HPP: a new software for constrained motion planning

Joseph Mirabel, Steve Tonneau, Pierre Fernbach, Anna-Kaarina Seppälä, Mylène Campana, Nicolas Mansard, Florent Lamiraux

HPP: a new software for constrained motion planning

Joseph Mirabel[1,2], Steve Tonneau[1,2], Pierre Fernbach[1,2], Anna-Kaarina Seppälä[1,2], Mylène Campana[1,2], Nicolas Mansard[1,2] and Florent Lamiraux[1,2]

***Abstract*— We present HPP, a software designed for complex classes of motion planning problems, such as navigation among movable objects, manipulation, contact-rich multiped locomotion, or elastic rods in cluttered environments. HPP is an open-source answer to the lack of a standard framework for these important issues for robotics and graphics communities.**

HPP adopts a clear object oriented architecture, which makes it easy to implement parts of an existing planning algorithm, or entirely new algorithms. Python bindings and a visualization tool allow for fast problem setting and prototyping: a new algorithm can be implemented in just a few lines of code.

HPP can be used for classic planning problems such as pick and place for mobile robots, but is specifically designed to solve problems where the motion of the robot is constrained. Examples of behaviors produced by HPP thanks to a generic constraint formulation include: maintaining a relative orientation between bodies, enforcing the static equilibrium of the robot, or automatically inferring that an object must be moved to allow locomotion. Constraints are tied to a custom representation of the kinematic chain, compatible with the Unified Robot Description format (URDF).

To illustrate the possibilities of HPP, we present several recent scientific contributions implemented with HPP, most of which are provided with Python tutorials. HPP aims at being seamlessly integrated within a global robot control loop: Pinocchio, the fast multi-body dynamics library, is currently being integrated in HPP, thus bridging the gap between the planning and control communities.

I. INTRODUCTION

Schwartz and Sharir define the motion planning problem as follows [1]: "*Given a body B, and a region bounded by a collection of "walls", either find a continuous motion connecting two given positions and orientations of B during which B avoids collision with the walls, or else establish that no such motion exists*". This formulation covers a "collision avoidance" aspect, and a "motion" aspect of the planning, both dependent on the body.

Motion Planning for collision avoidance

Efficient, generic algorithms [2], [3] have been proposed to explore the space of collision-free configurations [4]. The large majority of methods performs a sampling of this space, with the objective to capture its topology in a roadmap, a graph where nodes correspond to configurations of the robot, connected if a collision-free path exists between them.

Efficient implementation of these algorithms exist nowadays, in the frameworks OMPL [5], OpenRAVE [6], and Kineo CAM [7]. Thanks to them, assembly/disassembly tasks for manipulator arms, trajectory planning and control of wheeled robots, or human ergonomic studies are achieved on a daily-basis in the industry or the academia. These algorithms focus on the "collision avoidance" aspect, since the motion is controlled by well-known differential equations.

Constrained Motion planning

Recently, important scientific contributions have been proposed for more difficult classes of motion planning problems, where the augmented motion capabilities of the robot come with constraints on the movement.

For instance, multi-contact planning is the problem where an under-actuated multiped robot can only move through the contact forces exerted by its effectors on the environment [8]. Similarly in the problem of Navigation Among Movable Obstacles (NAMO) [9], [10], if a chair lies in front of a door, a robot may have to move it away before crossing the door. The latter problem can be seen as a motion planning problem where the body B is composed of both the chair (not able to move by itself!) and the robot. This formulation can be applied to the problem of manipulation [11], [12].

For a complex system such as a humanoid robot, all the mentioned aspects must be addressed simultaneously within the same framework [13]. For instance, the challenge course of the DARPA Robotics Challenge integrated multi contact planning, manipulation planning and NAMO altogether. One option is to implement ad-hoc planners for each problem, and to try to use them together, at the risk that each planner outputs conflicting solutions. The only open source solution which takes this approach (to our knowledge) is the Choreonoid software [14]. However it only implements a grasp plugin at the moment.

To our knowledge, no existing software chooses the other option, which is to propose a generic formulation for constrained motion planning, aiming at simplifying programming burden, but also at addressing simultaneously all constraints within the same algorithm, thus avoiding conflicts. **The motivation for developing HPP is to propose the first generic and open source implementation of a constraint-based motion planning software.**

Contributions and paper structure

HPP is a set of C++ open-source libraries for Linux. The first building block is `hpp-model`. It provides definitions and functions for the kinematic model of a robot and the geometric objects of a problem (Section II). The key library is `hpp-constraints`, a generic constraint framework implemented on top of `hpp-model` (Section III). The motion planning features of HPP are implemented in `hpp-core`,

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

using a object-oriented architecture that allows customizing parts of a motion planning algorithm in a few lines of C++, or in Python through CORBA bindings (Section IV).

