HPP: a new software framework for manipulation planning

Florent Lamiraux, Joseph Mirabel

HPP: a new software framework for manipulation planning

Florent Lamiraux[1,2] and Joseph Mirabel[1,2]

***Abstract*— We present a new open-source software framework (called Humanoid Path Planner – HPP) for path and manipulation planning. Some features like built-in kinematic chains and implementation of non-linear constraints make the framework a good fit for humanoid robot applications. The implementation of kinematic chains takes into account the Lie-group structure (rotation in 3D space SO(3)) of robot configuration spaces. Robots and obstacles can be loaded from ROS-URDF files. Manipulation problems are modeled by a graph of constraints. At installation, a corba server is installed. the server can be controlled by python scripting to define and solve a problem.**

I. INTRODUCTION

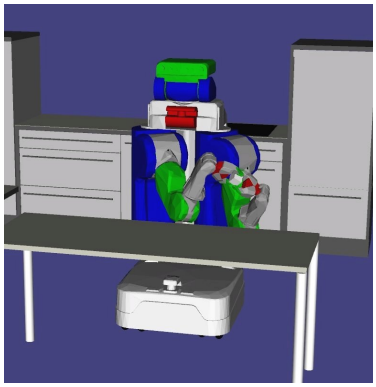

Fig. 1: PR2 passing a box from left hand to right hand

This paper presents a new open-source software framework (called Humanoid Path Planner – HPP) for path and manipulation planning, with some features dedicated to humanoid robots. The framework is the result of a deep refactoring work relying on a long standing experience in developing path planning algorithms. Note that this paper includes many links to external material (source code, documentation) that are not active on printed versions.

Unlike other existing open-source frameworks for path planning [1], [2], [3], the code is distributed into atomic software packages organized as a dependency tree hierarchy, with separate repositories. Developments have been focused on the algorithmic structure and only minimal effort has been devoted to graphical user interface.

As in Move3D [3], a unique class implements the concept of roadmap that is used by many path planning algorithms. A roadmap can therefore be used successively by several path planning algorithms. This makes our framework – hopefully – simple and easy to use. Robots and obstacles can be loaded from URDF files and applications can be controlled via python scripting.

1) Kinematic chain: Robots are modelled as trees of joints moving inertial and geometric bodies. As such our framework is closer to Openrave than to OMPL. Rotations in 3D space are represented by unit quaternions (4 components) in the robot configuration vector, while angular velocities are represented by 3D vectors. As such the configuration space of a robot is a Lie group. Velocity and configuration vectors do not necessarily have the same dimension.

2) Non-linear constraint: Non-linear constraints like position constraint of an end-effector, or quasi-static equilibrium of a legged robot are modeled at the core level and handled in the basic implementation of RRT [4] that is provided by default. This implementation is described in [5].

3) Manipulation: Several concepts already introduced in [6] have been integrated in order to define and solve manipulation planning problems. The mains concepts are composite robots containing a kinematic chain gathering several robots and objects, a graph structure the nodes of which define which robot is grasping which object with which effector. The edges of the graph define the connectivity between different grasps and provide additional constraints that should be satisfied along motions. For instance, an object may be grasped with various relative positions between the object and the effector (node constraint), but once the objects is grasped the relative position, between the effector and the object is fixed during motion (edge constraint).

4) Tutorial: A tutorial explains how to extend the framework to implement a new path planning algorithm. An example of basic PRM [7] is proposed to new users.

The contribution described in this paper is more the framework and the technical and architectural choices than the algorithms provided by the framework. The tutorial shows that implementing a new path planning algorithm only takes a few tens of lines.

In the following two sections, we describe in more details the core of the framework and the manipulation part built on top of the core part. Section IV shows a few examples of path and manipulation planning problems solved by the framework.

II. CORE PART

In this section, we describe the main concepts implemented in the core part of the framework. More details can

*This work has been partially supported by the national PSPC-Romeo 2 project, has received funding from the European Communitys Seventh Framework Programme (FP7/2007-2013) under grant agreement n 609206 and n 608849 and from the French agency ANR under grant agreement 13-CORD-002-01.

