Prehensile Manipulation Planning: Modelling, Algorithms and Implementation
Florent Lamiraux, Joseph Mirabel

To cite this version:
Florent Lamiraux, Joseph Mirabel. Prehensile Manipulation Planning: Modelling, Algorithms and Implementation. Rapport LAAS n° 20298. 2020. hal-02995125

HAL Id: hal-02995125
https://hal.laas.fr/hal-02995125
Submitted on 9 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Prehensile Manipulation Planning: Modelling, Algorithms and Implementation

Florent Lamiraux¹, Joseph Mirabel¹

Abstract
This paper presents a software platform tailored for prehensile manipulation planning named Humanoid Path Planner. The platform implements an original way of modelling manipulation planning through a constraint graph that represents the numerical constraints that define the manipulation problem. We propose an extension of RRT algorithm to manipulation planning that is able to solve a large variety of problems. We provide replicable experimental results via a docker image that readers may download to run the experimental results by themselves.

Keywords
robotics, manipulation planning, constrained path planning, path planning, software platform

1 Introduction
Robots in industrial manufacturing are today mostly programmed by hand. They repeat the same motion thousands of times with a high accuracy. Automating a task with some variability is however very challenging since it requires more programming effort to integrate sensors and motion planning in the process. A good example of this difficulty is the Amazon picking challenge Eppner et al. (2018). The work described in this paper is a small step to the simplification of industrial process automation in the presence of some variability, like the variation of the initial position of some object or the presence of unknown obstacles. The work only covers the motion planning – and more accurately the manipulation planning – issue. The integration in a whole process is still under development. We think that it is important not only to develop algorithms, but also to provide them within an open-source software platform in order to make the evaluation and then the integration of those algorithms easier.

This paper therefore describes a software platform called Humanoid Path Planner tailored for manipulation planning in robotics. It can handle many types of robots, from manipulator arms to legged humanoid robots. The main contributions are:

- an original and general modelling of prehensile manipulation based on non-linear constraints,
- an original solver for non-linear constraints that is able to handle implicit and explicit constraints,
- a manipulation planning algorithm that is able to tackle a great variety of manipulation planning problems,
- an open-source software suite that implements all the above following state of the art development tools and methods,
- a docker image of the above software with installation instructions provided with this paper.

This paper extends the work presented in Mirabel et al. (2016) and Mirabel and Lamiraux (2018) with some new material.

The paper is organized as follows. Section 2 presents some related work for constrained motion planning and manipulation planning. Section 3 introduces some preliminary notions like kinematic chains and Lie groups that are used to model the configuration space of each joint. Section 4 introduces non linear constraints and solvers that are at the core of manipulation problem definition. Section 5 defines the problem of prehensile manipulation in the general setting. Section 6 provides a general algorithm that solves manipulation planning problems. Finally, section 7 is devoted to the software platform implementing the notions

¹LAAS-CNRS, University of Toulouse, France

Corresponding author:
Florent Lamiraux, LAAS-CNRS, 7 avenue du Colonel Roche, Toulouse, France
Email: florent.lamiraux@laas.fr
introduced in the previous sections. Experimental results are provided for a large variety of problems.

Each section is implemented by one or several software packages. For some values that need to be computed, instead of providing formulas, we sometimes provide a link to the C++ or python implementation.

2 Related work

Motion planning has given rise to a lot of research work for the past decades. The problem consists in finding a collision-free path for a given system in an environment populated with obstacles. The field covers a large variety of different applications ranging from navigation for autonomous vehicles in partially known environments Likhachev et al. (2005) to path planning for deformable objects Lamiraux and Kavraki (2001), Roussel et al. (2020) and many other applications like coverage path planning Choset (2001), Galceran and Carreras (2013), or pursuit evasion planning Guibas et al. (1999).

Planning motions for high dimensional robots like humanoid robots or multi-arm systems has been proven to be of a very high complexity Schwartz and Sharir (1983), Canny (1983). Starting in the 1990’s random sampling methods have been proposed to solve the problem, trading the completeness property against efficiency in solving problems in high dimensional configuration spaces Kavraki et al. (1996), Hsu et al. (1999), Kuffner and LaValle (2000). The latter methods are said probabilistically complete since the probability to find a solution if one exists converges to 1 when the time of computation tends to infinity. Since then, asymptotically optimal random sampling algorithms have been proposed Karaman and Frazzoli (2011).

2.1 Path planning with non-linear constraints

Some systems are subject to non-linear constraints. These non-linear constraints define a sub-manifold of the configuration space the robot must stay on. For instance, legged robots that must keep contact with the ground and enforce quasi-static equilibrium, or multi-arm systems grasping a same object are subject to this type of constraints. As the volume of the constrained manifold is usually equal to zero, sampling random configurations satisfying the constraints is an event of zero probability. To sample configurations on the constrained manifold, Dalibard et al. (2009) and Berenson et al. (2011) project random configurations using a generalization of Newton-Raphson algorithm. Another way to sample configurations on a constrained manifold consists in expressing some configuration variables with respect to others Cortés et al. (2002), Mirabel and Lamiraux (2018) when this is possible. Jaillet and Porta (2013) propose another method based on non-linear projection. They cover the constrained manifold by growing an atlas composed of local charts. This approximation provides a probability distribution that is closer to the uniform distribution over the manifold than the projection of a uniform distribution over the configuration space. Kim et al. (2016) propose a variation of the latter paper. The main difference resides in the fact that the nodes built on the tangent space are not immediately projected on the manifold. Cefalo and Oriolo (2019) propose a general framework to plan task constrained motions in the presence of moving obstacle. Kingston et al. (2019) provide an in-depth review of the various approaches to motion planning with non-linear constraints.

2.2 Manipulation planning

Manipulation planning is a particular instance of path planning were some objects are moved by some robots. Although several instances of manipulation problem exist like manipulation by pushing Ben-Shahar and Rivlin (1998), or by throwing Woodruff and Lynch (2017), as well as multicontact planning Bretl (2006), Lengagne et al. (2013), Tonneau et al. (2018), in this paper, we are only concerned with prehensile manipulation. The configuration space of the whole system is constrained by non-linear constraints due to the fact that objects cannot move by themselves and should stay in a stable pose when not grasped by a robot. The accessible configuration space is thus a union of submanifolds as defined in the previous section. Each of these manifolds may moreover be a foliation where each leaf corresponds to a stable position of an object or to a grasp of an object by a gripper. The geometrical structure of the problem has been well understood for a long time Alami et al. (1989). Some specific instances of the problem have even been addressed recently Vendittelli et al. (2018).

The first attempt to solve manipulation planning problems using random sampling was proposed by Simon et al. (2004) where a reduction property reduces the complexity of the problem.

Papers about manipulation planning are commonly divided into several categories.

Navigation Among Movable Obstacles (NAMO) Wilfong (1988), Stilman and Kuffner (2008) consists in finding a path for a robot that needs to move objects in order to reach a goal configuration. The final position of the objects does not matter in this case.
Rearrangement planning Ota (2004), Garrett et al. (2017), Krontiris and Bekris (2015), Lertkultanon and Pham (2015) consists in finding a sequence of manipulation paths that move some objects from an initial pose to a final pose. The final configuration of the robot is not specified. A simplifying assumption is the existence of a monotone solution, that is a solution where each object is grasped at most once and is moved from its initial pose to its final pose Stilman and Kuffner (2008), Stilman et al. (2007), Srivastava et al. (2014), Nieuwenhuisen et al. (2008), Ota (2004). They mostly use two-level methods composed of a symbolic or task planner and of a motion planner Cambon et al. (2009), Kaelbling and Lozano-Pérez (2013), Toussaint et al. (2018).

Other contributions in manipulation planning explicitly address the problem of multi-arm manipulation Gharbi et al. (2009), Harada et al. (2014), Dobson and Bekris (2015), Xian et al. (2017).

Schmitt et al. (2019) propose an approach where two robots manipulate an object in a dynamic environment. The output of the algorithm is a sequence of controllers rather than a sequence of paths.

Our work shares many ideas with Hauser and Ng-Thow-Hing (2011) where the notion of constraint graph is present, although not as clearly expressed as in this paper. The main contribution of our work with respect to this latter paper is that the constraint graph is built automatically at the cost of a more restricted range of applications. We address indeed only prehensile manipulation.

### 2.3 Open-source software platforms

Open-source software platforms are an important tool to make possible fair comparison between algorithms. Several software platforms are available for motion planning and/or manipulation planning in the robotics community. The most popular one is undoubtedly OMPL. Sucan et al. (2012). OMPL integrates many randomized path planning algorithms and is widely used for teaching. Recently, Kingston et al. (2019) proposed an extension for systems subject to non-linear constraints.

OpenRave Diankov (2010) is a software platform that addresses motion and manipulation planning. It includes computation of forward kinematics.

One of the main differences between our solution and the previously cited ones resides in the way manipulation constraints are compiled into a graph. To our knowledge none of the previous solutions can handle a variety of problems as large as those exposed in Section 7.2.

### 3 Preliminaries

#### 3.1 Kinematic chains and Lie groups

A kinematic chain is commonly understood as a set of rigid-body links connected to each other by joints. Each joint has one degree of freedom either in rotation or in translation. A configuration of the kinematic chain is represented by a vector. Each component of the vector represents the angular or linear value of the corresponding joint.

This representation although well suited for fixed base manipulator arms is not well adapted for robots with a mobile base like wheeled mobile robots, aerial robots or legged robots, since the mobility of the base cannot be correctly represented by translation or rotation joints. Representing a freeflying object by three virtual translations followed by three virtual rotations called roll, pitch and yaw is indeed a poor workaround due to the presence of singularities. A good illustration of this is the gimball lock issue that arose during Apollo 13 flight. To avoid singularities, we propose the following definition.