At the end of the paper, we present several cases studies: we compare HPP with OMPL in classic motion planning scenarios, and present several scientific contributions that illustrate the interest of HPP (Section V).

II. Kinematic model

Contrary to OMPL, and similarly to OpenRAVE and Choreonoid, HPP commits to a kinematic model. This is justified by the fact that implementing a generic constraint system requires defining the notion of `Joint` and its associated joint velocity space, organized in a kinematic tree (or `Device`). Though it is possible to extend the `hpp-model` library with custom `Joint` types and kinematic chains, HPP conveniently handles natively most of the common joints.

URDF files are used for the external representation of the kinematic model, making it compatible with the Robotics Operating System (ROS) [15].

A. Joint and Kinematic chain implementation

A kinematic chain is implemented as a tree of joints moving inertial and geometrical objects. 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`, used for manipulating external objects, and described later). Table I displays the default joint types handled in HPP. Users can also define their own joints.

We explicitly and natively handle joints for which the configuration space is a Lie group [16] (e.g. ball joints). For these joints, we use a representation both continuous and robust to singularities (e.g. for ball joints, the space is $SO(3)$, and we use quaternions rather than Euler angles representation). It results that the joint velocity has a smaller dimension than the joint configuration (i.e. the velocity belongs to the Lie Algebra, tangent to the Lie Group).

B. Operations on the configuration space

The configuration of the robot is described by a vector $\mathbf{q} \in \mathbb{R}^n$, where n is the sum of dimensions of each joint, while the configuration velocity, denoted (abusively) by $\dot{\mathbf{q}}$, may have a smaller dimension. Functions are provided to manipulate configuration and velocity vectors:

- `integrate` $(\mathbf{q}, \dot{\mathbf{q}})$ computes the configuration reached from $\mathbf{q}$ after applying velocity $\dot{\mathbf{q}}$ during unit time.
- `difference` $(\mathbf{q}_1, \mathbf{q}_2)$ computes the velocity that leads from $\mathbf{q}_1$ to $\mathbf{q}_2$ in unit time.

These two functions are a generalization of most simple operations typically used in motion planning on vector spaces, such as linear interpolation between two configurations.

Geometrical objects are stored using a modified version of `FCL` [17]. We thus rely on this library for distance and collision computations.

Type	Configuration space	Velocity space
Prismatic (1D)	$\mathbb{R}$ (translation)	$\mathbb{R}$ (linear)
Unbounded revolute (1D)	$\mathbf{S}^1 \subset \mathbb{R}^2$ (unit complex)	$\mathbb{R}$ (angular)
Bounded revolute (1D)	$\mathbb{R}$ (angle)	$\mathbb{R}$ (angular)
Ball joints (3D)	$\mathbf{S}^3 \subset \mathbb{R}^4$ (unit quaternion)	$\mathbb{R}^3$ (angular)

TABLE I: Main types of joints provided by default. For each joint, we specify the representation of their configuration space ($\mathbf{q}$ vector) and their tangent space (velocity $\dot{\mathbf{q}}$ vector).

III. Differentiable functions as constraints

The key asset of HPP is the implementation of non-linear constraints as differentiable functions. HPP also provides methods to automatically project a sampled configuration into the configuration space described by the constraint.

OMPL does not handle constraints natively since it does not commit to a kinematic model. Regarding OpenRAVE, and the integration of OMPL with Moveit!, it appears that only positional constraints are handled (inverse kinematics), and mostly limited to manipulator arms. For instance OpenRAVE relies on analytic inverse kinematics methods, which do not apply to chains with more than 6 or 7 degrees of freedom (unless some are disabled). Choreonoid on the other hand also handles static equilibrium and advanced manipulation constraints, in two independent plugins, with no trivial mean to coordinate them.

HPP handles indifferently **any** constraint that can be written with a differentiable function in a unified way, and proposes several default implementations, including the one proposed by other softwares (see the list below). Any number of constraints can be handled simultaneously.

A. Non-Linear equality and inequality constraints

Differentiable functions can be used by a motion planner to ensure that the configurations of a path (or a subpath) verify one or several constraints. Given a function f that takes its values in a vector space of arbitrary dimension m, a constraint can be defined as:

$$f(\mathbf{q}) = \mathbf{c}$$

where $\mathbf{c} \in \mathbb{R}^m$ is constant. A user-defined constraint is thus defined by three elements: the function f, the objective value $\mathbf{c}$, and the Jacobian of f. Automatic differentiation is also implemented to spare the need of manually computing the analytical formulation of the Jacobian of f, thanks to a symbolic library. These parameters are used in a Newton's descent algorithm to project a given configuration on the constraint. The user implements these elements by overloading the abstract class `DifferentiableFunction`. Several concrete constraints are implemented in HPP, among which:

- orientation and / or position (for inverse kinematics);
- relative orientation and / or position of two bodies;
- static equilibrium;
- position of the center of mass;
- distance between bodies;
- contact between a robot body and a surface.