[1]CNRS, LAAS, 7 avenue du colonel Roche, F–31400 Toulouse, France
[2]Univ de Toulouse, LAAS, F–31400 Toulouse, France

type	configuration space	velocity space
translation	$\mathbf{R}$	$\mathbf{R}$ (linear velocity)
unbounded rotation	$\mathbf{S}^1 \subset \mathbf{R}^2$ (unit complex number)	$\mathbf{R}$ (angular velocity)
bounded rotation	$\mathbf{R}$ (angle)	$\mathbf{R}$ (angular velocity)
3D orientation	$\mathbf{S}^3 \subset \mathbf{R}^4$ (unit quaternion)	$\mathbf{R}^3$ (angular velocity)

TABLE I: types of joints provided by default with representation of their configuration space and their tangent space.

be found on the project page[1]. HPP is a collection of software packages developed in C++ for the algorithmic part and in python for the scripting control part. Installation is handled by `cmake`. Each package has its own repository hosted on `github`.

A. Kinematic chain

Kinematic chains are implemented in `hpp-model` as a tree of joints moving inertial and geometrical objects. Geometrical objects are modeled by a slightly modified version of `fcl` library[8]. We thus rely on this library for distance computations and collision checking.

1) Lie Group structure: The configuration space of a robot is the Cartesian product of the configuration spaces of its joints and possibly of a vector space called `ExtraConfigSpace` and described later. Table I displays the default joint types provided by `hpp-model`. Users can also specialize the abstract class `Joint` to define their own joint type. The configuration space of a robot is therefore a Lie-group. Configuration and velocity vectors do not necessarily have the same dimension. The representation is robust to singularities induced by minimal representations like (roll, pitch, yaw) for $SO(3)$ and do not suffer discontinuity when unbounded rotations cross π. Functions are provided to manipulate configuration and velocity vectors:

- `integrate` $(q, \dot{q})$ computes the configuration reached from q after applying velocity $\dot{q}$ during unit time,
- `difference` (q_2, q_1) computes the velocity that leads from q_1 to q_2 in unit time.

These two functions enable us to generalize most operations that are usually defined on vector spaces like linear interpolation or extension from a configuration toward another configuration (for RRT).

B. Differentiable functions

We have taken great care in modelling differentiable function that is a fundamental concept in motion planning. Paths are (piecewise) differentiable functions from time to the configuration space. Non-linear constraints are differentiable functions from the configuration space to a vector space that should be equal to a constant value. These abstract classes should be derived by users in order to express their own type of constraints or paths. Default implementations are provided for common constraints (position of center of mass, position / orientation constraint on an effector) and two implementations of path are provided as a straight interpolation in the configuration space and as a concatenation of paths.

[1]http://projects.laas.fr/gepetto/index.php/Software/Hpp

C. Steering method

Steering methods are represented by an abstract class that maps to a pair of configurations a continuous path between these configurations. Specializing this class enables users to plan motions for non-holonomic systems for instance. The default steering method returns the straight interpolation between the input configurations.

D. Extra configuration space

The dimension of the configuration of a robot is defined by the tree of joints of the robot. For some applications like kino-dynamic motion planning, it might be necessary to plan in the state space. For that, users can add an `ExtraConfigSpace` of appropriate dimension to the robot in order to store the velocity. Thus, roadmap nodes contain states instead of configurations. Together with a steering methods that link states, it is possible to implement classical kino-dynamic motion planning algorithms [4], [9], [10], [11].

III. Manipulation

In this section, we present the `hpp-manipulation` package. The package implements a planner for manipulation planning problems. It is made up by the following features, detailed later in this section:

- a definition of a manipulation planning problem with extra information;
- a model of interaction between robots and objects;
- a basic implementation of placement validation;
- an algorithm to solve manipulation problems.

A. Definition of the problem

A manipulation problem is defined by:

- Robots with end-effectors;
- Objects with handles;
- Environments with contact surfaces.