#### 3.1.1 Kinematic chain

A kinematic chain is a tree of joints where each joint represents a mobility of a rigid-body link with respect to another link or with respect to the world reference frame. To each joint is associated a configuration space called the joint configuration space. The most common joints with their respective configuration spaces are

- linear translation with configuration space \( \mathbb{R} \),
- bounded rotation with configuration space \( \mathbb{R} \),
- unbounded rotation with configuration space \( SO(2) \),
- planar joint with configuration space \( SE(2) \),
- freeflyer joint with configuration space \( SE(3) \).

\( SO(n) \) and \( SE(n) \) respectively stand for special orthogonal group and special Euclidean group. They represent the group of rotations and the group of rigid-body transformations in \( \mathbb{R}^n \).

#### 3.1.2 Lie groups

The joint configuration spaces enumerated in the previous paragraph: \( \mathbb{R}^n \), \( SO(n) \), and \( SE(n) \) are all Lie groups. The group operation is \( + \) for \( \mathbb{R}^n \), and composition denoted as \( \cdot \) for \( SE(n) \). We refer to Murray et al. (1994) – Appendix A for a thorough definition of Lie groups. We expose here only properties that are useful for the following developments.

For any Lie group \( L \) with neutral element \( n \), the tangent space at the neutral element \( T_n L \) of the group naturally maps to the tangent space at any point of the group. This means that any velocity \( v \in T_n L \) uniquely defines
1. a velocity \( \mathbf{w} \in T_g \mathcal{L} \) at any point \( g \) of the group, and thus,
2. a vector field on the tangent space \( T \mathcal{L} \), and
3. by integration during unit time of the latter vector field, starting from the origin, a new point \( g_1 \in \mathcal{L} \).

Item 1 above is called the transport of velocity \( \mathbf{v} \) to \( g \). Item 3 is called the exponential map of \( \mathcal{L} \) and is denoted by \( \exp \).

Geometric interpretations

- \( \mathbb{R} \) (and by trivial generalization \( \mathbb{R}^n \)): the neutral element is 0. The tangent space at 0 is isomorphic to \( \mathbb{R} \) and
  \[ \forall \theta \in \mathbb{R}, \exp(\theta) = \theta. \]
- \( SE(3) \): an element \( g \) of \( SE(3) \) can be seen as the position of a moving frame in a fixed reference frame. A point \( x \in \mathbb{R}^3 \) is mapped to \( g(x) \). Note that \( x \) is also the coordinate vector of \( g(x) \) in the moving frame \( g \). If \( \mathbf{v}, \omega \) are a linear and an angular velocities at the origin, \( (\mathbf{v}, \omega) \) is transported to \( g \) as the same linear and angular velocities expressed in the moving frame. In other words, if
  \[ M = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} \]  \tag{1}
with \( R \in SO(3) \) and \( t \in \mathbb{R}^3 \) is the homogeneous matrix representing \( g \), and \( (\mathbf{v}, \omega) \) is a velocity in \( T_{g}SE(3) \), the velocity transported to \( g \) corresponds to linear and angular velocities \( R \mathbf{v} \) and \( R \omega \) of the moving frame. Integral curves of the vector field mentioned in item 2 above correspond to helicoidal motions of constant velocity expressed in the moving frame.

\( SE(2), SO(3), \) and \( SO(2) \) are (or are isomorphic to) subgroups of \( SE(3) \) and follow the same geometrical interpretation.

Vector representations Each Lie group element is represented by a vector. Rotations are represented by unit quaternions. Therefore elements of \( SE(3) \) are represented by a vector in \( \mathbb{R}^7 \) where the 3 first components represent the image of the origin (vector \( t \) in Equation 1), the 4 last components \((x, y, z, w)\) represent unit quaternion \( w + xi + yj + zk \).

Elements of \( SO(3) \) are likewise represented by a unit vector of dimension 4.

Elements of \( SE(2) \) are represented by a vector of dimension 4. The 2 first components represent the image of the origin. The 2 last components represent the cosine and sine of the rotation angle. Therefore the homogeneous matrix associated to \( \mathbf{q} = (q_1, q_2, q_3, q_4) \) is

\[ M = \begin{pmatrix} q_3 & -q_4 & q_1 \\
q_4 & q_3 & q_2 \\
0 & 0 & 1 \end{pmatrix}. \]

The following operators. For any \( \mathbf{q}_0, \mathbf{q}_1 \in \mathcal{L} \) and \( \dot{\mathbf{q}} \in T_{\mathbf{n}} \mathcal{L} \):

\[ \mathbf{q}_0 \circ \dot{\mathbf{q}} \triangleq \mathbf{q}_0, \exp(\dot{\mathbf{q}}) \in \mathcal{L}, \tag{2} \]

\[ \mathbf{q}_1 \circ \mathbf{q}_0 \triangleq \log(\mathbf{q}_0^{-1}, \mathbf{q}_1) \in T_{\mathbf{n}} \mathcal{L}. \tag{3} \]

*More precisely, following the vector field generated by \( \dot{\mathbf{q}} \in T_{\mathbf{n}} \mathcal{L} \) according to the Lie group structure.
3.1.3 Robot configuration space Given a kinematic chain with joints \((J_1, \ldots, J_{n_{\text{joints}}})\), ordered in such a way that each joint has an index bigger than its parent in the tree, the configuration space of the robot is the cartesian product of the joint configuration spaces.

\[
C \triangleq C_{J_1} \times \cdots \times C_{J_{n_{\text{joints}}}}.
\]

\(C\) naturally inherits the Lie group structure of the joint configuration spaces through the Cartesian product. We denote by \(nq_i\) and \(nv_i\) the sizes of the configuration and velocity vector representations of joint \(J_i\), as defined in Table 1. The configuration and velocity of the robot can thus be represented by vectors of size \(nq\) and \(nv\) such that

\[
nq = \sum_{i=1}^{n_{\text{joints}}} nq_i, \quad nv = \sum_{i=1}^{n_{\text{joints}}} nv_i
\]

We denote by \(i_q\), and \(iv\) the indices of joint \(i\) in the robot configuration and velocity vectors.

\[
i_q = \sum_{j=1}^{i-1} nq_j, \quad iv = \sum_{j=1}^{i-1} nv_j
\]

With these definitions and notation, the linear interpolation between two robot configurations \(q_0\) and \(q_1\) is naturally written:

\[
q(t) = q_0 \oplus t(q_1 - q_0)
\]

This formula generalizes the linear interpolation to robots with freeflying bases, getting rid of singularities of roll – pitch – yaw parameterization. Cartesian products of Lie groups are represented by Class \texttt{LiegroupSpace}.

Elements of these spaces are represented by classes

- \texttt{LiegroupElement}
- \texttt{LiegroupElementRef}, and
- \texttt{LiegroupElementConstRef}.

4 Non-linear constraints and solvers

Some tasks require the robot to enforce some non-linear constraints. Foot contact on the ground for a humanoid robot, center of mass projection on a horizontal plane, gaze constraint are a few examples.

4.1 Non-linear constraints

**Definition 1. Non-linear constraint.** A non linear constraint is defined by a differentiable mapping \(h\) from \(C\) to a vector space \(\mathbb{R}^m\) and is written

\[
h(q) = 0.
\]

If the robot is subject to several numerical constraints, \(h_1, \ldots, h_k\) with values in \(\mathbb{R}^{m_1} \cdots \mathbb{R}^{m_k}\), these constraints are equivalent to a single constraint \(h\) with values in \(\mathbb{R}^m\), where

\[
m = \sum_{i=1}^{k} m_i,
\]

such that

\[
h(q) \triangleq \begin{pmatrix} h_1(q) \\ \vdots \\ h_k(q) \end{pmatrix}.
\]

It may be useful to use a non zero right hand side for the same function \(h\). We define for that parameterized non-linear constraints.

**Definition 2. Parameterized non-linear constraint.** A parameterized non-linear constraint is defined by a differentiable mapping \(h\) from \(C\) to a vector space \(\mathbb{R}^m\) and by a vector \(h_0\) of \(\mathbb{R}^m\) and is written

\[
h(q) = h_0.
\]

Differentiable mappings are represented by abstract Class \texttt{DifferentiableFunction}.

**Jacobian** In this paper, we will make use of the term Jacobian in a generalized way. If \(h\) is a differentiable function from a Lie group \(L_1\) to a Lie group \(L_2\), and \(q_1\) an element of \(L_1\), we will denote by \(\frac{\partial h}{\partial q}(q_1)\) the operator that maps velocities in \(T_{q_1}L_1\) to the velocity in \(T_{h(q_1)}L_2\) transported by \(h\).

This operator is represented by a matrix with \(nv_2\) lines and \(nv_1\) columns, where \(nv_1\) and \(nv_2\) are the dimensions of the tangent spaces of respectively \(L_1\) and \(L_2\).

4.2 Newton based solver

It is sometimes useful to produce a configuration \(q\) that satisfies a constraint (or a set of constraints) of type (4) from a configuration \(q_0\) that does not. This action is called the projection of \(q_0\) onto the submanifold defined by the constraint and is performed by a Gauss-Newton solver (Nocedal and Wright 2006, Chapter 10) that iteratively linearizes the constraint as follows:

\[
h(q_i+1) \approx h(q_i) + \frac{\partial h}{\partial q}(q_i)(q_{i+1} - q_i) = 0
\]

Iterate \(q_{i+1}\) is computed as follows:

\[
q_{i+1} = q_i - \alpha_i \frac{\partial h}{\partial q}(q_i)h(q_i)
\]

\(^1\)If \(\dot{q} \in T_{q_1}L_1\) is a velocity along a time parameterized curve \(\gamma\), \(\frac{\partial h}{\partial q}(q_1)\dot{q}\) is the velocity at the same time along Curve \(h(\gamma)\).
where $\cdot^+$ denotes the Moore Penrose\(^1\) pseudo inverse, and $\alpha_i$ is a positive real number called the step size. Taking $\alpha_i = 1$ exactly solves the linear approximation, but it may not be the best choice in general.