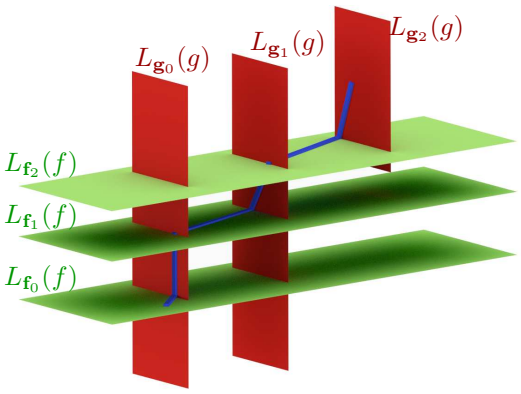


Fig. 1: Example of level sets of two functions f and g in the configuration space. A global path (blue) is decomposed as elementary paths in each level set. Note that every elementary path lie in a unique level set. For illustration purposes, one can consider the $L_{\mathbf{f}_s}(f)$ constraints as relative to floors, and the $L_{\mathbf{g}_s}(g)$ constraints as relative to stairs, for a problem consisting in navigating through floors.

Additionally, HPP can also handle inequality constraints, written in the form $f(\mathbf{q}) \leq \mathbf{c}$. One motivational example for inequality constraints is to address a scenario where a robot carries a glass containing a liquid that must not be spilled. This can be expressed by constraining the angle between the glass inclination and to be lesser than a threshold value, and thus be automatically handled by HPP in the planning.

B. Constraint graphs

Furthermore, HPP supports constraint graphs [11]. This representation has the huge advantage of allowing the integration of the discrete, higher-level task scheduling problem into the motion planning problem.

A constraint graph can be seen as a finite state machine, with special nodes and transitions. Each node is characterized by a constraint. The constraint is the level set of a function f denoted by $L_{\mathbf{c}_0}(f) = \{\mathbf{q} \in \mathcal{C} | f(\mathbf{q}) = \mathbf{c}_0\}$. $L_{\mathbf{c}_0}$ is a submanifold of the configuration space. The submanifolds form a foliation of the configuration space, and intersect in a combinatorial manner. Sampling configurations at the intersection of the submanifolds is required to travel between the level sets. A metaphor for this issue is to consider the problem of going from one floor to another floor of a building, which requires to cross some stairs. Each floor and staircase can be seen as level sets, illustrated in Fig. 1. In general, the probability to sample configurations at the intersection of manifolds is small or null. A constrained motion planner must account for this property and explicitly bias the sampling, typically using the projectors presented in Section III-A.

HPP is provided with an implementation of such algorithm, called Manipulation-RRT, presented in details a paper submitted to IEEE IROS [18]. It takes a constraint graph as input, and automatically generates a motion that respects the transition rules of the graph. This formulation can be used to address simultaneously various problems such as quasi-static locomotion (Fig. 2), advanced manipulation (Fig. 3), and even more complex problems, which require multiple manipulations of the same object (such as the Hanoï towers game).

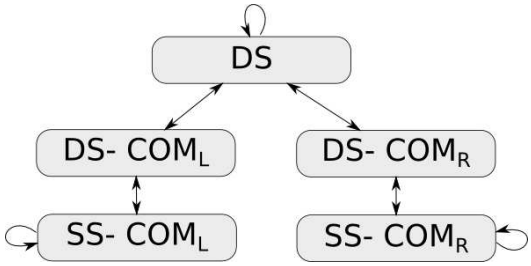


Fig. 2: "Constraint graph" for quasi-static walk on flat ground. Such locomotion requires the Center Of Mass lying above the support polygon defined by the contact points at all times. The difficulty to handle the transition between double and simple support phases is automatically addressed with this constraint graph. DS stands for Double Support, SS for Single Support, and COM$_L$ (resp COM$_R$) for the Center Of Mass lying above the left (resp. right) foot.

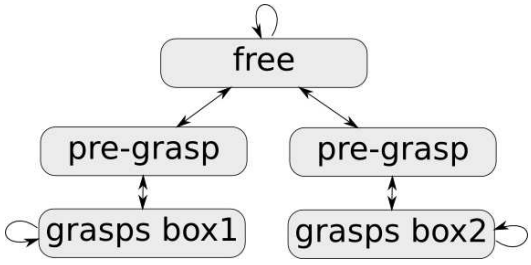


Fig. 3: "Constraint graph" for a manipulation scenario involving two boxes. The graph is similar to the walking graph. The "pre-graph" constraints correspond to a box being in contact with a surface, with the robot gripper seizing them. From this state the box can be released safely ("free"), or grasped and moved along with the effector ("grasp").