[6] has shown the high level of complexity of this problem. To our best knowledge, there has not been any significant breakthrough in the field that would provide an efficient algorithm able to solve this problem in the general case. The current version of HPP does not provide such an algorithm. Instead, HPP provides tools to model the problem. To the above list of elements a manipulation problem is defined by, we add the following element required by our solver to find a solution:

- A set of states and actions.

A state represents a relationship between robots and objects. Typically, an end-effector grasping a handle is a state. An action represents a transition between two states. From this, we build a graph called "constraint graph", the nodes being states and the edges being actions.

A node contains constraints a configuration should satisfy to be in the represented state. These constraints are "configuration constraints". An edge contains constraints a motion should satisfy to perform the represented action. These constraints are "motion constraints".

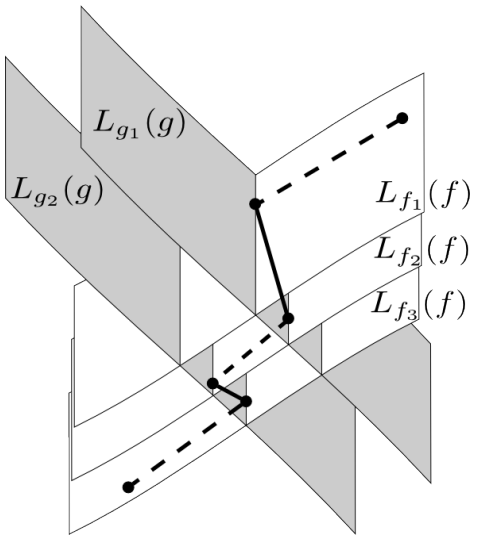

Fig. 2: Example of level sets of two constraints f and g in the configuration space. A manipulation path leading from one level set to another is represented, with its elementary paths. A dashed line, resp. solid, represents an elementary path in a level of f, resp. g. Note that every elementary path lie in a unique level set.

The notions of "configuration constraints" and "motion constraints" come from their effects on the configuration space. Constraints in HPP are defined by a differentiable function f. The level sets of f are submanifold of the configuration space. "configuration constraints" select only one of those level sets whereas "motion constraints" does not. For instance, the grasping constraints between a two fingers gripper and a cylindrical pole can be detailed as follow, z-axis being the cylinder main axis:

- translation along z-axis and rotation around z-axis are "motion constraints";
- translation along x and y axes and rotation around x and y axes are "configuration constraints";

The difference between "motion constraints" and "configuration constraints" is detailed later.

Robots, objects, their end-effectors and handles and contact surfaces are parsed from URDF and SRDF files. The description syntax is defined in `hpp-manipulation-urdf`. A python interface allows the user to create grasping constraints between an end-effector and a handle. The "constraint graph" is thus built through the python interface.

B. Manipulation planning concepts

The concepts implemented in HPP for manipulation planning were introduced in [6]. Let us define two "motion constraints" f and g, and the level set $L_{f_0}(f) = \{q \in \mathcal{C} | f(q) = f_0\}$. "Motion constraints" are built such that their level sets represent sets of configurations accessible to each other by the action represented by the constraint. For instance, f can represent the position of an object being manipulated. The robot manipulating the object can move freely while the object stands still, that is to say while f is constant, i.e. while the motion lies in a level set of f.

Figure 2 shows a configuration space and several level sets of f and g. Assume f returns the position of an object - f_1, f_2 and f_3 being valid placement of the object - and g returns a parameter uniquely identifying a grasp between the robot and the object.

Figure 2 also shows a possible path where the object goes from f_1 to f_3, i.e. goes from $L_{f_1}(f)$ to $L_{f_3}(f)$. The object cannot move without being grasped and the grasp cannot change while the object is not in a valid placement. Thus, the solution must be a sequence of path in $L_{f_i}(f)$ and $L_{g_i}(g)$. Using the framework developed in [6], the solution is a sequence of *transit* and *transfer* paths.