The computation of $\alpha_i$ is performed by a line search algorithm. The algorithm stops when the norm of each $h_i(q_{i+1})$ is below a given error threshold. Class HierarchicalIterative implements the above Newton method. Several linesearch methods are implemented: Backtracking, ErrorNormBased, FixedSequence, and Constant. Note that to define a new constraint, the user needs to derive class DifferentiableFunction and to implement methods impl\_compute and impl\_jacobian.

### 4.3 Explicit constraints

In manipulation planning applications, where robots manipulate objects, once an object is grasped, the position of the object can be computed explicitly from the configuration of the robot. In this case, some configuration variables of the system depend on other:

$$\mathbf{q} = (\mathbf{q}_{\text{rob}}, \mathbf{q}_{\text{obj}}) \in \mathcal{C}, \quad \mathbf{q}_{\text{obj}} = g_{\text{grasp}}(\mathbf{q}_{\text{rob}}).$$

Although this constraint may fit definition (4) by defining

$$h(\mathbf{q}) \triangleq \mathbf{q}_{\text{obj}} \ominus g_{\text{grasp}}(\mathbf{q}_{\text{rob}}),$$

solving this constraint possibly with other constraints using an iterative scheme (5) is obviously sub-optimal.

More generally, let us denote by

- $I_{\text{nq}}$ the set of positive integers not greater than $nq = \dim \mathcal{C}$,
- $I$ a subset of $I_{\text{nq}}$,
- $\bar{I}$ the complement in $I_{\text{nq}}$ of $I$,
- $|I|$ the cardinal of $I$.

If $\mathbf{q} \in \mathcal{C}$ is a configuration, we denote by $\mathbf{q}_I \in \mathbb{R}^{|I|}$ the vector composed of the components of $\mathbf{q}$ of increasing indices in $I$.

**Example** if $\mathbf{q} = (q_1, q_2, q_3, q_4, q_5, q_6, q_7)$ and $I = \{1, 2, 6\}$, then $\mathbf{q}_I = (q_1, q_2, q_6)$. $\mathbf{q}_{\bar{I}} = (q_3, q_4, q_5, q_7)$.

Similarly, if

- $m$ and $n$ are two integers,
- $M$ and $N$ are two subsets of respectively $I_m$ and $I_n$,
- $J$ is a matrix with $m$ rows and $n$ columns,

we denote by

$$J_{M,N}$$

the matrix of size $|M| \times |N|$ obtained by extracting the rows of $J$ of indices in $M$ and the columns of $J$ with indices in $N$.

**Example** If $m = 3$, $n = 4$, $M = \{2, 3\}$ and $N = \{1, 2, 4\}$,

$$J = \begin{pmatrix}
J_{1,1} & J_{1,2} & J_{1,3} & J_{1,4} \\
J_{2,1} & J_{2,2} & J_{2,3} & J_{2,4} \\
J_{3,1} & J_{3,2} & J_{3,3} & J_{3,4} \\
J_{4,1} & J_{4,2} & J_{4,3} & J_{4,4}
\end{pmatrix}$$

then

$$J_{M \times N} = \begin{pmatrix}
J_{1,1} & J_{1,2} & J_{1,4} \\
J_{3,1} & J_{3,2} & J_{3,4}
\end{pmatrix}.$$

**Definition 3.** An explicit constraint $E = (in, out, f)$ is a mapping from $\mathcal{C}$ to $\mathcal{C}$, defined by the following elements:

- a subset of input indices $in \subset \{1, \ldots, nq\}$,
- a subset of output indices $out \subset \{1, \ldots, nq\}$,
- a smooth mapping $f$ from $\mathbb{R}^{|in|}$ to $\mathbb{R}^{|out|}$,

satisfying the following properties:

- $in \cap out = \emptyset$,
- for any $p \in \mathcal{C}$, $\mathbf{q} = E(p)$ is defined by $\mathbf{q}_{out} = p_{out}$, $\mathbf{q}_{out} = f(p_{in})$.

### 4.4 Solver by substitution

To optimize constraint resolution, we perform variable substitution when possible in order to reduce both the number of variables and the dimension of the resulting implicit constraint. We describe here the method that has been first published in Mirabel and Lamiraux (2018). Unlike in the former paper, the description we give in Algorithm 1 is closer to the real implementation. Some links to the source code are indeed provided in the algorithm description. Once several compatible explicit constraints have been inserted in the solver, they behave as a single one. For instance, if $\mathbf{q} = (q_1, q_2, q_3)$,

\[
\begin{align*}
q_1 &= f_1(q_2) \\
q_2 &= f_2(q_3)
\end{align*}
\]

becomes

$$\begin{bmatrix}
q_1 \\
q_2
\end{bmatrix} = \begin{bmatrix}
f_1(f_2(q_3)) \\
f_2(q_3)
\end{bmatrix},$$

and $f_2$ should be evaluated before $f_1$.

**Substitution** When an explicit constraint is not successfully added following Algorithm 1, it is handled as an implicit constraint. Therefore, after inserting implicit and

---

\(^1\)He has just been awarded the Nobel Prize.
Algorithm 1 Insertion of an explicit constraint in the solver. Line 1 is called once at initialization of the solver. *explicit* is a vector that stores the constraints that are successfully added to the solver. *nc* is the size of the latter. *args* is an array that stores for each configuration variable, the index in *explicit* of the constraint that computes this configuration variable. -1 if no constraint computes the index. Procedure ADD tests whether explicit constraint *E* is compatible with the previously inserted constraints. Line 6 checks whether any output variable of *E* is already computed by a previous explicit constraint. If so the procedure returns failure and *E* is not inserted. The Loop at line 9 recursively checks that all elements of *out* are computed by *E* and *E* is inserted in the vector of constraints. Function computeOrder at line 20 recursively computes the order in which the explicit constraints are evaluated, following the rule that the input of a constraint should be evaluated before the output.

1: function INITIALIZE
2:  explicit ← empty vector of explicit constraints
3:  nc ← 0
4:  args ← array of size *ng* filled with -1
5: function ADD(*E* = (*in*, *out*, *f*))
6:  if *args* contains an element ≥ 0 then
7:    return failure
8:  queue *idxArg* ← elements of *in*
9:  while *idxArg* not empty do
10:     *iArg* ← *idxArg* first element
11:     remove *idxArg* first element
12:     if *iArg* ∈ *out* then
13:       return failure
14:     if *args*[*iArg*] == -1 then
15:       continue
16:     else
17:       push *explicit*[args[*iArg*]].*in* elements into *idxArg*
18:     fill *args* with *nc*
19:     explicit.add(*E*); nc ← nc + 1
20: computeOrder()
21: return success

The solver by substitution applies iteration (5) to $\tilde{h}$, instead of *h*. We need therefore to compute the Jacobian of $\tilde{h}$:

$$\frac{\partial \tilde{h}}{\partial \mathbf{q}_{in}} = \frac{\partial h}{\partial \mathbf{q}_{in}} + \frac{\partial h}{\partial \mathbf{q}_{out}} \frac{\partial f}{\partial \mathbf{q}_{in}}.$$ 

As the Jacobian of *h* is provided with the implicit constraint, we need to compute $\frac{\partial J}{\partial \mathbf{q}_{in}}$. Let us recall that *f* may be the combination of several compatible explicit constraints. Let us denote by *E* the mapping from $\mathcal{C}$ to $\mathcal{C}$ associated to *f* by Definition 3. Let *J* denote the *nv* × *nv* Jacobian matrix of *E*. Then *J* is defined by blocks as follows:

$$J_{in \times in} = I_{[in]} \quad J_{in \times out} = 0$$
$$J_{out \times in} = \frac{\partial f}{\partial \mathbf{q}_{in}} \quad J_{out \times out} = 0$$

(11)

If *E* is the composition of several explicit constraints $E_i = (i_{in}, i_{out}, f_i)$ of Jacobian $J_i$, $i \in I_{nc}$, for an integer *nc*, then

$$J = \prod_{i=nc}^1 J_i$$

(12)

with $J_i$ obtained by expression (11) after replacing *in*, *out*, and *f* by $i_{in}$, $i_{out}$, and $f_i$.

$\frac{\partial f}{\partial \mathbf{q}_{in}}$ is then obtained by extracting from *J* block $out \times in$.

Let us now detail the iterative computation of (12). Let *J* be the product of $J_i$ for *f* from *nc* to *i* + 1. Note that if $J_i$ and *J* are square matrices of size *nv*, of the form (11), $J_i$.*J* can be computed by block as follows:

$$(J_i).J)_{in \times I_{nv}} = J_{in \times I_{nv}}$$
$$(J_i).J)_{out \times I_{nv}} = \frac{\partial f_i}{\partial \mathbf{q}_{in}}.J_{in \times I_{nv}}$$

and as columns *out* of *J* are equal to 0, left multiplying *J* by $J_i$ consists in modifying only the following block of *J*:

$$(J_i).J)_{out \times in} = \frac{\partial f_i}{\partial \mathbf{q}_{in}}.J_{in \times in}$$

Other coefficients of $J_i$.*J* are equal to the corresponding coefficients of *J*. An implementation of the above Jacobian product can be found here.

The solver by substitution described in this section is implemented by Class SolverBySubstitution, that store an instance of ExplicitConstraintSet.

**Important remark** As mentioned in Table 1, the configuration and velocity vectors may have different sizes. As a consequence, index sets *in* and *out* in Definition 3 correspond to configuration vector indices, while in Expression (11), they correspond to velocity vector.
indices. To keep notation simple, we use the same notation for different sets.

4.5 Constrained path

Now that we are able to project configurations onto submanifolds defined by numerical constraints – up to some numerical threshold, we need now to define paths on such submanifolds. The usual way of doing so is to discretize the path and to project each sample configuration. The shortcoming of this method is that it requires to choose a discretization step at path construction and to lose the continuous information of the path.