IV. Using HPP

HPP can be used either as an off-the self motion planner, or as a research tool to implement and test new algorithms. Similarly to existing softwares, it provides Python bindings. Additionally benchmarking tools compatible with the OMPL API are provided. Lastly, a unique feature of HPP is the automatic export of computed motion to the Blender [19] animation software for high-quality presentation videos.

A. CORBA Architecture

The client / server CORBA architecture adopted by HPP is an essential asset. It allows to consider integration with other softwares developed in different languages, as well as distributed execution on different robots and computers. One such integration is delivered in HPP in the form of exhaustive Python bindings. Efficient C++ algorithms can

be easily called from simpler Pythons scripts, and allows to interact with the software using a command line interpreter, thus enhancing interactivity.

Thus these bindings are really useful for fast prototyping and testing new algorithms. Setting up a problem, and even implementing a new motion algorithm, only requires a few lines of code, as shown in the following lines.

B. Setting up and solving a problem

HPP is delivered with tutorials addressing classic or constrained motion planning, available on the documentation. Code Listing 1 provides an example code for setting up a problem. It shows that the helper class `ProblemSolver` can be used to customize a problem solver: a single operation is required to change any component of the planning (here, the motion planner type and the path optimization algorithm).

Listing 1: Python code to set up and solve a problem

```
from hpp.corbaserver.pr2 import Robot
robot = Robot ("pr2")
robot.setJointBounds ("base_joint_xy",
                      [-6, -3, -5, -3])

from hpp.corbaserver import ProblemSolver
ps = ProblemSolver (robot)
ps.selectPathPlanner("VisibilityPrmPlanner")
ps.addPathOptimizer("GradientBased")

# copy the initial configuration
q_init = robot.getCurrentConfig ()
q_goal = q_init [::]
# ask the robot to move backwards
q_goal [0:2] = [-6, -3]

from hpp.gepetto import Viewer
r = Viewer (ps)
# helper function to load the obstacle mesh
# into both viewer and problem solver
r.loadObstacleModel ("iai_maps",
                     "kitchen_area", "kitchen")

ps.setInitialConfig (q_init)
ps.addGoalConfig (q_goal)
ps.solve ()
```

Constraints and constraint graphs are created by instanciating provided or user-defined classes. For arbitrary constraints, the constraint graph has to be specified manually. For placement constraints, as in Fig. 3, the constraint graph can be automatically built with helper functions.

C. Testing, benchmarking, and presenting results

In HPP productiveness is enhanced with tools for generating comparative results and demonstrations.

First, HPP is compatible with the benchmarking API of OMPL (although as of today not all features are implemented). A Python script can be used to produce the desired output, thus facilitating comparison with other algorithms. We used this API for the benchmarks presented in Section V.

Then, HPP is delivered with a viewer based on OpenSceneGraph, the Gepetto viewer (`Viewer` in Listing 1). The viewer can receive Python command line prompts, and be used *via* a GUI, for testing problems.

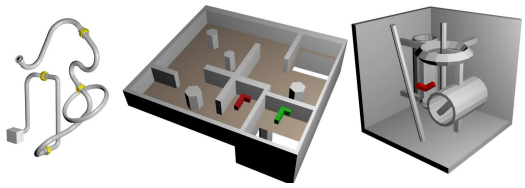


Fig. 4: Pipedream-Ring, Abstract and Cubicles benchmarks.

Lastly, HPP includes Python methods to automatically export a computed motion to the open-source Blender software. Entire paths can be exported with a simple method call, and loaded into Blender by executing a Python script. This allows the automatic production of clearer, better looking demonstrations of scientific contributions.

D. Extending HPP

Thanks to the modular architecture of HPP, any part of a motion planning algorithm can be extended and used in a transparent manner with the rest of the framework. Thanks to the CORBA architecture, it is also trivial to implement the Python bindings for new user-defined methods. For prototyping purposes, it is even possible to write a complete motion planning algorithm in a few lines of Python. For instance, the standard RRT planner can be implemented with the short code provided in the Appendix.

V. Results

A. Benchmarks

We compared the performance of the implementation of the RRT-connect algorithm of OMPL and HPP. OMPL proposes other planners, not implemented in HPP. We did not find other benchmarks from other softwares to compare ourselves too. This benchmark shows that the efficiency of the default planner of HPP is comparable to OMPL.

To carry out the comparison, we used the benchmark database provided by OMPL, picking the three problems solved with RRT-Connect. Figure 4 shows a screen capture of each of the three problems[1]. To provide a comparison as fair as possible, we had to take into account some implementation details of OMPL and HPP. Firstly, OMPL uses a `range` parameter, which determines the maximum distance between two nodes in the roadmap, automatically computed for each scenario. Depending on the benchmark, this value can improve or slow down the computation time. The HPP implementation does not use such parameter.