Figure 2 also shows a nice illustration of the difference between configuration constraints and motion constraints. Configuration constraints are intrinsic to a state and they will never change. "Motion constraints" have a reference that:

- is constant along a path;
- varies from one path to another.

They generate a foliation of the configuration space [6].

To picture the problem, one can think of navigation inside a multi-storey building. To move between different floors, you must find the intersection between the stairs and the floor. Once you are in some stairs, you must go through a floor if you need to take another stairs. Here, the set $\{L_{f_1}(f), \ldots, L_{f_n}(f)\}$ is the set of floors and the set $\{L_{g_1}(g), \ldots, L_{g_n}(g)\}$ is the set of stairs. An example of motion constraint is *stay in the floor* and its reference is the floor number.

C. Constraint graph

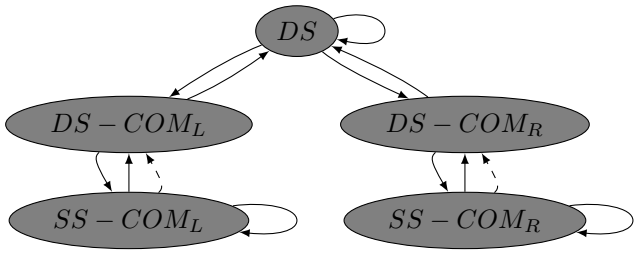

Fig. 3: "Constraint graph" for HRP-2 walking quasi-statically. DS stands for Double Support, SS for Single Support, COM for Center Of Mass, COM_L for COM on left foot and COM_R for COM on right foot. A solid edge, resp. dashed, represents a basic edge, resp. a "level set edge".

In HPP, a solution to a manipulation planning problem is a sequence of constrained paths. Each of this constrained paths refers to an edge of the "constraint graph". Figure 3 shows the "constraint graph" used to generate a quasi-static walk for HRP-2.

- The node DS represents HRP-2 standing on its two feet.
- The node $\text{DS} - \text{COM}_L$ represents the robot on its left foot, its right foot being on the ground. From that state, it is possible to lift the right foot from the ground while keeping the balance.

- The node $\text{SS} - \text{COM}_L$ represents the robot on its left foot, the right being unconstrained.
- The two other nodes $\text{DS} - \text{COM}_R$ and $\text{SS} - \text{COM}_R$ are their equivalent for the right foot.

Note that the graph is symmetric.

A configuration of the robot is in a node when it satisfies its constraints. As it may satisfy several node constraints, and to ensure that a configuration will never be in two nodes, these latter are prioritized. The set of configuration in $\text{DS} - \text{COM}_L$ is included in the set of configuration satisfying the constraints of node DS. To be accessible, $\text{DS} - \text{COM}_L$ must have a higher priority than DS. The order of priority, from high to low, for the graph in Figure 3 is: $\text{DS} - \text{COM}_R$, $\text{DS} - \text{COM}_L$, $\text{SS} - \text{COM}_R$, $\text{SS} - \text{COM}_L$, DS . So far, no automatic way of sorting the nodes has been implemented.

Several types of edges have been implemented in HPP and are explained in further details in section III-F. Among their common properties are:

- their ability to deduce a configuration from a random configuration, reachable from an initial configuration, ;
- their ability to generate a feasible path between reachable configurations of the two end nodes;
- the probability for an outgoing edge of a node to be selected with respect to the other outgoing edges of the same node.

D. Placement validation

Objects have an intrinsic property of placement. Some objects are always in a valid placement, for instance a door, whereas others are not. In order to generate feasible paths, an object must be either grasped by the robot or stand still in a valid placement.

HPP has a basic implementation of this feature based on contact surfaces. The environment description file must contain meta data defining where an object can be put. Object description files must also contain meta data defining where contacts may occur. Based on this information, the constraint builds pairs of $(\text{environment contact surface}, \text{object contact surface})$. For each pair, a distance is computed based on the contact normals and centers. The constraint value is the relative transformation between the elements of the pair with the shortest distance.