Instead, we propose an alternative architecture where paths store the constraints they are subject to and apply the constraints at path evaluation. Let \( P \in C^1([0,T],\mathcal{C}) \) be a path without constraint defined on an interval \([0,T]\), and \( \text{proj} \) a projector onto a submanifold defined by numerical constraints (i.e. an instance of \( \text{SolverBySubstitution} \)).

Then the corresponding constrained path \( \tilde{P} \) is defined on the same interval by

\[
\forall t \in [0,T], \quad \tilde{P}(t) = \text{proj}(P(t))
\]

Paths are implemented by Class \( \text{Path} \). Several implementations of unconstrained paths are provided: \( \text{StraightPath} \) for linear interpolation generalized to Lie groups, \( \text{ReedsSheppPath}, \text{DubinsPath} \) for nonholonomic mobile robots.

4.5.1 Continuity of projection along a path Projecting configurations at path evaluation has the advantage of not losing information. In return, the projection of a continuous path may be discontinuous. Before inserting a projected path in a roadmap for instance, it is therefore necessary to detect possible discontinuities. Hauser (2013) proposes a solution to this problem. Mirabel and Lamiraux (2016) describe two algorithms to check whether a projected path is continuous. These algorithms are implemented by classes \( \text{pathProjector::Dichotomy} \) and \( \text{pathProjector::Progressive} \).

5 Manipulation Problem

The previous sections have presented how we model kinematic chains, configurations and velocities for a given robotic system, how configurations and paths can be projected onto a submanifold of the configuration space defined by numerical constraints. In this section, we will use these notions to represent a robotic manipulation problem.

Definition 4. Prehensile manipulation problem

A prehensile manipulation problem is defined by

- one or several robots,
- one or several objects,
- a set of possible grasps,
- environment contact surfaces,
- object contact surfaces,
- an initial configuration,
- a final configuration.

Admissible configurations of the system are configurations that satisfy the following property:

- each object is either grasped by a robot, or lies in a stable contact pose.

Admissible motions of the system are motions that satisfy the following property:

- configurations along the motion are admissible, and
- the pose of objects in stable contact is constant,
- the relative pose of objects grasped by a gripper with respect to the gripper is constant.

The solution of a prehensile manipulation problem is an admissible motion that links the initial and goal configurations.

We will now provide precise definitions for grippers, grasps and stable contact poses.

5.1 Grasp

Configuration space The configuration space of a manipulation problem is the Cartesian product of the configuration spaces of the robots and of the objects.

\[
\mathcal{C} = C_{r_1} \times C_{r_{nr}} \times SE(3)^{no}
\]

where \( nr \) is the number of robots, \( no \) is the number of objects, \( C_{r_i}, i \in \{1,...,nr\} \) is the configuration space of robot \( r_i \).

Definition 5. Gripper. A gripper \( g \) is defined as a frame attached to the link of a robot. \( g(q) \). \( q \in \mathcal{C} \) denotes the pose of the frame when the system is in configuration \( q \).

Definition 6. Handle. A handle is composed of

- a frame \( h \) attached to the root joint of an object,
• a list flags = (x, y, z, rx, ry, rz) of 6 Boolean values.

h(q), q ∈ C denotes the pose of the frame when the system is in configuration q.

Definition 7. Grasp. A grasp is a numerical constraint h over C, defined by

• a gripper g,
• a handle h.

Let h be the smooth mapping from C to R^6 that maps to any configuration q of the system, the expression

\[ \bar{h}(q) = \log_{SE(3)}(g^{-1}(q)h(q)) \]  

(13)

h(q) is obtained by extracting from h the components of which are true in the handle flag.

Definition 8. Grasp complement. Given a grasp constraint defined by gripper g, handle h and some flag vector, the grasp complement is a parameterized non-linear constraint defined by

\[ h_{\text{comp}}(q) = h_0 \]

where \( h_{\text{comp}} \) is composed of the components of \( \bar{h} \) that are not in \( h \) and \( h_0 \) is a vector with the same size as \( h_{\text{comp}} \) output.

Geometric interpretation and examples The 3 first components of \( \bar{h}(q) \) in equation (13) correspond to the position of the center of \( h(q) \) in the frame of \( g(q) \). The 3 last components of \( \bar{h}(q) \) is a vector representing the relative orientation of \( h(q) \) with respect to \( g(q) \). The direction of the vector represents the axis of rotation, the norm of the vector represents the angle of rotation.

• If flags = (true, true, true, true, true, true) the grasp is satisfied iff \( g(q) \) and \( h(q) \) coincide:

\[ h = \bar{h}, \]

\( h_{\text{comp}} \) is an empty constraint;

• if flags = (true, true, true, true, true, false) the grasp is satisfied iff the centers and z axes of \( g(q) \) and \( h(q) \) coincide (free rotation around z). This is useful for cylindrical objects:

\[ h = (\bar{h}_1, \bar{h}_2, \bar{h}_3, \bar{h}_4, \bar{h}_5), \]

\[ h_{\text{comp}} = (\bar{h}_6); \]

• if flags = (true, true, true, false, false, false) the grasp is satisfied iff the centers of \( g(q) \) and \( h(q) \) coincide (free rotation). This is useful for spherical objects.

\[ h = (\bar{h}_1, \bar{h}_2, \bar{h}_3), \]

\[ h_{\text{comp}} = (\bar{h}_4, \bar{h}_5, \bar{h}_6). \]

If \( q_0 \) is a configuration satisfying the grasp constraint: \( h(q_0) = 0 \), then the submanifold defined by

\[ \{ q \in C, \ h(q) = 0 \ h_{\text{comp}}(q) = h_{\text{comp}}(q_0) \} \]

contains all the configurations that are reachable from \( q_0 \) while keeping the grasp. Note that this representation of relative pose constraints has been used in the Stack of Task software, although it is not described in the corresponding paper Mansard et al. (2009). It is different from Task Space Regions Berenson et al. (2011) where open domains of \( SE(3) \) are defined.

5.2 Stable Contact pose

When an object is not grasped, it should lie in a stable pose.

There are two simple methods to enforce that:

1. defining virtual grippers in the environment and virtual handles on the object, implicitly defines a discrete set of poses,
2. defining a virtual gripper on an horizontal plane and a virtual handle on the object, and using a grasp with flags (false, false, true, true, true, false) constrains the object to move on a infinite horizontal plane.

We propose here a third method that enables users to define contact surfaces in a more flexible way. In this aim, we denote by:

• \( (o_i)_{i \in I} \) a set of convex polygons attached to an object,
• \( (f_j)_{j \in J} \) a set of convex polygons attached to the environment or to a mobile part of a robot that can receive objects (mobile robot for instance),

\[ \bar{d}_j = P(C_m, f_j) \]
• respectively \( C_{o_i}, n_{o_i}, \) the barycenter of \( o_i \) and the normal to the plane containing \( o_i, \)
• \( C_{f_j}, n_{f_j} \) the barycenter of \( f_j \) and the normal to the plane containing \( f_j, \)
• \( P(C_{o_i}, f_j), \) the orthogonal projection of \( C_{o_i} \) onto the plane containing \( f_j. \)

Then we define the distance between polygons \( o_i \) and \( f_j \) as the distance of \( C_{o_i} \) to the cylindrical volume of generatrix \( n_{f_j} \) and of directrix \( f_j: \)

\[
d(f_j, o_i) = \sqrt{d_1^2 + d_2^2} \tag{14}\]

where

\[
d_i = \begin{cases} 
  d(f_j, P(C_{o_i}, f_j)) & \text{if } P(C_{o_i}, f_j) \text{ inside } f_j \\
  0 & \text{otherwise} 
\end{cases}
\]

\[
d_\perp = n_{f_j} \cdot C_{f_j} - C_{o_i}. \]

Figure 1 illustrates this definition.

We denote by \( o_i(q) \) and \( f_j(q) \) the poses in configuration \( q \) of frames with respective centers \( C_{o_i} \) and \( C_{f_j} \) and with \( x \)-axis normal to each polygon. In an analogous way as in Definition 7, we define

\[
h(q) = \log_{\mathbb{R}^3 \times SO(3)} (f_j(q)^{-1} o_i(q)) \tag{15}\]

The contact constraint is defined by the following piecewise differentiable function:

\[
h(q) = \begin{cases} 
  (\bar{h}_1, 0, 0, \bar{h}_4, \bar{h}_5) & \text{if } P(C_{o_i}, f_j) \text{ inside } f_j \\
  (\bar{h}_1, \bar{h}_2, \bar{h}_3, \bar{h}_4, \bar{h}_5) & \text{if } P(C_{o_i}, f_j) \text{ outside } f_j 
\end{cases} \tag{16}\]

It is straightforward that the above function vanishes if and only if two convex polygons \( o_i \) and \( f_j \) are in contact and \( C_{o_i} \) is inside \( f_j. \)

As for grasps, we need to define a parameterized complement constraint for the contact constraint in order to specify the submanifold of configurations that are reachable from one configuration while keeping the object in a constant stable pose. The naive way consists in defining

\[
h_{\text{comp}}(q) = \begin{cases} 
  (\bar{h}_2, \bar{h}_3, \bar{h}_6) & \text{if } P(C_{o_i}, f_j) \text{ inside } f_j \\
  (0, 0, \bar{h}_6) & \text{if } P(C_{o_i}, f_j) \text{ outside } f_j 
\end{cases} \tag{17}\]

The submanifold defined by

\[
\{q \in \mathcal{C}, \ h(q) = 0, h_{\text{comp}}(q) = h_{\text{comp}}(q_0)\} \tag{18}\]

contains one object pose for each pair of polygons \( (o_i, f_j), \) and there are \(|\mathcal{I}| \cdot |\mathcal{J}| \) possible combinations. This constraint is therefore not suitable to enforce object immobility along a path since the object may jump from one pose to another one.