Secondly, HPP includes a continuous collision checking method, additionally to the classic discretized one. The advantage of this method is that it does not require specifying a discretization step to test collisions along a path. A continuous collision test has a higher atomic cost than a discretized one, but in general it improves the computation

[1] In the third scenario (Pipedream-Ring), no mesh of the ring-shaped robot was provided by OMPL, so we replaced it with a ring mesh of 982 triangles.

scenario	min time (s)				avg time (s)				max time (s)			
	HPP-D	HPP-C	OMPL	OMPL-NR	HPP-D	HPP-C	OMPL	OMPL-NR	HPP-D	HPP-C	OMPL	OMPL-NR
1. Pipedream-Ring	0.065	0.043	0.458	0.618	1.242	2.053	2.998	4.237	6.519	7.356	10.483	14.071
2. Abstract	0.159	0.408	23.523	14.345	47.654	34.395	106.866	106.814	257.573	178.013	296.518	269.94
3. Cubicles	0.049	0.024	0.096	0.118	0.271	0.130	0.277	0.329	0.902	0.946	0.665	1.059

scenario	avg number of nodes				success rate (%)				time-out (s)
	HPP-D	HPP-C	OMPL	OMPL-NR	HPP-D	HPP-C	OMPL	OMPL-NR	
1. Pipedream-Ring	2283	2452	16100	22681	100	100	100	100	20
2. Abstract	11927	10807	177914	181427	94	94	96	98	300
3. Cubicles	495	302	261	307	100	100	100	100	20

TABLE II: Results for 50 runs of each considered planner. Green values are used when the HPP implementation performs better than all OMPL implementations. Red values are used when HPP performs worse than at least one OMPL implementation. Statistics are only considered when the planning succeeded. A planning is considered to have failed after running longer than the specified timeout value.

times because less tests are required by the planning (only one test is required on collision free sections of a path, regardless of its length).

To compare HPP and OMPL on an equivalent implementation of RRT-Connect, we consider on one hand a "no range" version of OMPL (OMPL-NR), where the range is set to a high value, thus not considered. On the other hand we consider a HPP implementation that uses discretized collision checking (HPP-D). The discretization step used for the discrete collision checking is the same in HPP than the one chosen in OMPL[2]. However to be exhaustive, and to make sure the planners are also compared relatively to their nominal use, we also included benchmarks performed with the specificities of the softwares: we thus also consider the standard "range" version of OMPL (OMPL), and the continuous collision checking version of HPP (HPP-C).

Table II presents the results for all three scenarios and implementations. The success rate represents the relative number of runs that succeeded before a given maximum time limit. When computing minimum, average and maximum time values, only successful runs were considered. The runs, single-threaded, were performed on a 64 bits computer with 8 processors of 1.2Ghz, 64Go or RAM.

In any considered case, HPP implementation presents equivalent or better average computation times compared to OMPL. The important point is that the performances remain in the same order of magnitude between HPP and OMPL.

B. Use cases

HPP is already actively used for research purposes, some of which are presented here. All the presented projects are open source and accessible to the community on github.com. Most involve constraint-based motion planning, for which HPP is specifically designed, though other applications demonstrate that HPP can be used for classic motion planning, or even be integrated within third-party simulations. The results obtained are presented in the companion video.

1) NAMO and Manipulation: Manipulation-RRT is implemented in the module `hpp-manipulation`. This algorithm can be used indifferently for complex manipulation and NAMO. A first example scenario shows the Baxter robot permuting the positions of three boxes on a table (Fig. 5). The only inputs specified are the start and goal configurations of the boxes, as well as the constraint graph from Fig. 3, extended to handle a third box. In the second example the Romeo robot puts an object in a fridge (Fig. 6 while maintaining equilibrium. Again, only the final position of the object is specified. With the constraint graph the door is automatically grasped and openend. These two examples are developed in [18].

2) Gradient-based path optimization: `hpp-core` provides a gradient-based path optimizer to refine the indirect paths generated by probabilistic planners [20]. The algorithm is a Linearly Constrained Quadratic Program that modifies the waypoints of a path to make it shorter. To do so, constraints are automatically generated between the objects that might collide during the optimization. Thus, only part of the configuration variables are constrained at some points of the path, while others are optimized. A result based on a PR2 robot avoiding a table is presented in Fig. 7.

