (JRL), UMI3218/RL, Tsukuba, Japan. He then worked as a research scientist in CEA-LIST at Fontenay-aux-Roses, France, until the end of 2012 and is now back at JRL. His current research interests include whole-body planning and control for humanoid robots and mathematical optimization for robotics.

Abderrahmane Kheddar (M'04, SM'12) received the BS in Computer Science degree from the Institut National d'Informatique (ESI), Algiers, the MSc and PhD degree in robotics, both from the University of Pierre et Marie Curie, Paris. He is presently Directeur de Recherche at CNRS. His research interests include haptics, humanoids and thought-based control using brain machine interfaces. He is a founding member of the IEEE/RAS chapter on haptics, the co-chair and founding member of the IEEE/RAS Technical committee on model-based optimization. He is a member of the steering committee of the IEEE Brain Initiative, Editor of the IEEE Transactions on Robotics and within the editorial board of some other robotics journals. He is member of the National Academy of Technology of France, and knight in the National Order of the Merit.