To disambiguate the various combination of convex polygons that can be in contact, we define

\[
h_{\text{comp}}(q) = \begin{cases} 
  (\bar{h}_2 + 2jM, \bar{h}_3 + 2iM, \bar{h}_6) & \text{if } P(C_{o_i}, f_j) \text{ inside } f_j \\
  (2jM, 2iM, \bar{h}_6) & \text{if } P(C_{o_i}, f_j) \text{ outside } f_j 
\end{cases} \tag{19}\]

where

• \( i \) and \( j \) are the indices that minimize distance (14),
• \( M \) is a positive real number such that for any \( \kappa \in J, \)
all vertices of \( f_k \) are included in the disk of center \( C_{f_k} \) and of radius \( M. \)

With this definition, the submanifold defined by (17), (16), (18) contains configurations where the object is in a unique stable pose. The polygon indices \( i \) and \( j \), as well as their relative position can indeed be recovered from (18):

\[
\begin{align*}
  i & = \left\lfloor \frac{\bar{h}_2}{2M} + \frac{1}{2} \right\rfloor \\
  j & = \left\lfloor \frac{\bar{h}_3}{2M} + \frac{1}{2} \right\rfloor \\
  h_1 & = \bar{h}_4 = \bar{h}_5 = 0 \\
  h_2 & = h_{\text{comp}1} - 2jM \\
  h_3 & = h_{\text{comp}2} - 2iM \\
  h_6 & = h_{\text{comp}3} 
\end{align*} \tag{20}\]

and from (15),

\[
f_j(q)^{-1} o_i(q) = \exp_{\mathbb{R}^3 \times SO(3)} \left( \bar{h} \right). \tag{21}\]

Uniqueness comes from the fact that when two convex polygons are in contact, necessarily, \( |\bar{h}_2| \leq M, |\bar{h}_3| \leq M. \)

5.3 Merging constraint and complement into an explicit constraint

Note that when a grasp constraint and its complement are combined, they constitute an explicit constraint since the pose of the object grasped uniquely depends on the configuration of the robot that grasps the object.
Similarly, when a placement constraint and its complement are combined, they constitute an explicit constraint since the pose of the object placed uniquely depends on the pose of the contact surface on which the object is placed. This latter pose

- either depends on the configuration of the robot the contact surface belongs to,
- or is constant if the contact surface belongs to the environment.

In any case, the explicit expression of the object pose depends on the right hand side of the complement constraint that is constant along transition paths.

During the construction of the constraint graph (described in Section 5.4), grasp and placement constraints, their complements and the associated explicit constraints are created together and registered using method registerConstraint of Class ConstraintGraph.

5.4 Constraint Graph

According to Definition 4, the set of admissible configurations of a manipulation problem is the union of sub-manifolds of the configuration space of the system. Each sub-manifold is defined by grasp and/or stable contact constraints. We call each sub-manifold a state of the problem.

A state can be defined by a subset of active grasps, any object not grasped being in a stable contact pose. Let \( n_g \), \( n_h \), and \( n_o \) respectively denote the number of grippers, handles, and objects.

We denote by

- \( \text{grasp}_{ij} \) \( i \in \{1, \cdots, n_g\} \), \( j \in \{1, \cdots, n_h\} \) the grasp constraint of handle \( j \) by gripper \( i \),
- \( \text{grasp}_{ij}/\text{comp} \) \( i \in \{1, \cdots, n_g\} \), \( j \in \{1, \cdots, n_h\} \) the complement constraint of the latter,
- \( \text{place}_i \) \( i \in \{1, \cdots, n_o\} \) the placement constraint of object \( i \),
- \( \text{place}_i/\text{comp} \) \( i \in \{1, \cdots, n_o\} \) the complement constraint of the latter.

A state \( \mathcal{S} \) is denoted by a vector of size \( n_g \):

\[
\mathcal{S} = (h_1, \cdots, h_{n_h})
\]

where \( h_i \in \{0, 1, \cdots, n_h\} \) denotes the id of the handle grasped by gripper \( i \); \( h_i = 0 \) means that gripper \( i \) does not grasp any handle.

**Number of states:** note that for \( i \in \{1, \cdots, n_h\} \) the number of occurrences of \( i \) in \( \mathcal{S} \) is at most 1: a handle cannot be grasped by several grippers. Note also that the number of occurrences of \( \emptyset \) is not limited: several grippers may hold nothing. Let \( m \) be a non-negative integer not greater than \( n_g \) nor \( n_h \) and let us count the number of states with \( m \) handles grasped. The number of subset of \( m \) handles among \( n_h \) is equal to \( \frac{n_h!}{(n_h-m)!m!} \). And the number of ways of dispatching them among the \( n_g \) grippers is equal to \( \frac{n_g!}{(n_g-m)!} \). Thus, the total number of states is equal to

\[
\min(n_g, n_h) \sum_{m=0}^{\min(n_g, n_h)} \frac{n_h!}{(n_h-m)!m!} \frac{n_g!}{(n_g-m)!}
\]

**Definition 9. Neighboring states**

Two states \( \mathcal{S}_1 = (h_{11}, \cdots, h_{n_h1}) \) and \( \mathcal{S}_2 = (h_{12}, \cdots, h_{n_h2}) \) are neighboring if they differ by only one grasp and the grasp is empty in one of the states:

\[
\exists i \in \{1, \cdots, n_g\}, h_{1i} \neq h_{2i} \text{ and } (h_{1i} = \emptyset \text{ or } h_{2i} = \emptyset), \text{ and } \forall j \in \{1, \cdots, n_h\}, j \neq i, h_{1j} = h_{2j}.
\]

**Definition 10. Constraint graph**

The constraint graph related to a manipulation problem as defined in Definition 4 is a graph

- the nodes of which are states defined by subsets of grasps (19),
- two edges (back and forth) connect two states if they are neighboring,
- one edge connect each state to itself.

**Edges are also called transitions. Nodes contain**

- the grasp constraints that are active in the corresponding state,
- a placement constraint for each object that is not grasped by any handle.

**Transitions contain**

- the constraints of the node they connect with the least active grasps,
- the parameterized complement constraint of each of the latter.

5.5 Example

To illustrate the notions exposed in the previous sections, let us consider an example of two UR3 robots manipulating a cylinder illustrated in Figure 2. The robot is equipped with one gripper attached to the end-effector. The cylinder is equipped with two handles and with two square contact surfaces corresponding to the top and bottom faces of the cylinder. \( n_g = 2, n_h = 2, n_o = 1 \). The flag of the handles are

\[(true, true, true, false, true, true).\]
Algorithm 2 Recursive Construction of the constraint graph.
The construction starts by the state with no grasp. Call to `RECURSE` function loops over the available grippers and handles and creates states with one more grasp, and a transition to these new states. In each state, a placement constraint is added for each object of which no handle is grasped. Variables $G$ and $H$ contain the indices of the free grippers and handles. Variable $Gr$ stores the current set of grasps following Expression (19). Lines 29 to 33 compute which objects are not grasped. Lines 44 to 47 insert placement constraints in the state for those objects. Line 24 recurses only if the last node reached is new.

1: global variables
2: $n_o$ \quad \triangleright\ number of objects
3: $n_g$ \quad \triangleright\ number of grippers
4: $n_h$ \quad \triangleright\ number of handles
5: 
6: function `BUILDCONSTRAINTGRAPH`
7: $G \leftarrow [0, \ldots, n_g - 1]$ \quad \triangleright\ list of gripper indices
8: $H \leftarrow [0, \ldots, n_h - 1]$ \quad \triangleright\ list of handle indices
9: $Gr \leftarrow [\emptyset, \ldots, \emptyset]$ \quad \triangleright\ list of size $n_g$
10: RECURSE($G, H, Gr$)
11: function `RECURSE($G, H, Gr$)`
12: CREATESTATE($Gr$)
13: if $G = \emptyset$ or $H = \emptyset$ then
14: return
15: for $g$ in $G$ do
16: $G' \leftarrow G \setminus \{g\}$
17: for $h$ in $H$ do
18: $H' \leftarrow H \setminus \{h\}$
19: $Gr' \leftarrow Gr$
20: $Gr'[g] \leftarrow h$
21: isNewState $\leftarrow$ not EXISTSSTATE($Gr'$)
22: CREATESTATE($Gr'$)
23: CREATETRANSITION($Gr, Gr'$)
24: if isNewState then RECURSE($G', H', Gr'$)
25: function `CREATESTATE($Gr$)`
26: if EXISTSSTATE($Gr$) then
27: return
28: $S$ $\leftarrow$ new state
29: $S.Pl \leftarrow [\text{true}, \ldots, \text{true}]$ \quad \triangleright\ list of size $n_o$
30: for $o$ in $[0, \ldots, n_o - 1]$ do
31: $h \leftarrow Gr[g]
32: if $S.Pl[\text{OBJECTINDEX}(h)] \leftarrow \text{false}$
33: $S$ ADDGRASPCONSTRANT($g, h$)
34: if $S.Pl[o]$ then
35: $S$ ADDPLACECONSTRANT($o$)
36: function `CREATETRANSITION($Gr_1, Gr_2$)`
37: $T$ $\leftarrow$ new transition($Gr_1, Gr_2$)
38: $S_1 \leftarrow$ STATE($Gr_1$) \quad \triangleright\ Recover state for this set of grasps
39: for $g$ in $[0, \ldots, n_g - 1]$ do
40: $h \leftarrow Gr_1[g]
41: T$ ADDGRASPCONSTRANT($g, h$)
42: T$ ADDGRASPCONSTRANTCOMP($g, h$)
43: for $o$ in $[0, \ldots, n_o]$ do
44: if $S_1.Pl[o]$ then
45: $T$ ADDPLACECONSTRANT($o$)
46: $T$ ADDPLACECONSTRANTCOMP($o$)
47: $T_1 \leftarrow$ new transition($Gr_2, Gr_1$)
48: $T_1$ SETCONSTRANTS($T$ $\cdot$ constraints($))$

Table 2. State constraints.