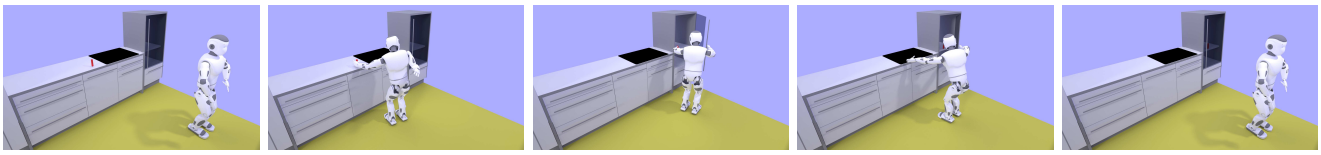
3) Acyclic contact planning: is a class of problem where an under-actuated multiped robot must be in contact at every configuration to maintain static equilibrium, and exert the forces allowing it to move. We address this issue sequentially: first, a path is computed for the root of the robot, in a low dimensional space. Then along this path, a discrete sequence of "key" contact configurations is computed. These key frames are interpolated dynamically using a 3D pattern-generator [21]. The first two steps are implemented in HPP, using a planner called RB-RRT [22], [23], for Reachability-Based planner. RB-RRT uses extensively position and orientation constraints to maintain and generate contacts. It also takes advantage of the flexibility of HPP to easily replace the default sampling method for a variant of OB-PRM [24]. The pattern generator is implemented thanks to the Pinocchio dynamics library [25], soon to be integrated with HPP. A stair climbing using a handrail scenario is illustrated in Fig. 8.

4) Elastic rod planning: A special case of manipulation planning for an extensible elastic rod, either collision-free or in contact [26]. We assume the rod can be handled by grippers at one or both extremities. External libraries

[2]converted to the standard metric system from OMPL that uses inches.

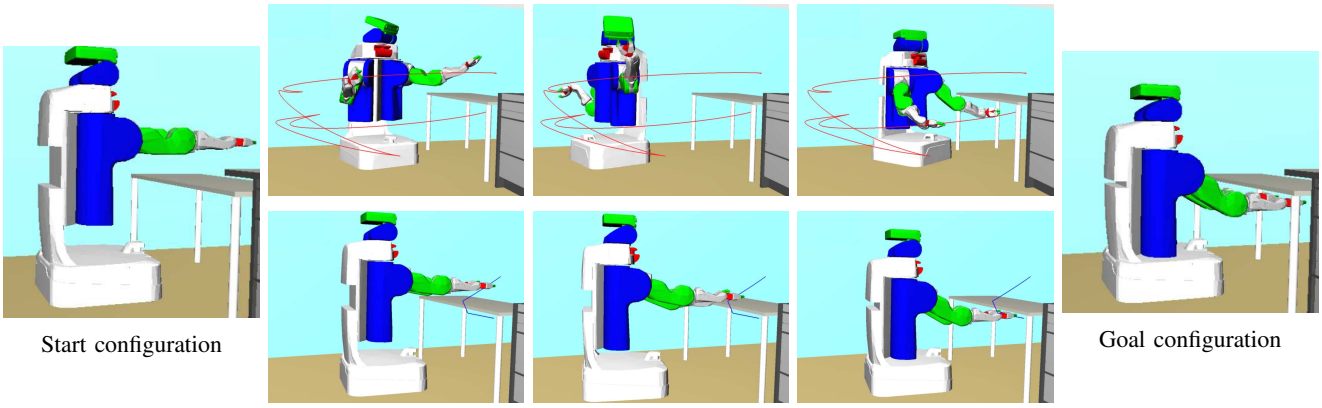
(a) Start Configuration (b) Grasp green and red (c) Release red / grasp blue (d) Release blue and green (e) Goal configuration

Fig. 5: A complex manipulation example for the Baxter robot. The task is to swap the position of three boxes. This requires a sequential task decomposition, automatically inferred by Manipulation-RRT .



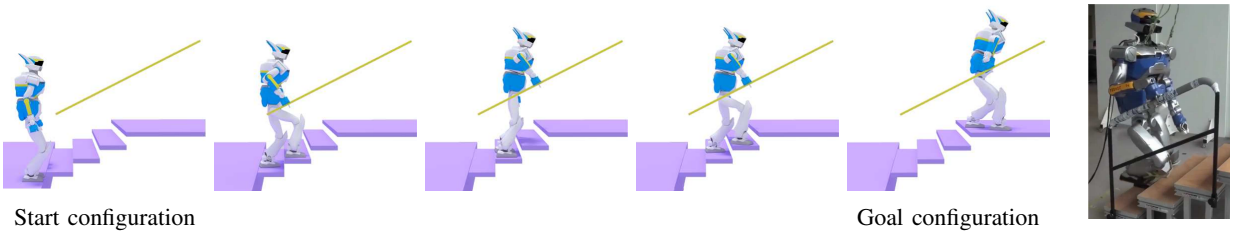
(a) Start Configuration (b) Grasp object (c) Open fridge / place object (d) Close fridge door (e) Goal configuration

Fig. 6: A complex manipulation example for the Romeo robot. The task is to put an object in a fridge. The planner automatically infers that the fridge door must be opened.



Start configuration Goal configuration

Fig. 7: Illustration of the path optimization algorithm. Given the start and goal configurations shown on the left, a rough path is computed with a sampling-based planner (top). The path is reduced to limit the motion of all joints (bottom). The red (resp. blue) curve denote the path followed by the right effector along the motion.