E. Manipulation RRT

A manipulation planner has been developed in HPP. The pseudo-code is given in Algorithm 1. The original version of RRT [4] cannot solve a manipulation problem because it cannot sample configuration on a submanifold that has a zero volume inside the configuration space; neither it deals with the task scheduling problem of a manipulation problem.

The main idea of the "manipulation RRT" algorithm is to integrate the discrete task scheduling problem into the motion planning problem. This is done by using the information stored in the "constraint graph". A pseudo-code of the algorithm is shown in Algorithm 1 and 2. It is a modified version of the RRT algorithm.

Algorithm 1 Manipulation RRT

```
1:  function EXPLORETREE(q_0)
    ▷ Randomly exploring Random Tree from q_0 using the constraint graph
2:      T.init(q_0)
3:      for i = 1 → K do
4:          q_rand ← RAND(CS)
5:          q_near ← NEAREST(q_rand, T)
6:          e ← CHOOSEEDGE(q_near)
7:          path ← CONSTRAINEDEXTEND(q_near, q_rand, e)
8:          if last step failed then Continue
9:          end if
10:         T .INSERTPATH(path)
11:     end for
12: end function
```

Lines 6 and 7 of Algorithm 1 differ from the classic RRT.

- Line 6: an outgoing edge of the node that contains q_{near} is randomly chosen. The choice honors the probabilities of the edges.
- Line 7: it refers to Algorithm 2. It generates a configuration that is in $L_{f(q_{near})}(f)$ where the edge constraints yield f.

Algorithm 2 extends a configuration in a level set of the configuration from which we extend.

Algorithm 2 Constrained extention

```
    ▷ Extend q_near towards q_rand while staying on the level set defined by edge and q_near
1:  function CONSTRAINEDEXTEND(q_near, q_rand, edge)
2:      f ← edge.MOTIONCONSTRAINT
3:      f.SETREFERENCE(q_near)
4:      q_new ← f.APPLYCONSTRAINTS(q_rand)
5:      p ← edge.BUILDPATH(q_near, q_new)
6:      p ← GETLONGESTNONCOLLIDINGPATHFROM(p)
7:      return p
8:  end function
```

F. Different types of edges

Additionally to the basic edge, which does not do more than what has already been said, two others types of edges have been introduced: the "waypoint edge" and "level set edge".

1) Waypoint Edge: to the basic edge, this edge adds waypoints in order to specify how a specific action is performed. For instance, one can specify a pre-grasping operation or the action to be done when a tool is being grasped. Waypoints are nodes of the "constraint graph" that can have only one ingoing and one outgoing basic edge. They correspond to states that make sense only in a succession of states. For instance, a pre-grasping posture precedes a grasping posture.

There is no randomness in what to do next. Figure 4 shows a "waypoint edge" with one waypoint. The two edges are basic edges. When using this edge to generate reachable configurations, resp. feasible paths, the solver will generate:

- reachable configurations q_w in the waypoint node, resp. feasible paths to the waypoint node;
- from q_w, reachable configurations in the goal node, resp. feasible paths to the goal node;

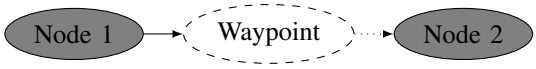

Fig. 4: Example of "waypoint edge", with one waypoint.

The difference between using "waypoint edge" and using two basic edges and a node is in most cases efficiency. With waypoints, the solver computes in one step what it would compute in as many steps as there are edges.

2) Level Set Edge: "level set edge" has been introduced to deal with a problem that is specific to manipulation planning and the foliation of the configuration space. Algorithm 1 shows the "manipulation RRT" algorithm. The solver builds a tree on both the initial and goal configurations. Every time a configuration is added to a tree, the solver tries to connect both trees. For the trees to be reachable, they must have visited a common leaf of the foliation, i.e. they must have a common $L_{f_{common}}(f)$ or $L_{g_{common}}(g)$, with the notations of Figure 2. If both the set $Range(f)$ and $Range(g)$ are not countable, the "manipulation RRT" has a zero probability of finding a common element in either $Range(g)$ and $Range(f)$.