<table>
<thead>
<tr>
<th>state</th>
<th>active constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\emptyset, \emptyset)$</td>
<td>place$_1$</td>
</tr>
<tr>
<td>$(j, \emptyset), j \in {1, 2}$</td>
<td>grasp$_{1j}$</td>
</tr>
<tr>
<td>$(\emptyset, j), j \in {1, 2}$</td>
<td>grasp$_{2j}$</td>
</tr>
<tr>
<td>$(i, j), i, j \in {1, 2}$</td>
<td>grasp$<em>{1i,j}, \text{grasp}</em>{2i,j}$</td>
</tr>
</tbody>
</table>

Therefore grasp constraints are of dimension 5 and keep the rotation of the gripper around the cylinder axis free. Table 2 indicates which constraints are active for each state. Table 3 indicates which constraints are active for each transition.

Figure 2. Top: two UR3 robots with one gripper each (X=red, Y=green, Z=blue) manipulating a cylinder with two handles. The environment contains one rectangular contact surface (in red). The cylinder has two rectangular contact surfaces (in green).

Bottom: the corresponding constraint graph. Names of states follow Expression (19); for instance, $(\emptyset, 1)$ means that gripper of robot 2 grasps handle 1 of the cylinder. In this state, there is no placement constraint.
<table>
<thead>
<tr>
<th>transition</th>
<th>belongs to</th>
<th>additional constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>((0, 0)) → ((0, 0))</td>
<td>((0, 0))</td>
<td>place(_1/)comp</td>
</tr>
<tr>
<td>((0, 0)) → ((0, i))</td>
<td>((0, 0))</td>
<td>place(_1/)comp</td>
</tr>
<tr>
<td>((i, 0)) → ((i, j))</td>
<td>((i, 0))</td>
<td>grasp(_{1/)comp}</td>
</tr>
<tr>
<td>((0, j)) → ((0, j))</td>
<td>((0, j))</td>
<td>grasp(_{2/)comp}</td>
</tr>
</tbody>
</table>

Table 3. Transition constraints: \(i, j\) are either 1 or 2. Column "belongs to" means that paths along the transition belong to the state, i.e. the transition contains the state constraints.

5.6 Automatic construction

Given a set of grippers, handles and objects, the constraint graph can be constructed automatically. Here is an implementation in python. Algorithm 2 describes this implementation. Functions

- **graspConstraint**,
- **graspConstraintComp**, build grasp constraint and complement constraint as defined in Section 5.1,
- **placeConstraint**,
- **placeConstraintComp** build placement constraints and complement as defined in Section 5.2,
- **existState\((Gr)\) returns true if a state has already been created for the set of grasps given as input,
- **state\((Gr)\) returns the state created with the set of grasps given as input,
- **objectIndex\((h)\) returns the index of the object handle \(h\) belongs to.

6 Manipulation planning

In this section, we show how the constraint graph defined in the previous section is used to plan collision-free manipulation paths. Although we are working on an extension of Algorithm RMR* Schmitt et al. (2017) to several grippers, objects and handles, the only manipulation planning algorithm available up to now in HPP is an extension of RRT algorithm described in the next section.

6.1 Manipulation-RRT

Manipulation Randomly exploring Random Tree is an extension of RRT algorithm LaValle and Kuffner (2001) that grows trees in the free configuration space, exploring the different states of the manipulation problem. Algorithm 3 describes the algorithm implemented in C++ here.

After initializing the roadmap with the initial and goal configurations, the algorithm iteratively calls method ONESTEP until a solution path is found or the maximum number of iterations is reached. This latter method shoots a random configuration (line 6) and for each connected component of the roadmap and each state of the constraint graph, extends the nearest node in the direction of the random configuration (lines 7–10). For each successful extension, the end of the extension path is stored for later connection (line 11). After the extension step, the algorithm tries to connect new nodes to other connected components using two strategies:

1. function **TRYCONNECTNEWNODES** calls method **CONNECT** between all pairs of new nodes,
2. function **TRYCONNECTTORDOMAP** tries to connect each new node to the nearest nodes in other connected components of the roadmap also using function **CONNECT**.

Function **CONNECT** attempts to connect two configurations in two states. First, it checks whether there exists a transition between the states. If so, it checks that the right hand side of the transition parameterized constraints is the same for both configurations (up to the error threshold). Then it returns the linear interpolation between the configurations, projected onto the submanifold defined by the transition constraints. If the path is in collision, only the collision-free part at the beginning of the path is returned.

Function **EXTEND** attempts to generate a path from a configuration in a state to another state following a random transition. Similarly as for function **CONNECT**, the path is projected onto the sub-manifold defined by the transition constraints. The end configuration is obtained by applying to the random configuration the constraints of the transition and of the goal state.

6.2 Examples

In this section, we illustrate the algorithm described in the previous section with two examples. Figure 3 illustrates function **EXTEND** defined in the previous section on the example of Figure 2. The system considered is composed of two robots and a cylinder with two handles. The top picture displays \(q_{rand}\). The middle picture displays \(q_{near}\) that lies in state \((0, 0)\). The transition that is randomly selected (Algorithm 3, line 33) is \((0, 0) \rightarrow (1, 0)\), meaning that robot 1 will try to grasp handle 1. According to tables 2 and 3, the constraints of the transition are \((\text{place}_1, \text{place}_1/\text{comp})\). The first one is of type \((16)\), the second one is of type \((18)\) and is parameterized: the right hand side uniquely defines the contact surfaces and the position of the object on the contact surface. \(q_{target}\) is obtained by projecting \(q_{rand}\) onto the manifold defined by the following constraints (Algorithm 3, lines 35–39):

- **place\(_1, place\(_1/\text{comp}\)** that belong to the transition,
Algorithm 3 Manipulation RRT algorithm iteratively calls method oneStep until a solution path is found or the maximum number of iterations is reached.

1: function INITIALIZEROADMAP(qinit,qgoal)
2: \( \Gamma \leftarrow \) new roadmap
3: \( \Gamma .ADDNODE(q_{init}); \Gamma .ADDNODE(q_{goal}) \)
4: function oneSTEP(\( \Gamma \))
5: newNodes \leftarrow \) empty list
6: \( q_{rand} \leftarrow \) SHOOTRANDOMCONFIG()
7: for \( cc \) in connected components of \( \Gamma \) do
8: for \( s \) in constraint graph states do
9: \( q_{near} \leftarrow NEARESTNODE(cc,s,q_{rand}) \)
10: \( p \leftarrow \) EXTEND\((s,q_{near},q_{rand})\)
11: if \( p \) then newNodes \leftarrow newNodes \cup \{p\}
12: \( nc \leftarrow TRYCONNECTNEWNODES(\Gamma ,newNodes) \)
13: if \( nc = 0 \) then
14: TRYCONNECTTOROADMAP(\( \Gamma ,newNodes) \)
15: function TRYCONNECTNEWNODES(\( \Gamma ,nodes\))
16: for \( q_1,q_2 \) in nodes, \( q_1 \neq q_2 \) do
17: \( s_1 \leftarrow \) STATE\((q_1); s_2 \leftarrow \) STATE\((q_2)\)
18: \( p \leftarrow \) CONNECT\((q_1,s_1,q_2,s_2)\)
19: if \( p \) then \( \Gamma .ADDEDGE(q_1,q_2,p) \)
20: function TRYCONNECTTOROADMAP(\( \Gamma ,nodes\))
21: for \( q_1 \) in nodes do
22: \( s_1 \leftarrow \) STATE\((q_1)\)
23: for \( cc \) in connected components of \( \Gamma \) do
24: if \( q_1 \notin cc \) then
25: \( near \leftarrow K \) nearest neighbors of \( q_1 \) in \( cc \)
26: for \( q_2 \) in \( near \) do
27: \( s_2 \leftarrow \) STATE\((q_2)\)
28: \( p \leftarrow \) CONNECT\((q_1,s_1,q_2,s_2)\)
29: if \( p \) then \( \Gamma .ADDEDGE(q_1,q_2) \)
30: function EXTEND\((s,q_{near},q_{rand})\)
31: solver \leftarrow SOLVERBYSUBSTITUTION
32: \( T \leftarrow \) random edge getting out of \( s \)
33: \( g \leftarrow \) state \( T \) points to
34: for \( c \) in \( g\).CONSTRAINTS() do
35: solver.ADDCONSTRAINT(c(q) = 0)
36: for \( c \) in \( T\).CONSTRAINTS() do
37: solver.ADDCONSTRAINT(c(q) = c(q_{near})
38: if \( q_{target} \leftarrow \) solver.SOLVE\((q_{rand})\)
39: if \( q_{target} \) then
40: \( p \leftarrow \) linear interpolation from \( q_{near} \) to \( q_{target} \)
41: \( p \).ADDCONSTRAINT(S(T).CONSTRAINTS())
42: \( p \) collision free then return \( p \)
43: else return collision-free portion of \( p \) starting at \( q_{near} \)
44: function CONNECT\((q_1,s_1,q_2,s_2)\)
45: \( T \leftarrow \) TRANSITION\((s_1,s_2)\)
46: if not \( T \) then return \( \emptyset \)
47: for \( c \) in \( T\).CONSTRAINTS() do
48: if \( \|c(q_2) - c(q_1)\| \geq \epsilon \) then return \( \emptyset \)
49: \( p \).ADDCONSTRAINT(c(q) = c(q_1))
50: if \( p \) in collision then return \( \emptyset \)
51: return \( p \)

Figure 3. Example of extension along a transition of the constraint graph. Top \( q_{rand} \), middle \( q_{near} \), bottom \( q_{target} \).

- \( g \) grasp\(_{11} \) that belongs to the goal state.

According to Section 5.3, the two first constraints can be replaced by an explicit constraint: the position of the object can be deduced from the right hand side of place\(_{1}/\)comp that is initialized with configuration \( q_{near} \).

After substitution, the set of constraints is reduced to an implicit constraint on the configuration variables of robot 1 (6 variables). The solution found by the solver, \( q_{target} \) (line 39) is displayed in Figure 3 bottom. Notice that as expected, the position of the object is the same in \( q_{target} \) as in \( q_{near} \).