Start configuration Goal configuration

Fig. 8: Multi-contact planning for the HRP-2 robot climbing stairs using a handrail, a typical scenario proposed by the DARPA challenge. Through a connection with the Pinocchio library, the plan can be executed on the real robot (right).

(QSERL and XDE) are used to compute the deformation and the dynamics of the rod. This demonstrates the compatibility of HPP with external software. A dedicated steering method, which uses these libraries, is implemented within a slightly modified RRT algorithm. This implementation allows to plan a motion for an elastic rod egressing from a complex engine through a small hole (Fig. 9).

VI. Conclusion

This paper introduces HPP, the Humanoid Path Planner, a constraint-based motion planning software. HPP provides a framework for easily testing and implementing algorithms inspired from recent contributions from the scientific community, including Navigation and Among Movable Obstacles, manipulation and acyclic contact planning.

To our knowledge no existing software provides implementations to address these issues in a unified manner. While in theory it is possible to extend existing softwares to implement these algorithms, integrating constraint-related features from the conception of HPP allows for a simple architecture, providing many helper methods to facilitate the use of complex notions such as differentiable functions and constraint graphs. Moreover, this unified architecture also comes with excellent performances, and HPP is able to compete with the best actual planning implementations on the standard benchmarks.

Bindings of the API using a CORBA architecture allow for fast prototyping and testing in Python, and a simple integration with other softwares. It is also delivered with benchmarking tools compatible with the OMPL benchmark API, and Blender export functions for high-quality rendering.

HPP is already a mature software, which has been used to implement several new scientific contributions, also demonstrated in the companion video. They demonstrate the ability to integrate HPP with complex third-party softwares, and offer a glimpse of the future features that HPP will be provided with. Among those, the integration of the Pinocchio dynamic library holds the promise of a seamless framework between motion planning and its execution on a real robot.

References

[1] J. T. Schwartz and M. Sharir, "On the piano movers' problem i. the case of a two-dimensional rigid polygonal body moving amidst polygonal barriers," *Communications on Pure and Applied Mathematics*, vol. 36, no. 3, pp. 345–398, 1983. [Online]. Available: http://dx.doi.org/10.1002/cpa.3160360305

[2] 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.

[3] S. M. LaValle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," *In*, vol. 129, no. 98-11, pp. 98–11, 1998.

[4] T. Lozano-Pérez, "Spatial planning: A configuration space approach," *IEEE Trans. on Computers*, vol. C-32, pp. 108–120, 1983. [Online]. Available: http://lis.csail.mit.edu/pubs/tlp/spatial-planning.pdf

[5] 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.

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

[7] Wikipedia, "Kineo cam — wikipedia, the free encyclopedia," 2015, [Online; accessed 19-February-2016]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Kineo_CAM&oldid=647097086

[8] T. Bretl, "Motion planning of multi-limbed robots subject to equilibrium constraints: The free-climbing robot problem," *The Int. Journal of Robot. Research (IJRR)*, vol. 25, no. 4, pp. 317–342, Apr. 2006. [Online]. Available: http://dx.doi.org/10.1177/0278364906063979

[9] M. Stilman and J. Kuffner, "Navigation among movable obstacles: Real-time reasoning in complex environments," in *Proceedings of the 2004 IEEE International Conference on Humanoid Robotics (Humanoids'04)*, vol. 1, December 2004, pp. 322 – 341.

[10] M. Levihn, L. Kaelbling, T. Lozano-Perez, and M. Stilman, "Foresight and reconsideration in hierarchical planning and execution," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, Nov 2013, pp. 224–231.

[11] T. Siméon, J.-P. Laumond, J. Cortés, and A. Sahbani, "Manipulation Planning with Probabilistic Roadmaps," *The International Journal of Robotics Research (IJRR)*, vol. 23, no. 7-8, pp. 729–746, 2004.

[12] K. Harada, T. Tsuji, and J.-P. Laumond, "A manipulation motion planner for dual-arm industrial manipulators," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE, 2014, pp. 928–934.

[13] F. Burget, A. Hornung, and M. Bennewitz, "Whole-body motion planning for manipulation of articulated objects," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, May 2013, pp. 1656–1662.

[14] S. Nakaoka, "Choreonoid: Extensible virtual robot environment built on an integrated gui framework," in *System Integration (SII), 2012 IEEE/SICE International Symposium on*, Dec 2012, pp. 79–85.

[15] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[16] R. M. Murray, S. S. Sastry, and L. Zexiang, *A Mathematical Introduction to Robotic Manipulation*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1994.

[17] 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.