The "level set edge" remembers the level sets that have been visited by each connected component of the roadmap. It uses then this information to generate reachable configurations on a level set that has been visited by a connected component different from the connected component of the node from which we extend. On Figure 2, a "level set edge" acting in $L_{g_{ref}}(g)$ will remember the set of $L_{f_{ref}}(f)$ visited by other connected components and will target $L_{g_{ref}}(g) \cap L_{f_{other}}(f)$. This strategy ensures that the two connected components have a non-zero probability of meeting.

Note that the "level set edge" targets the intersection but the generated path, valid regarding the manipulation rules, may be colliding and the "manipulation RRT" will shorten it from its end to have a collision-free path. It means a "level set edge" may generate paths that do not make connected component *visible* one from each other.

In Figure 3, a dashed line represents a "level set edge". Note the every "level set edge" is accompanied by a basic edge, so that we are sure the "manipulation RRT" will not only try to connect connected components directly but will also explore randomly level sets.

IV. Examples

HPP software has been successfully run on various examples two of which are presented here. Both examples are using `hpp-manipulation` package.

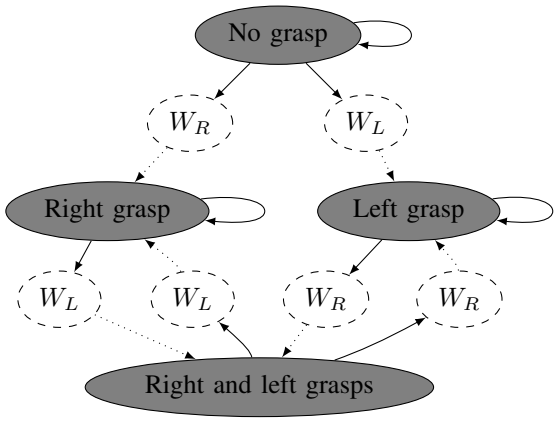

Fig. 5: Constraint graph describing PR2 manipulating a box with 2 hands. It is composed of 4 main nodes and 6 waypoints. Waypoint W_R corresponds to configurations where PR2 right hand is at 5cm from the box, the axis of the end-effector and the handle of the box being aligned. Waypoint W_L is similar with the left hand.

Computed paths are available here. Note that optimization for `hpp-manipulation` is still under developments so solution paths are not optimal.

A. PR2

In this example, PR2 manipulates a box. The problem has been designed such that it must pass the box from its left hand to its right hand without releasing the box. This example shows that, with a well-designed "constraint graph", the algorithm can generate a manipulation path with a narrow passage - the set of configuration where PR2 holds the box with its two hands is highly constrained.

The input of the problem is:

- Robot: PR2, its two hands being considered, its wheels being locked - it cannot navigate in the kitchen;
- Object: a box, with two possible grasps;
- Environment: a kitchen, with contact surfaces on the table, allowing the box to be put on it;
- Constraint Graph: shown in Figure 5;
- Initial configuration: the box is in a valid placement on the table, reachable only by PR2 left hand;
- Goal configuration: the box is in a valid placement on the table, reachable only by PR2 right hand.

Note that, in this case, as there is only two ways of grasping the box, the solver does not need a "level set edge" to make the two connected components meet. A "level set edge" would have been required if there had been an infinite number of grasps, for instance, by allowing the rotation around the axis going through the two fingers.