The path returned by function extend is the linear interpolation between \( q_{near} \) and \( q_{target} \) constrained with place\(_{1} \), place\(_{1}/\)comp with right hand side initialized with
As explained earlier, this constraint is replaced by an explicit constraint. Let us notice that the linear interpolation already satisfies the constraint, but this is not always the case. If the latter path is in collision, the collision-free part of the path starting at \( q_{\text{near}} \) is returned.

Figure 4 illustrates method CONNECT applied to two configurations \( q_1 \) (top) and \( q_2 \) (bottom). Both configurations lie in state \((1, \emptyset)\)\(^5\). The transition between those states \((1, \emptyset) \rightarrow (1, \emptyset)\) contains the following constraints (tables 2 and 3):

- \( \text{grasp}_{11}/\text{comp}, \text{grasp}_{11} \).

Method CONNECT checks that the right hand side of \( \text{grasp}_{11}/\text{comp} \) is the same for \( q_1 \) and \( q_2 \), up to the error threshold (Algorithm 3, line 51). From a geometrical point of view, this means that the orientation of the cylinder along its axis, with respect to the gripper is the same in both configurations. Let us recall that the right hand side of \( \text{grasp}_{11} \) is 0. If the condition is satisfied, the method builds the linear interpolation between \( q_1 \) and \( q_2 \) with the explicit constraint equivalent to \( \{ \text{grasp}_{11}/\text{comp}, \text{grasp}_{11} \} \) and returns this path if it is collision-free.

### 6.3 Waypoint transitions

A prehensile manipulation motion contains by definition configurations that are in contact:

- between gripper and object during grasp,
- between object and contact surface when the object lies in a stable pose.

Contacts are difficult to handle by classical collision detection libraries and are often considered as collisions. To overcome this issue, we keep the gripper open during grasp, and objects slightly above contact surfaces in stable poses.

---

\(^5\)Note that \( q_1 \) lies at the intersection between states \((0, \emptyset)\) and \((1, \emptyset)\).
Even with these simple tricks however, solution paths to a manipulation problem need to come close to collision, raising the well-known issue of narrow passages.

To cope with this issue, we define intermediate states in the constraint graph called waypoint states. These states are inserted between the regular states of the constraint graph. They require some prior definitions.

**Definition 11.** *pregrasp* A pregrasp is a numerical constraint \( h \) over \( \mathcal{C} \), defined by

- a gripper \( g \),
- a handle \( h \),
- a non-negative real number \( \Delta \).

Let \( h \) be the smooth mapping from \( \mathcal{C} \) to \( \mathbb{R}^6 \) that maps to any configuration \( q \) of the system, the expression

\[
\tilde{h}(q) = \log_{\mathbb{R}^3 \times SO(3)} \left( g^{-1}h(q) \right) - (\Delta \mathbf{0} \mathbf{0} \mathbf{0} \mathbf{0} \mathbf{0})^T.
\]

(20)

\( h(q) \) is obtained by extracting from \( \tilde{h} \) the components of which are \( \mathbf{true} \) in the handle flag.

Note that when this constraint is satisfied, the handle is translated along \( x \) axis by distance \( \Delta \) compared to a configuration satisfying the grasp constraint. The value of \( \Delta \) depends on the geometry of the gripper and object. Clearance values are associated to the handle: \( cl_h \) and to the gripper: \( cl_g \). \( \Delta \) is defined as \( cl_h + cl_g \). The clearance parameters are part of the definition of the gripper and handle and are stored in SRDF files.

**Definition 12.** *preplacement* A preplacement is a numerical constraint \( h \) over \( \mathcal{C} \), defined by

- \((o_i)_{i \in I}\) a set of convex polygons attached to an object,
- \((f_j)_{j \in J}\) a set of convex polygons attached to the environment or to a mobile part of a robot that can receive objects (mobile robot for instance),
- a non-negative real number \( \Delta \).

with the same notation as in Section 5.2, we define \( i \) and \( j \) as the indices that minimize \( d(f_j, o_i) \) (Equation (14)), and

\[
\tilde{h}(q) = \log_{\mathbb{R}^3 \times SO(3)} \left( f_j^{-1}o_i(q) \right) + (\Delta \mathbf{0} \mathbf{0} \mathbf{0} \mathbf{0} \mathbf{0})^T
\]

(21)
The left hand side of the preplacement constraint is defined by Equation (16).

Note that when this constraint is satisfied, the object is translated of a distance \( \Delta \) along the normal to the contact surface.

We denote by

- *pregrasp* \( i \in \{1, \ldots, n_g\} \) \( j \in \{1, \ldots, n_h\} \) the pregrasp constraint of handle \( j \) by gripper \( i \),
- *preplace* \( i \in \{1, \ldots, n_o\} \) the preplacement constraint of object \( i \).

We replace the transitions of the constraint graph defined in Section 5.4 by a sequence of intermediate states and transitions as follows.

Following definition 9, if two states \( S_1 \) and \( S_2 \) are neighbors, one of them contains an additional grasp with respect to the other one. Without loss of generality, let us consider that \( S_2 \) contains the additional grasp \( gr(g_i, h_j) \), \( i \in \{1, \ldots, n_g\} \), \( j \in \{1, \ldots, n_h\} \). Then there are two cases:

1. the object handle \( h_j \) belongs to is already grasped in state \( S_1 \), or
2. the object handle \( h_j \) belongs to is in placement is state \( S_1 \).

In case 1, we replace the transitions between \( S_1 \) and \( S_2 \) by an additional waypoint state and four waypoint transitions as explained in Figure 5.

In case 2, we replace the transitions between \( S_1 \) and \( S_2 \) by three additional waypoint states and eight waypoint transitions as explain in Figure 6.

**Construction of a path along a waypoint transition**

Function *extend* in Algorithm 3 builds a path along a transition from an initial configuration by projecting the configuration onto the sub-manifold defined by the goal state constraints and by the transition constraints. The right hand side of the transition constraint is priory initialized with the initial configuration.

A waypoint transition builds a path by defining a sequence of configurations that lie in the intermediate waypoint states, each waypoint is obtained by projecting the previous waypoint onto the corresponding manifold. Figure 8 proposes an example of extension along edge \((\emptyset, \emptyset) \rightarrow (1, \emptyset)\) from configuration \( q_{near} \) (Figure 3 middle). The edge contains three waypoints. The random configuration \( q_{rand} \) is displayed in Figure 3 top. Table 4 shows the waypoint transitions as explain in Figure 6.
Figure 8. Example of extension along the waypoint transition between states \((\emptyset, \emptyset)\) and \((1, \emptyset)\). Each picture represents a waypoint. The last waypoint lies in state \((1, \emptyset)\). \(q_{\text{near}}\) and \(q_{\text{rand}}\) are the same as in Figure 3.

configurations that are produced when extending \(q_{\text{near}}\) toward \(q_{\text{rand}}\), and the constraints applied to compute these configurations.

Implementation From an implementation point of view, Class WaypointEdge derives from class Edge. The waypoint configurations are computed by method generateTargetConfig that is specialized in Class WaypointEdge.

Note that waypoint states are internal to waypoint edges and are not known by the constraint graph when determining in which state a configuration lies (Algorithm 3 lines 17 and 23) and when visiting the states of the constraint graph (Algorithm 3 line 8).

7 Humanoid Path Planner

In this section, we describe in more details the software platform Humanoid Path Planner that implements the concepts and algorithms of the previous sections.

<table>
<thead>
<tr>
<th>(q_{\text{near}})</th>
<th>in state ((\emptyset, \emptyset))</th>
</tr>
</thead>
<tbody>
<tr>
<td>waypoint state (g_1 &gt; h_1(\emptyset, \emptyset)</td>
<td>p_g) in state ((\emptyset, \emptyset))</td>
</tr>
<tr>
<td>constraints</td>
<td></td>
</tr>
<tr>
<td>(\text{place}_1(q) = 0)</td>
<td></td>
</tr>
<tr>
<td>(\text{place}<em>{1/\text{comp}}(q) = \text{place}</em>{1/\text{comp}}(q_{\text{near}}))</td>
<td></td>
</tr>
<tr>
<td>(\text{pregrasp}_{11}(q) = 0)</td>
<td></td>
</tr>
<tr>
<td>solver initialized with (q_{\text{rand}})</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(q_1)</th>
<th>in state ((\emptyset, \emptyset))</th>
</tr>
</thead>
<tbody>
<tr>
<td>waypoint state (g_1 &gt; h_1(\emptyset, \emptyset)</td>
<td>p_g) in states ((\emptyset, \emptyset)) and ((1, \emptyset))</td>
</tr>
<tr>
<td>constraints</td>
<td></td>
</tr>
<tr>
<td>(\text{place}_1(q) = 0)</td>
<td></td>
</tr>
<tr>
<td>(\text{place}<em>{1/\text{comp}}(q) = \text{place}</em>{1/\text{comp}}(q_{\text{near}}))</td>
<td></td>
</tr>
<tr>
<td>(\text{grasp}_{11}(q) = 0)</td>
<td></td>
</tr>
<tr>
<td>solver initialized with (q_1)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(q_2)</th>
<th>in state ((1, \emptyset))</th>
</tr>
</thead>
<tbody>
<tr>
<td>waypoint state (g_1 &gt; h_1(\emptyset, \emptyset)</td>
<td>p_g) in states ((\emptyset, \emptyset)) and ((1, \emptyset))</td>
</tr>
<tr>
<td>constraints</td>
<td></td>
</tr>
<tr>
<td>(\text{place}_1(q) = 0)</td>
<td></td>
</tr>
<tr>
<td>(\text{place}<em>{1/\text{comp}}(q) = \text{place}</em>{1/\text{comp}}(q_{\text{near}}))</td>
<td></td>
</tr>
<tr>
<td>(\text{grasp}_{11}(q) = 0)</td>
<td></td>
</tr>
<tr>
<td>solver initialized with (q_2)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(q_{\text{target}})</th>
<th>in state ((1, \emptyset))</th>
</tr>
</thead>
<tbody>
<tr>
<td>waypoint state (g_1 &gt; h_1(\emptyset, \emptyset)</td>
<td>p_g) in states ((\emptyset, \emptyset)) and ((1, \emptyset))</td>
</tr>
<tr>
<td>constraints</td>
<td></td>
</tr>
<tr>
<td>(\text{grasp}_{11}(q) = 0)</td>
<td></td>
</tr>
<tr>
<td>(\text{grasp}<em>{11/\text{comp}}(q) = \text{grasp}</em>{11/\text{comp}}(q_{2}))</td>
<td></td>
</tr>
<tr>
<td>(\text{preplace}_1(q) = 0)</td>
<td></td>
</tr>
<tr>
<td>solver initialized with (q_2)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Waypoint configurations computed along edge \((\emptyset, \emptyset) \rightarrow (1, \emptyset)\). The resulting path between \(q_{\text{near}}\) and \(q_{\text{target}}\) is a concatenation of constrained linear interpolation. Constraints applied between a waypoint and its predecessor are shown in blue.