[18] J. Mirabel and F. Lamiraux, "Constraint graphs: Unifying task and motion planning for navigation and manipulation among movable obstacles," 2016. [Online]. Available: http://cpc.cx/eOE

[19] Blender Online Community, *Blender - a 3D modelling and rendering package*, Blender Foundation, Blender Institute, Amsterdam. [Online]. Available: http://www.blender.org

[20] M. Campana, F. Lamiraux, and J. P. Laumond, "A gradient-based path optimization method for motion planning," 2016. [Online]. Available: https://homepages.laas.fr/mcampana/drupal/sites/homepages.laas.fr.mcampana/files/u87/paper2.pdf

[21] J. Carpentier, S. Tonneau, M. Naveau, O. Stasse, and N. Mansard, "A versatile and efficient pattern generator for generalized legged locomotion," in *To appear in Proc. of IEEE Int. Conf. on Robot. and Auto (ICRA)*, Stockholm, Sweden, May 2016.

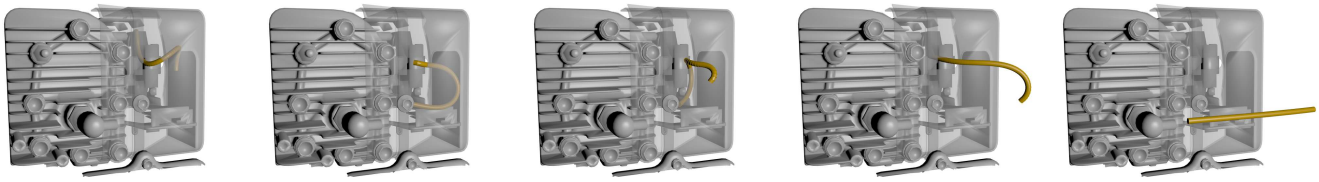
[22] S. Tonneau, N. Mansard, C. Park, D. Manocha, F. Multon, and J. Pettré, "A reachability-based planner for sequences of acyclic contacts in cluttered environments," in *Int. Symp. Robotics Research (ISRR), (Sestri Levante, Italy), September 2015*, 2015.

[23] S. Tonneau, A. D. Prete, J. Pettré, C. Park, D. Manocha, and N. Mansard, "An efficient acyclic contact planner for multiped robots," *Submitted to The Int. Journal of Robot. Research (IJRR)*, 2016.

[24] N. M. Amato, O. B. Bayazit, L. K. Dale, C. Jones, and D. Vallejo, "Choosing good distance metrics and local planners for probabilistic roadmap methods," *IEEE T. Robotics and Automation*, vol. 16, no. 4, pp. 442–447, 2000.

[25] N. Mansard, F. Valenza, and J. Carpentier, "Pinocchio: Fast forward inverse dynamic for multibody systems," 2014. [Online]. Available: http://stack-of-tasks.github.io/pinocchio/index.html

[26] O. Roussel, A. Borum, M. Taïx, and T. Bretl, "Manipulation planning with contacts for an extensible elastic rod by sampling on the submanifold of static equilibrium configurations," *IEEE International Conference on Robotics and Automation (ICRA 2015)*, May 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01096543



Start configuration Goal configuration

Fig. 9: Extraction of a deformable elastic rod from an engine. The green rod shows the goal configuration, the yellow shows the motion computed from the initial configuration(left picture). Engine model courtesy of Siemens-KineoCAM.

APPENDIX

Listing 2: Python implementation of a RRT

```
class MotionPlanner:
  def __init__ (self, robot,
                problemSolver):
    self.robot = robot
    self.ps = problemSolver

  def solveBiRRT (self, maxIter = float("inf")):
    self.ps.prepareSolveStepByStep ()
    finished = False

    nbCC = ps.numberConnectedComponents ()
    iter = 0
    while True:
      #### RRT begin
      qrand = robot.shootRandomConfig ()
      newConfigs = list ()

      for i in [0,1]:
        ## Extend connected components
        qnear, dist = ps.getNearestConfig (qrand, i)
        pathFullyValid, i_path =
                        ps.directPath (qnear, qrand)
        l = ps.pathLength (i_path)
        qnew = ps.configAtParam (i_path, l)
        ps.addConfigToRoadmap (qnew)
        newConfigs.append (qnew)
        ps.addEdgeToRoadmap (qnear, qnew, i_path, True)

      ## Try connecting the new nodes together
      for i in range (len(newConfigs)):
        for j in range (i):
          pathFullyValid, i_path =
        ps.directPath (newConfigs[i], newConfigs[j])
          if pathFullyValid:
            ps.addEdgeToRoadmap (
        newConfigs[i], newConfigs[j], i_path, True)

      #### RRT end
      ## Check if the problem is solved.
      nbCC = ps.numberConnectedComponents ()
      if nbCC == 1:
        # Problem solved
        finished = True
        break
      iter = iter + 1
      if iter > maxIter:
        break
    if finished:
        self.ps.finishSolveStepByStep ()
```