The "constraint graph" in Figure 5 has no edge from *Right grasp* to *No grasp* and from *Left grasp* to *No grasp*. They correspond to putting the ball on the table. This example aims at showing PR2 passing the box from one hand to the other. We thus did not allow PR2 to release the box. There is also no edge looping on *Right and left grasps* because it is not very useful in this example, though adding it would

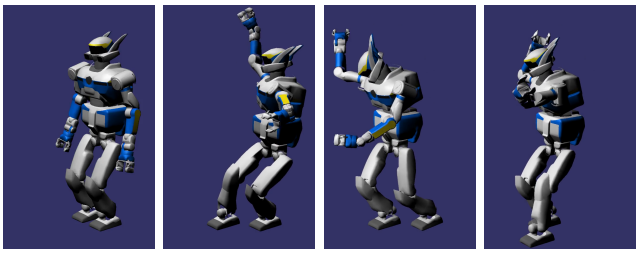

Fig. 6: HRP-2 walking quasi-statically.

not change much. There is no edge between *Right grasp* and *Left grasp* because it is not considered as an elementary step.

B. HRP-2

In this example, HRP-2 makes a few steps in a quasi-static equilibrium. The example aims at showing the planner ability to explore and find common level sets. Thus, the problem has been designed not to be achievable in only one step, but in such a way that common intermediate steps have to be found by the exploration from both ends of the paths.

The input of the problem is:

- Robot: HRP-2, its two feet can be on the ground, with three possible balance constraints;
- Object: None;
- Environment: None;
- Constraint Graph: shown in Figure 3;
- Initial configuration: HRP-2 is on its two feet, the COM between its feet.
- Goal configuration: the same as the initial configuration, shifted 50cm forward.

The foliated structure of the configuration space comes from the feet positions. Each of them is creating a foliation of the configuration space. Assume z-axis to be parallel to the gravity. Then, when a foot is on the floor, it is completely defined by x and y coordinates. Thus, the underlying problem is to find a common (x, y) for one of the feet. In Figure 3, the two edges from $\text{SS} - \text{COM}_L$ to $\text{DS} - \text{COM}_L$ represent a basic edge and a "level set edge". This allows the planner both to explore new random foot positions on the ground and to look for common foot positions.

V. Conclusion

The framework described in this paper aims at helping developers to implement their own path planning solution. Although it shares common features with already existing frameworks, we believe that original features like Lie-group representation for configuration space better fit some applications like aerial vehicle acrobatic motion planning for instance.

On-going developments concern the implementation of various path planning algorithm, and the implementing of path optimization techniques.

References

[1] R. Diankov and J. J. Kuffner, "Openrave: A planning architecture for autonomous robotics," Robotics Institute, Caneggie Mellon University, Pittsburgh, Pennsylvania, Tech. Rep., 2008.

[2] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, http://ompl.kavrakilab.org.

[3] T. Siméon, J. Laumond, and F. Lamiraux, "Move3d : a generic platform for path planning," in *4th International Symposium on Assembly and Task Planning*, 2001.

[4] S. M. Lavalle and J. J. Kuffner, "Randomized kinodynamic planning," *International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, May 2001.

[5] S. Dalibard, A. El Khoury, F. Lamiraux, A. Nakhaei, M. Taïx, and J.-P. Laumond, "Dynamic walking and whole-body motion planning for humanoid robots: an integrated approach," *The International Journal of Robotics Research*, vol. 32, no. 9-10, pp. 1089–1103, 2013. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00654175

[6] T. Simon, J.-P. Laumond, J. Corts, and A. Sahbani, "Manipulation planning with probabilistic roadmaps," *International Journal of Robotics Research*, vol. 23, no. 7/8, July 2004.

[7] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, August 1996.

[8] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE, 2012, pp. 3859–3866.

[9] E. Glassman and R. Tedrake, "A quadratic regulator-based heuristic for rapidly exploring state space," in *International Conference on Robotics and Automation (ICRA)*, 2010.

[10] A. Perez, R. Platt, G. Konidaris, L. Kaelbling, and T. Lozano-Pérez, "Lqr-rrt*: Optimal sampling-based motion planning with automatically derived extension heuristics," in *IEEE International Conference on Robotics and Automation*, May 2012, pp. 2537–2542. [Online]. Available: http://lis.csail.mit.edu/pubs/perez-icra12.pdf

[11] M. Bharatheesha, W. Caarls, W. Wolfslag, and M. Wisse, "Distance metric approximation for state-space rrts using supervised learning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2014.