Humanoid Path Planner is a collection of standard software packages that depend on each other. The main packages are the following:

- **hpp-fcl** a modified version of fcl. The main additional features are the following:
  - computation of a lower bound of the distance when testing collision between two objects. This is required for continuous collision detection,
  - security margins in collision checking,
- **pinocchio** Carpentier et al. (2019) a library computing forward kinematics and dynamics for multi-body kinematic chains,
- **hpp-constraints** a library that implements numerical constraints and solvers,
- **hpp-core** a library that implements most of the concepts relative to motion planning. The main features are
  - abstraction of paths in configuration spaces and some implementations,
  - abstraction of path planning and path optimization and some implementations,
  - abstraction of steering methods and some implementations,
  - roadmaps,
  - validation of configurations and paths,
• hpp-manipulation a library that implements manipulation problems and manipulation planning with
  – composite kinematic chains composed of the robots and objects,
  – the constraint graph,
  – M-RRT algorithm,
• hpp-manipulation-urdf an extension of the SRDF parser to retrieve information relative to objects, like the definition of grippers, handles, and contact surfaces.

An HPP session consists of a standalone executable hppcorbaserver that implements CORBA services. These services can be extended via a plugin system. The application can then be controlled via python script or C++ code. CORBA clients are provided in python and C++. The packages implementing CORBA clients and servers are

• hpp-corbaserver for canonical path planning problems, and
• hpp-manipulation-corba for manipulation problems. This latter package also provides an implementation of the automatic constraint graph construction in python.

The environment used for path planning as well as the paths computed can be displayed using gepetto-gui through packages

• gepetto-viewer,
• gepetto-viewer-corba, and
• hpp-gepetto-viewer.

7.1 Virtual machine

A virtual docker image can be downloaded to run, test and replicate the examples described in the next sections. An archive is provided with this paper. Inflate the archive and follow instructions in the README file.

7.2 Experimental results

In this section, we report on several experimental results obtained with HPP software on constrained motion planning and on manipulation planning problems. The raw data can be found in hpp_benchmark package. We present here only a few test cases. The benchmarks are run 20 times each on an Intel Core i7 at 2.60 GHz, with 32 Megabytes of RAM and 3 Gbytes of cache memory. For each test case, we report the minimum, maximum, mean and standard deviation of the time of computation on the one hand, and of the number of nodes in the roadmap built to solve the problem on the other hand.

7.2.1 Constrained motion planning One test case concerns constrained motion planning. The robot is an HRP-2 humanoid robot in quasi-static equilibrium that can slide on the ground. This type of motion can be post-processed into a walking motion using the method described in Dalibard et al. (2013). The results are displayed in Table 5.

7.2.2 Manipulation planning In this section, we present some experimental results of manipulation planning problems obtained with M-RRT algorithm described in Section 6.

The first test case features robot Baxter manipulating two boxes on a table (see Figure 10). The boxes are swaped

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
<th>mean</th>
<th>std dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (s)</td>
<td>0.03</td>
<td>11.64</td>
<td>1.32</td>
<td>2.55</td>
</tr>
<tr>
<td>nodes</td>
<td>4</td>
<td>136</td>
<td>32.40</td>
<td>30.72</td>
</tr>
</tbody>
</table>

Table 5. Experimental results for HRP-2 sliding on the ground: time of computation and number of nodes.
between the initial and final configurations. The robot has two grippers and each box is equipped with a handle. The constraint graph thus contains 7 nodes. The experimental results are displayed in Table 6.

The second test case features robot PR-2 manipulating a box on a table. The robot is requested to flip the box upside down from an initial pose to a goal pose as represented in Figure 11. The robot is equipped with two grippers, the box is equipped with two handles. The constraint graph contains 7 nodes. Table 7 shows the experimental results.

The third test case features Humanoid Robot Romeo manipulating a placard. The robot is requested to rotate the placard by 180 degrees. The robot is equipped with two grippers and the placard with two handles. Each handle is associated to a unique gripper. The number of states of the constraint graph is thus 3.

In the three previous test cases, the constraint graph was automatically built following Algorithm 2. If the number of grippers and handles increases, the number of states in the constraint may increase very quickly. Using python bindings, it is possible however to define constraint graphs with only the necessary states. We now present a test case
Figure 14. UR-5 robot manipulating a ball lying on a plane. The robot is requested to pick the ball and place it a few centimeters aside.

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
<th>mean</th>
<th>std dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>with waypoints</td>
<td>0.01</td>
<td>0.49</td>
<td>0.12</td>
<td>0.14</td>
</tr>
<tr>
<td>time (s)</td>
<td>4</td>
<td>30</td>
<td>9.75</td>
<td>7.26</td>
</tr>
<tr>
<td>nodes</td>
<td>4</td>
<td>30</td>
<td>9.75</td>
<td>7.26</td>
</tr>
<tr>
<td>without waypoints</td>
<td>4.15</td>
<td>73.53</td>
<td>26.24</td>
<td>17.07</td>
</tr>
<tr>
<td>time (s)</td>
<td>97</td>
<td>1609</td>
<td>711.55</td>
<td>407.98</td>
</tr>
<tr>
<td>nodes</td>
<td>97</td>
<td>1609</td>
<td>711.55</td>
<td>407.98</td>
</tr>
</tbody>
</table>

Table 10. UR-5 manipulating a box with and without waypoint transitions

that illustrates this possibility. The system is represented in Figure 13.

In this example, an operator provides the sequence of actions (transitions) the system needs to go through:

1. robot 1 grasps sphere 1,
2. robot 2 grasps cylinder 1,
3. robot 1 sticks sphere 1 on cylinder 1,
4. robot 1 releases sphere 1,
5. robot 1 grasps sphere 2,
6. robot 1 sticks sphere 2 on cylinder 1,
7. robot 1 releases sphere 2,
8. robot 2 put cylinder 1 on the ground.

From this sequence of actions, the sequence of states visited is computed and only those states (9 in total) are built in the constraint graph. Then, a sequence of subgoals in the successive states is computed, in such a way that each subgoal is accessible by the previous one (on the same leaf of the corresponding transition foliation). The subgoals are then linked by running a constrained visibility PRM algorithm (Simeon et al. (2000)) on each leaf. The python code can be found on github.com.

Figure 13 displays the initial configuration and the goal state. Table 9 shows the experimental results.

7.2.3 Influence of waypoint transitions All the previous experimental results have been obtained using waypoint transitions as described in Section 6.3. We now empirically show the positive effect of waypoints on the efficiency of manipulation planning. For that, we run 20 times Algorithm M-RRT on the same problem with and without waypoint transitions. The problem is defined by a UR-5 robot manipulating a ball as shown in Figure 14. The results are reported in Table 10. We can notice that for this example, the waypoint transitions make the time of computation and the number of nodes decrease by 2 orders of magnitude. This can be explained by the fact that in grasp configurations, the gripper is very close to the object and only a small part of the approaching directions of the gripper toward the object leads to collision-free paths. Waypoint states on the contrary are away from obstacles and easier to reach. The transition between the pregrasp waypoint and the grasp ∩ placement waypoint is always collision-free.

7.2.4 Analysis The experimental results show that M-RRT is able to solve a variety of manipulation planning problems including with a legged robot in quasi-static equilibrium. No parameter tuning is required between the different problems. All parameters are set to a default value for all test cases.

As for any random motion planning method, we observe a large standard deviation within the 20 runs of each test case, both for the number of nodes and for the time of computation.

We have also observed experimentally that the efficiency of M-RRT decreases when

1. the number of states to visit to solve a problem increases,
2. the number of foliated states increases.

For this reason, M-RRT is not able to solve the construction set problem in a reasonable amount of time. It remains to our knowledge however the only algorithm in the literature able to solve a variety of problems as large as those exposed in this section.

8 Conclusion and future work

This paper presents a software platform aimed at prototyping and solving a great variety of prehensile manipulation planning problems. The platform provides an original algorithm M-RRT that is an extension of RRT exploring the leaves of the foliations defined by the manipulation constraints. The automatic insertion of waypoint states makes the resolution more efficient and the resulting paths more natural.
We think that this platform is a perfect basis for researchers who want to develop and benchmark new manipulation planning algorithms. Note that some on-going work in humanoid locomotion Tonneau et al. (2018) is based on HPP.

To show the maturity of the project, we provide a docker image embarking the software.

As a future work, we aim at working on general manipulation planning algorithms that can handle use cases as diverse as those proposed in the benchmark section. A good candidate is a generalization of RMR* Schmitt et al. (2017). We also aim at working on manipulation path optimization since paths computed by random algorithms are too long to be applied on real robots as such.

Acknowledgements

This work has been partially supported by Airbus S.A.S. in the framework of the common laboratory Rob4Fam.

References


